

The Design of Web APIs

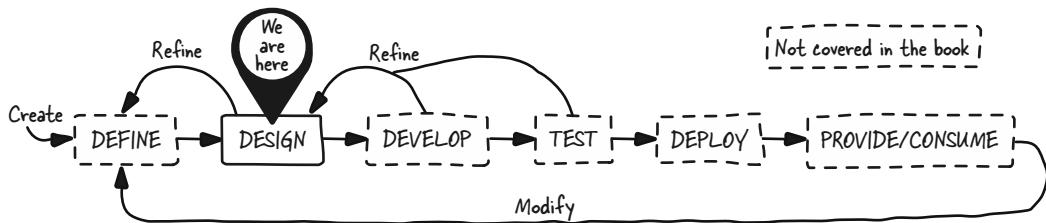
SECOND EDITION

Arnaud Lauret

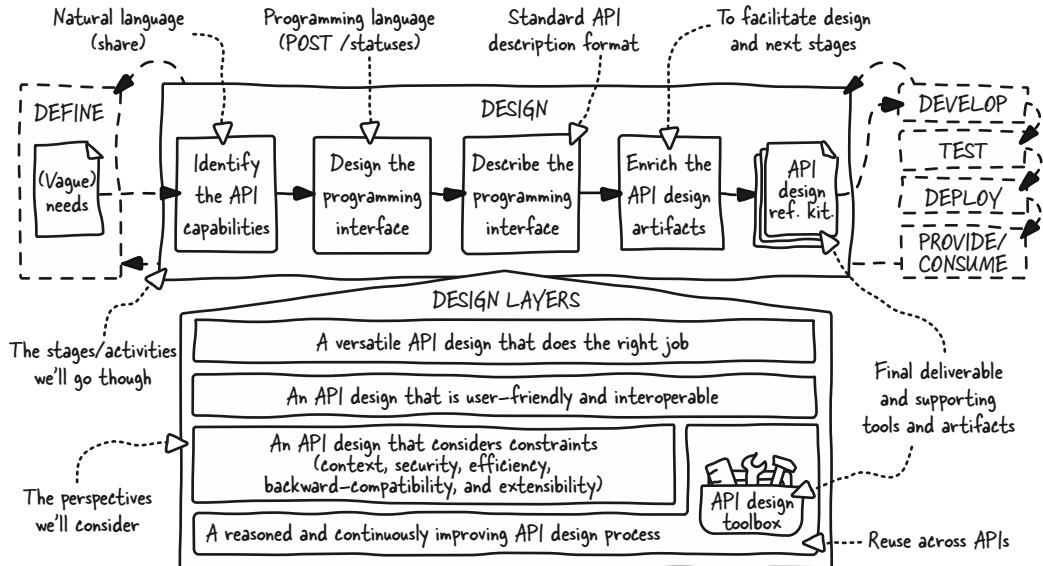
Foreword by Kin Lane



MANNING



This book focuses on the Design stage of the API lifecycle. It doesn't cover business or IT strategies for creating an API, defining its objectives, or desired business or IT outcomes. The book also excludes implementation code, architecture, tests, and deployment concerns, such as API developer portal resources like documentation. However, it explains how other stages can use the work and artifacts created during Design.



This book's methodology breaks down the design process in a step-by-step, layered approach to address one main problem at a time, facilitating learning and execution.

Praise for the first edition

Stop scouring the internet for information, it's in this book! A great resource for all skill levels on designing a Web API.

—Shayn Cornwell, Senior Software Consultant at XeroOne Systems LLC

Answers nagging and complicated questions with a simple philosophy, and never tries to hide anything from you. A fantastic introduction to the field.

—Bridger Howell, Software Engineer at Social Finance

A strong, structured, and well documented resource the community lacked.

—George Onofrei, Web Developer at DevEx Solutions

A journey from novice to professional for developing Web APIs that are robust, friendly, and easy to consume.

—Mohammad Ali Bazzi, Lead Software Architect at Seek

A must-read for API programmers and architects.

—Sanjeev Kumar Jaiswal, Lead Security Engineer at Gainsight

Finally, a systematic approach to API design.

—Javier Collado, Backend Engineer at Constructor.io

The Design of Web APIs, Second Edition

ARNAUD LAURET
FOREWORD BY KIN LANE



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


©2025 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

∞ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The authors and publisher have made every effort to ensure that the information in this book was correct at press time. The authors and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Marina Michaels
Technical editor: Jeremy Glassenberg
Review editor: Aleksandar Dragosavljević
and Kishor Rit
Production editor: Andy Marinkovich
Copy editor: Tiffany Taylor
Proofreader: Jason Everett
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781633438149
Printed in the United States of America

To all API designers: it's dangerous to go alone! Take this.

contents

<i>foreword</i>	<i>xxvii</i>
<i>preface</i>	<i>xxix</i>
<i>acknowledgments</i>	<i>xxxi</i>
<i>about this book</i>	<i>xxxiii</i>
<i>about the author</i>	<i>xxxvii</i>
<i>about the cover illustration</i>	<i>xxxviii</i>

1 What is API design? 1

1.1 What is a web API? 2

A web API is a remote interface for applications 2 ▪ *A web API uses the HTTP protocol* 3 ▪ *A web API is an interface to an implementation* 4 ▪ *A web API is an interface for others* 5

1.2 Why does the design of any API matter? 6

What if a terrible API was a kitchen appliance? 6 ▪ *Poor web API design affects developers and architecture* 7 ▪ *Poor web API design puts security and infrastructure at risk* 8 ▪ *Poor web API design affects end-user and third-party experiences* 9 ▪ *Taking care of design unleashes the power of APIs* 9

1.3 When to design web APIs? 10

Any new API must be designed 10 ▪ *Any modification of any existing API must be designed* 11 ▪ *Design happens between choosing to create an API and coding it* 11

- 1.4 Who designs web APIs? 12
The three profiles needed to design an API 13 ■ The stakeholders influencing API design 13
- 1.5 How do we design web APIs? 14
- 1.6 Designing APIs step by step 15
Identifying the API capabilities 15 ■ Designing the programming interface 15 ■ Describing the programming interface 16
Enriching API design artifacts 16
- 1.7 Designing APIs layer by layer 16
Designing a versatile API that does the right job 16 ■ Designing a user-friendly and interoperable API 17 ■ Integrating constraints in an API design 17 ■ Using a reasoned and continuously improving design process 18

PART 1 FUNDAMENTALS OF API DESIGN21

2 Identifying API capabilities 23

- 2.1 An overview of identifying API capabilities 24
Starting with the output of the Define stage 25 ■ Analyzing what users need to achieve 26 ■ Identifying versatile operations to achieve use cases 26 ■ Keeping programming interface design concerns for later 27 ■ Clarifying the subject matter and input 27
- 2.2 Introducing the API Capabilities Canvas 28
How does the API Capabilities Canvas work? 29 ■ Tools to use along with the API Capabilities Canvas 30
- 2.3 Walking the nominal paths 31
Identifying users 32 ■ Listing use cases 32 ■ Decomposing use cases in steps 33 ■ Determining inputs and success outcomes 33
Spotting missing elements with sources and usages 34 ■ Analyzing the spotted elements 34
- 2.4 Walking the alternative and failure paths 35
Analyzing failures for each step 36 ■ Adding alternative branches on each use case 37 ■ Analyzing the alternative users and use cases 38
- 2.5 Refining steps to identify operations 38
Differentiating steps and operations 38 ■ Identifying unique and versatile operations 39

- 2.6 Focusing on the proper needs 40
 - Staying within the Define stage's needs scope* 40 ▪ *Focusing on the proper perspectives* 41 ▪ *Asking why to investigate any problem* 41
- 2.7 Avoiding integrating too specific consumers' perspective 41
 - Avoiding mapping consumers' UI* 42 ▪ *Avoiding integrating consumers' business logic* 42
- 2.8 Avoiding exposing the provider's perspective 43
 - Avoiding exposing the provider's data organization* 43 ▪ *Avoiding exposing the provider's business logic* 44 ▪ *Avoiding exposing the provider's software architecture* 45

3 *Observing operations from the REST angle* 49

- 3.1 An overview of programming interface design 50
 - Introducing the basics of the HTTP protocol* 51 ▪ *Introducing the basics of REST APIs* 52 ▪ *Contrasting REST with non-HTTP-compliant web APIs* 53 ▪ *How do we design a REST programming interface?* 53 ▪ *Why not discuss HTTP and REST when identifying capabilities?* 54
- 3.2 Observing the API Capabilities Canvas from the REST angle 55
 - Reorganizing and expanding the API capabilities canvas* 56
 - How to observe operations from the REST angle* 57
- 3.3 Identifying resources and their relations 58
 - What is a resource?* 58 ▪ *Identifying an operation's resource* 58
 - Tweaking an operation's description to identify resource* 59
 - Identifying resource relations* 60 ▪ *Using patterns and recipes to identify resources and relations* 60
- 3.4 Identifying resource actions 61
 - What is an action, and how can it be identified?* 61 ▪ *Listing an action's inputs* 62 ▪ *Dealing with the operation's resource when listing an action's inputs* 63 ▪ *Listing an action's outputs* 64
 - Dealing with contradictory successes and failures when listing outputs* 65

4 *Representing operations with HTTP* 69

- 4.1 Representing operations with HTTP 70
 - What an operation looks like with HTTP* 71 ▪ *How to represent operations with HTTP* 72

- 4.2 Representing resources with paths 73
 - What is a resource path?* 73
 - *Designing meaningful resource paths* 74
 - *Targeting specific elements with path parameters* 74
 - Showing resource relationships with a hierarchy* 75
 - *Representing lists and their elements* 76
- 4.3 Representing actions with HTTP methods 77
 - Determining which HTTP methods to use* 77
 - *Choosing HTTP methods to represent actions* 78
 - *Representing search, read, and delete actions* 78
 - *Representing update actions* 79
 - Representing create actions* 79
 - *Mapping typical operations to HTTP* 80
- 4.4 Choosing input data locations in HTTP requests 80
 - Where to put input data in an HTTP request* 80
 - *An overview of input data natures* 81
 - *Choosing a location for resource identifiers* 82
 - Choosing a location for resource representations* 82
 - *Choosing a location for resource modifiers* 83
 - *Hesitating between resource identifiers and modifiers* 83
 - *Choosing input data locations for typical operations* 84
- 4.5 Representing output types with HTTP statuses 84
 - What is an HTTP status?* 84
 - *Choosing HTTP statuses for outputs* 85
 - *Choosing successful HTTP statuses for read operations* 86
 - *Choosing successful HTTP statuses for delete operations* 86
 - *Choosing successful HTTP statuses for update operations* 87
 - *Choosing successful HTTP statuses for search operations* 87
 - *Choosing successful HTTP statuses for create operations* 87
 - *Choosing error HTTP statuses* 87
 - *Ensuring exhaustive error-handling* 88
 - Choosing HTTP statuses for typical operations* 89
- 4.6 Choosing output locations in HTTP responses 90
 - Where to put data in an HTTP response* 90
 - *Filling the output data gaps* 90
 - *Choosing output locations* 91
 - *Choosing output data locations for typical operations* 92
- 4.7 Representing a “do” operation with HTTP 92
 - Using an action resource* 93
 - *Turning the action into a business concept* 93
 - *Focusing on the result* 94
- 4.8 Using the REST architectural style principles for API design 94
 - Introducing the REST architectural style* 94
 - *Applying REST principles to API design* 95
 - *Debates about what is (or is not) REST* 96

5 *Modeling data* 99

- 5.1 An overview of data modeling 100
 - Which data are we modeling?* 102 ▪ *Introducing the JSON portable data format* 103 ▪ *Modeling data* 104
- 5.2 Designing theoretical resource data models 105
 - Determining a resource's structure* 105 ▪ *Choosing an object resource's properties* 106 ▪ *Choosing a property name and type* 107 ▪ *Indicating required properties* 107 ▪ *Listing and modeling properties efficiently* 108
- 5.3 Designing inputs and outputs data models 108
 - Designing a read operation's inputs and success outputs* 108
 - Designing a search operation's inputs and success outputs* 109
 - Designing a create operation's inputs and success outputs* 111
 - Designing an update operation's inputs and success outputs* 112
 - Designing a delete operation's inputs and success outputs* 113
 - Designing a temporary error data model* 113
- 5.4 Streamlining input and output data modeling 113
 - Designing and using the complete, summarized, minimal, and identifier models* 114 ▪ *Designing and using the creation, replacement, and modification models* 115 ▪ *Modeling data for "do" operations* 116 ▪ *Differentiating similarly named elements* 116
- 5.5 Using data to ensure completeness and proper focus 117
 - Spotting missing elements by analyzing input sources and output usages* 117 ▪ *Ensuring complete business error-handling* 117
 - Focusing on the proper elements* 118

6 *Describing HTTP operations with OpenAPI* 122

- 6.1 Overview of describing the programming interface 123
 - Introducing the OpenAPI Specification* 124 ▪ *Using OpenAPI during design* 125 ▪ *Introducing the YAML format* 126
 - Contrasting an OpenAPI document with our API spreadsheet* 127
 - Describing the programming interface while designing it* 128
- 6.2 Authoring OpenAPI documents 129
 - Introducing the specification-first and code-first approaches* 129
 - Contrasting the specification-first and code-first approaches* 130
 - Picking an OpenAPI editor* 131 ▪ *Choosing an OpenAPI version* 132 ▪ *Choosing between JSON and YAML* 132
- 6.3 Describing HTTP operations with OpenAPI 132

- 6.4 Describing resource paths 133
 - Initiating an OpenAPI document 134* ▀ *Describing a path 135*
 - Describing a path with path parameters 135*
- 6.5 Describing operations 136
- 6.6 Describing operation inputs 138
 - Describing query parameters and other non-body parameters 138*
 - Describing request bodies 139*
- 6.7 Describing operation output HTTP status codes 140
 - Describing an output case type with an HTTP status 141*
 - Dealing with outputs sharing the same HTTP status code 142*
- 6.8 Describing operation output contents 142
 - Describing response bodies 143* ▀ *Dealing with responses without bodies 144* ▀ *Describing response headers 144*

7 Describing data with JSON Schema in OpenAPI 148

- 7.1 An overview of describing data 149
 - Introducing JSON Schema 149* ▀ *Contrasting OpenAPI and JSON Schema with our API spreadsheet 150* ▀ *Describing data while designing it 151*
- 7.2 Authoring a JSON Schema data model in OpenAPI 152
- 7.3 Adding complete resource data models to the OpenAPI document 152
 - Choosing a location for the resource model in the OpenAPI document 152* ▀ *Initiating the resource model description 153*
- 7.4 Describing complete resource data models with JSON Schema 153
 - Describing an object 154* ▀ *Adding properties to an object 154*
 - Describing an atomic property 155* ▀ *Describing an object property 157* ▀ *Describing an array property 158* ▀ *Stating which properties are required 158*
- 7.5 Describing operation input and output data 160
- 7.6 Describing operation non-body data 161
 - Describing non-body request parameters with inline schemas 161*
 - Tweaking non-atomic parameter serialization 162* ▀ *Describing response headers with inline schemas 162*
- 7.7 Describing operation body data 163
 - Using references to resource models in response bodies 163*
 - Deriving the complete resource model to create other reusable*

models 165 ▪ *Using references to resource models in request bodies* 167 ▪ *Mixing inline schema and reference* 167

PART 2 USER-FRIENDLY, INTEROPERABLE API DESIGN ...173

8 *Designing user-friendly, interoperable data* 175

- 8.1 The user-friendliness and interoperability layer of API design 176
 - Overview of the API user experience* 177 ▪ *Which users' experiences matter to us?* 177 ▪ *How API design user-friendliness and interoperability affect UX* 178
- 8.2 What makes data user-friendly and interoperable? 179
 - User-friendly data meets user needs* 179 ▪ *User-friendly data helps us find and interpret information* 179 ▪ *User-friendly data limits consumers' work* 180 ▪ *User-friendly data is consistent* 180
 - Interoperable data is consistent and standard* 181
- 8.3 When and how to design user-friendly, interoperable data 181
 - Which data must be user-friendly and interoperable?* 182 ▪ *When to address user-friendly, interoperable data* 182 ▪ *How to design user-friendly, interoperable data* 183
- 8.4 Selecting and crafting ready-to-use data 183
 - Choosing simple and meaningful but useful data* 184
 - Adding supporting data to ease and secure interpretation* 185
 - Adding processed data to reduce consumer effort* 185 ▪ *Choosing well-known or standard resource identifiers* 185 ▪ *Choosing well-known or standard data* 186
- 8.5 Choosing user-friendly, interoperable atomic types and formats 186
 - Considering formatting numbers as strings* 187 ▪ *Managing non-human-readable codes* 187 ▪ *Managing dates and times* 188
- 8.6 Organizing data 189
 - Grouping data with objects* 189 ▪ *Grouping data with arrays* 191 ▪ *Sorting data in arrays and objects* 192
- 8.7 Choosing data granularity and scope 192
 - Considering relevance, not size* 192 ▪ *Embedding lists in a resource model* 193 ▪ *Modeling embedded resources* 194

- 8.8 Designing user-friendly names 195
 - When to design user-friendly names* 195 ■ *Designing simple, clearly organized, concise names* 195 ■ *Learning by fixing non-user-friendly names* 196
- 8.9 Aiming for consistency and standardization 197
 - Seeking local, domain, or global standardization* 197 ■ *Using well-known or standard identifiers consistently* 198 ■ *Defining a naming pattern for identifiers* 198 ■ *Naming, typing, and structuring consistently* 199

9 Designing user-friendly, interoperable operations 203

- 9.1 What makes operations user-friendly and interoperable? 204
 - User-friendly operations expose clear capabilities that meet the needs* 204 ■ *User-friendly operations use user-friendly data and are helpful* 204 ■ *User-friendly, interoperable operations are consistent and standard* 205
- 9.2 When and how to design user-friendly, interoperable operations 205
 - When to take user-friendly, interoperable operations into consideration* 206 ■ *How to design user-friendly, interoperable operations* 206
- 9.3 Designing easy-to-understand, guessable operations 207
 - Combining meaningful resource paths and HTTP compliance* 208
 - Creating predictable resource paths* 208 ■ *Crafting short but accurate resource paths* 209
- 9.4 Requesting easy-to-provide inputs 210
 - Using typical and HTTP-compliant input locations* 210
 - Mapping inputs to outputs* 212 ■ *Requesting well-known or standard data* 212 ■ *Minimizing inputs with default and server-processed data* 213
- 9.5 Returning ready-to-use successful responses 214
 - Choosing adequate HTTP status and HTTP-compliant data locations* 214 ■ *Returning sufficiently informative data* 215
- 9.6 Filtering, sorting, and paginating lists 216
 - Designing guessable filters that map returned data* 216
 - Designing flexible filters* 217 ■ *Enabling free search and complex logic with a *q* filter* 217 ■ *Minimizing filters* 218 ■ *Enabling sort with helpful defaults* 218 ■ *Paginating lists* 218
 - Returning filter, sort, and pagination metadata* 219

- 9.7 Adapting request and response data 220
 - Handling different data formats* 220 ▪ *Translating data and adapting to locale* 222 ▪ *Tweaking returned data* 222
- 9.8 Handling consumer errors gracefully 223
 - Limiting consumer errors* 223 ▪ *Using adequate HTTP status codes* 224 ▪ *Providing informative, problem-solving feedback* 224 ▪ *Returning machine-readable feedback* 225 ▪ *Returning an exhaustive list of errors* 225 ▪ *Using standards* 226
- 9.9 Avoiding hiding multiple capabilities in a single operation 227
 - Reconsidering request and response data granularity* 227
 - Reconsidering an operation's purpose* 228
- 9.10 Aiming for consistency and standardization 228
 - Using standardized data consistently* 229 ▪ *Adopting standardized behavior consistently* 229 ▪ *Offering standardized features consistently* 230

10 *Designing user-friendly, interoperable operation flows* 235

- 10.1 What makes an operation flow user-friendly and interoperable? 236
 - Using user-friendly, interoperable elements* 236 ▪ *Being designed as a whole* 236 ▪ *Being concise and flexible* 237 ▪ *Meeting user needs within the flow* 237 ▪ *Being helpful across operations* 238 ▪ *Aiming for consistency and standardization* 238
- 10.2 When and how to optimize flows 238
 - When to consider flow optimization* 239 ▪ *How to optimize flows* 239
- 10.3 Designing concise, error-limiting, flexible flows 240
 - Introducing the money-transfer use case* 240 ▪ *Uncovering operation flow problems* 241 ▪ *Calling read and search operations once* 242 ▪ *Enhancing operations with use-case-specific features* 242 ▪ *Adding use-case-specific operations* 243 ▪ *Combining operations into a use-case-specific operation* 243 ▪ *Adding use-case-specific output data* 244 ▪ *Avoiding constraining consumer flow* 245
- 10.4 Designing flexible data-saving flows 246
 - Introducing the "Open an account" use case* 246
 - Understanding how data-saving constrains consumer flow* 247
 - Enabling partial data-saving* 247 ▪ *Carefully aggregating saving*

operations 248 ■ *Smoothing validation and separating it from completion* 249 ■ *Enabling full and partial data-saving flows* 249 ■ *Redirecting the consumer to the finalized resource* 250

11 *Designing user-friendly, interoperable APIs* 253

11.1 What makes an API user-friendly and interoperable? 254

Having a clear purpose that meets focused needs 254 ■ *Enabling discovery and navigation* 255 ■ *How to create user-friendly, interoperable APIs* 255

11.2 Creating one or multiple APIs 255

When to discuss API granularity 256 ■ *Identifying independent sets of operations* 256 ■ *Keeping in mind that sub-APIs can be related* 257

11.3 Clarifying the API's purpose with its name 258

When to choose an API name 258 ■ *Choosing an API name* 258
Adding the API name to the API base path 258

11.4 Enabling interoperable API browsing with HTTP and hypermedia APIs 259

Listing a resource's operations with the OPTIONS HTTP method 260 ■ *Providing pagination, formats, and resources links with the Link header* 260 ■ *Using hypermedia formats for relations and actions* 261 ■ *Using content negotiation to select hypermedia or plain JSON format* 262 ■ *Ensuring that subject matter data is always available* 262 ■ *Considering browsing capabilities* 263

PART 3 CONSTRAINED API DESIGN267

12 *Designing a secure API* 269

12.1 Overview of API security 270

What happens during an API call? 270 ■ *Uncovering design-related API security problems* 272

12.2 When and how to handle security during design 273

When to consider security during API design 273 ■ *How API design contributes to API security* 274

12.3 Exposing only the necessary data and operations 275

What are sensitive operations and data? 275 ■ *Challenging sensitive and non-sensitive data and operations* 276 ■ *Modifying*

	<i>data to make it less sensitive or non-sensitive</i>	276	▪	<i>Splitting an operation to separate concerns</i>	277	▪	<i>Separating sensitive operations in dedicated APIs</i>	277
12.4	Ensuring that implemented operations behave according to context	278						
	<i>Describing who sees or does what</i>	278	▪	<i>Describing what list or search operations return</i>	279	▪	<i>Describing how inputs narrow access</i>	279
	<i>Describing all expected implementation checks and behaviors</i>	280	▪	<i>Narrowing access by design</i>	280			
12.5	Ensuring data integrity	281						
	<i>Corrupting data with regular API calls</i>	281	▪	<i>Correctly implementing HTTP methods</i>	282	▪	<i>Preventing request replay</i>	282
	<i>Enabling and enforcing conditional updates</i>	283						
12.6	Avoiding protocol- or architecture-based security problems	284						
	<i>What may not be secured on an API call over HTTPS</i>	284						
	<i>Dealing with sensitive search parameters</i>	285	▪	<i>Dealing with sensitive resource IDs</i>	286	▪	<i>Integrating data encryption or signing in the design</i>	286
12.7	Limiting consumer access with scopes	287						
	<i>Limiting access to an operation with a scope</i>	287	▪	<i>Measuring the importance of scopes and their design</i>	288			
12.8	Designing scopes	289						
	<i>Creating operation-based scopes</i>	289	▪	<i>Creating resource-, concept-, or use-case-based scopes</i>	290	▪	<i>Creating scopes for read or write operations</i>	291
	<i>Creating end-user- or consumer-based scopes</i>	291	▪	<i>Tweaking operation behavior with scopes</i>	292			
	<i>Deciding which scope types to use</i>	292						
12.9	Describing scopes with OpenAPI	292						
	<i>Defining scopes</i>	292	▪	<i>Using scopes</i>	293			
12.10	Erroring securely	294						
	<i>Handling token-related errors</i>	294	▪	<i>Handling missing scopes or permissions</i>	294	▪	<i>Avoiding disclosing implementation details on server errors</i>	295
	<i>Providing implementation details in response descriptions in OpenAPI</i>	296	▪	<i>Enforcing expected error data with JSON Schema</i>	297			

13 *Designing an efficient API* 302

- 13.1 An overview of API efficiency 303
 - How an API can be inefficient* 303 ▪ *When to be concerned about efficiency* 304 ▪ *How design contributes to API efficiency* 305
- 13.2 Optimizing the design only when necessary 305
 - Ensuring HTTP configuration efficiency* 305 ▪ *Limiting API usage with rate-limiting* 306 ▪ *Enhancing response with rate-limiting headers* 307 ▪ *Finding the true root cause* 307
- 13.3 Focusing on user needs and user-friendliness to be efficient 307
 - What we've learned so far* 308 ▪ *Analyzing an inefficient flow* 308 ▪ *Optimizing each operation* 309 ▪ *Rethinking the flow* 310
- 13.4 Enabling caching and conditional readings 311
 - An overview of caching and conditional readings* 311 ▪ *Not letting consumers decide how to cache* 312 ▪ *Defining caching policies based on data and context* 312 ▪ *Returning cache directives* 313 ▪ *Retrieving data only when modified* 314
- 13.5 Optimizing data volume 314
 - Enabling resource model selection* 315 ▪ *Toggling the return of updated or created resources* 316 ▪ *Enabling field selection* 316 ▪ *Centralizing redundant data in dedicated operations* 317 ▪ *Considering a partial update over total replacement* 317 ▪ *Contrasting JSON Merge Patch and JSON Patch for array updates* 318
- 13.6 Optimizing pagination 319
 - Optimizing page size limits* 319 ▪ *Choosing cursor- or index-based pagination* 320
- 13.7 Processing multiple elements with bulk or batch operations 320
 - Designing bulk operation requests* 321 ▪ *Optimizing bulk operation requests* 322 ▪ *Clarifying a bulk operation error policy* 322 ▪ *Designing a mixed response* 323 ▪ *Designing an all-or-nothing response* 324 ▪ *Optimizing bulk request responses* 324 ▪ *Partitioning access to bulk operations* 325
- 13.8 Considering a separate optimized API 325

14 *Adapting the API design to the context* 330

- 14.1 Integrating context into the API design 331
 - How context can affect the design of an API* 332 ▪ *Seeking constraints and limitations during design* 333 ▪ *Challenging constraints and limitations* 333 ▪ *Making trade-offs* 334
- 14.2 Dealing with consumer and provider constraints 334
 - Working around consumer HTTP method limitations* 334
 - Accommodating consumers who are used to different data formats* 335 ▪ *Managing planned interruptions* 336
 - Ensuring data and URL compatibility* 336 ▪ *Implementing partial updates* 337
- 14.3 Handling data and files 337
 - Collecting data and files in a flow* 338 ▪ *Sending data and files with a single call* 338 ▪ *Retrieving data and files with a single call* 340 ▪ *Describing files with OpenAPI* 340 ▪ *Describing mixed data and files with OpenAPI* 341
- 14.4 Providing efficient file management features 343
 - Returning file data only when necessary* 343 ▪ *Enabling partial downloads and uploads* 343 ▪ *Preventing unnecessary uploads* 344
- 14.5 Delegating file downloads and uploads 345
 - Downloading files from another system* 345 ▪ *Uploading files to another system* 346
- 14.6 Notifying consumers about provider-sourced events with a webhook 347
 - What is a webhook, and why should we consider using one?* 347
 - Webhooks should be optional* 348 ▪ *Designing a webhook operation* 348 ▪ *Using a standard event format* 348
 - Choosing event data granularity* 349 ▪ *Designing a secure webhook* 350 ▪ *Defining the expected webhook behavior* 351
 - Dealing with webhook failures* 351 ▪ *Describing a webhook with OpenAPI* 352
- 14.7 Handling long operations 353
 - Starting a long operation and monitoring its status with polling* 353 ▪ *Using a callback API to avoid polling* 354
 - Describing a callback with OpenAPI* 354 ▪ *Choosing an execution mode with the Prefer header* 356
- 14.8 Considering other API types 356
 - Introducing REST API alternatives* 356 ▪ *When to select an API type* 357

15 *Modifying an API* 361

- 15.1 An overview of API modification concerns 362
 - What can happen when modifying an API?* 363 ■ *Uncovering API design modification concerns* 363 ■ *How to design API modifications* 364
- 15.2 Identifying breaking changes and ensuring backward compatibility 365
 - Modifying output data* 365 ■ *Modifying input data* 367
 - Modifying resource paths* 370 ■ *Modifying operations or their HTTP methods* 370 ■ *Modifying HTTP statuses* 371
 - Modifying operation flows* 371 ■ *Being aware of the invisible contract* 372 ■ *Preventing unintended modifications* 372
- 15.3 Identifying security-breaking changes and preventing breaches 373
- 15.4 Assigning a version to an API 373
 - Differentiating interface and implementation versioning* 374
 - Choosing an API version identifier* 375 ■ *How the API version can be represented in a request* 376 ■ *Choosing how to represent the API version in a request* 377 ■ *When to choose an API version scheme and representation* 377 ■ *Avoiding sub-API-level versioning* 377
- 15.5 Carefully breaking and versioning an API 378
 - Listing consumers and their types* 379 ■ *Checking whether consumers use what we break* 379 ■ *Determining whether it's possible to expose multiple API versions* 379 ■ *Complying with the API versioning policy* 380 ■ *Balancing effects and benefits of breaking changes* 380 ■ *Accumulating trade-offs or breaking regularly* 380
- 15.6 Creating extensible API designs 381
 - Designing a user-friendly, interoperable REST API that does the job* 381 ■ *Learning from past decisions* 382 ■ *Using extensible design patterns* 383 ■ *Providing deprecation runtime information* 384
- 15.7 Describing the design modifications with OpenAPI 384
 - Indicating the API version* 384 ■ *Deprecating elements* 385
 - Adding a changelog* 386

PART 4 SCALED AND SIMPLIFIED API DESIGN393

16 *Facilitating API design decision-making* 395

- 16.1 Making design decisions confidently and consistently 396
 - Ensuring that it's the right time to make a decision* 397
 - Evaluating the scope of the decision* 397 ▪ *Deciding based on trusted past decisions* 397 ▪ *Deciding based on trusted external sources* 398 ▪ *Backing decisions with reasoning and sourced information* 398 ▪ *Explaining out loud* 398
- 16.2 Researching solutions to API design questions 399
 - Where to research solutions to design questions* 399 ▪ *Searching and considering* 400 ▪ *Using an architectural decision record format* 400
- 16.3 What are API design guidelines? 402
 - How design guidelines can help us* 402 ▪ *How API design guidelines relate to API governance* 403 ▪ *When do we need design guidelines?* 403
- 16.4 What to put in user-friendly API design guidelines 403
 - Listing principles and rules* 404 ▪ *Providing actionable recipes* 404 ▪ *Providing ready-to-use artifacts and tools* 405
 - Helping with the API design process* 405 ▪ *Adding implementation or architecture considerations* 405
- 16.5 How to build API design guidelines 406
 - Starting with basic API design guidelines* 406 ▪ *Considering existing APIs* 407 ▪ *Expanding the guidelines when new questions arise* 407 ▪ *Ensuring that each rule brings value* 408
 - Carefully modifying API design guidelines* 408

17 *Optimizing an OpenAPI document* 411

- 17.1 An overview of OpenAPI document optimization 412
- 17.2 Defining consistent data models 413
 - Reusing schemas* 413 ▪ *Defining subschemas* 414 ▪ *Targeting part of a schema with a deep reference* 414 ▪ *Overriding descriptions when using a \$ref* 415 ▪ *Creating unique read-and-write models* 416 ▪ *Defining a complete schema from its summary* 418 ▪ *Considering schema optimizations* 419
- 17.3 Defining consistent parameters 419
 - Using path-level parameters* 419 ▪ *Reusing parameters* 420
 - Defining reusable groups of query parameters* 421
- 17.4 Defining consistent request bodies 422

- 17.5 Defining consistent responses 423
 - Reusing response headers* 423 ▪ *Reusing responses* 424
- 17.6 Ensuring cross-API consistency with external shared components 426
 - Defining a library of reusable components* 426 ▪ *Using a shared component in an API* 427 ▪ *Ensure that library files are editable independently* 427
- 17.7 Enhancing API design guidelines 428

18 Automating API design guidelines 432

- 18.1 What API linting is and how it can help us 433
 - Detecting API design and OpenAPI authoring problems* 434
 - Applying guidelines seamlessly and concentrating on user needs* 435
- 18.2 Using an API linter to automate API design guidelines 435
 - Developing linting rules to automate guidelines* 435 ▪ *Using our automated guidelines while designing APIs* 436 ▪ *Choosing an API linter* 436
- 18.3 Introducing Spectral 437
 - Linting an OpenAPI document with Spectral CLI* 437 ▪ *How Spectral lints an OpenAPI document* 438 ▪ *Editing Spectral rulesets* 438
- 18.4 Deciding what API linting rules verify 439
 - Using our guidelines to create only needed rules* 439 ▪ *Finding small problems to solve* 440 ▪ *Simplifying rules with shared OpenAPI components* 441 ▪ *Ensuring appropriate granularity with a concise name and description* 441
- 18.5 Targeting elements to check in the OpenAPI documents 442
 - Starting rule development by targeting the proper elements* 442
 - Targeting any element in the OpenAPI document* 443 ▪ *Dealing with references to local or shared components* 445 ▪ *Creating a library to target typical elements* 445
- 18.6 Checking element values 447
 - Performing basic checks on values and keys* 447 ▪ *Ensuring that an element is defined* 448 ▪ *Ensuring that an element is not defined* 449 ▪ *Checking references* 450 ▪ *Checking partial JSON schemas* 451 ▪ *Performing cross-element checks* 452

- 18.7 Returning helpful feedback when problems are detected 454
 - Stating the importance or nature of a problem with a severity* 455
 - Returning problem-solving message* 456 ■ *Splitting rules due to severity or message concerns* 457
- 18.8 Organizing rules 457
- 18.9 Using our automated guidelines when designing APIs 458
 - Importing and tweaking the guidelines ruleset* 458 ■ *Ignoring certain problems* 459

19 *Enriching API design artifacts* 463

- 19.1 Crafting an API design reference kit 464
 - What an API design reference kit can contain* 465 ■ *Using the kit to design the API* 466 ■ *Using the kit to develop the API* 466
 - Using the kit to test the API* 466 ■ *Using the kit to deploy the API* 466 ■ *Using the kit to provide and consume the API* 467
 - What we already have and what we may want to add* 467
- 19.2 Providing an overview of the API design with OpenAPI 468
 - Adding links to other artifacts and describing the API* 468
 - Organizing operations around concepts and use cases* 469
 - Describing use cases* 470
- 19.3 Enhancing the precision of data models with JSON Schema 471
 - Describing a number or element size range* 471 ■ *Describing a value with pattern, enum, and default* 472
- 19.4 Providing examples to illustrate data and operations 473
 - Adding property examples with JSON Schema* 474 ■ *Adding examples of parameters, request and response bodies, and headers with OpenAPI* 475 ■ *Authoring accurate and realistic examples* 475 ■ *Sharing OpenAPI examples across operations* 476 ■ *Connecting examples to each other* 477
- 19.5 Enhancing and adapting artifacts for implementers 478
 - Embedding implementation notes in artifacts* 478 ■ *Enhancing or adapting OpenAPI for code generation* 479
- 19.6 Considering API mocking or prototyping during API design 479
 - Creating a basic mock with OpenAPI* 480 ■ *Favoring an early prototype over a complex mock during design* 480

19.7	Considering creating functional API tests during API design	480
	<i>Clarifying logic</i>	481
	<i>Smoothing collaboration</i>	481
	<i>Designing standard APIs</i>	481
<i>appendix</i>	<i>Solutions to the exercises</i>	485
	<i>index</i>	523

foreword

The discipline of web API design has been maturing for well over a decade, and each edition of *The Design of Web APIs* by Arnaud Lauret provides a milestone marker for this collective API journey. To help you properly absorb and apply what you are about to read, you need to understand that the seeds of the specifications and techniques put forth by Arnaud in this book emerged between 2010 and 2015, when he began to help properly document this movement in character as the API Handyman. There is nobody better to learn API design from than Arnaud Lauret, and no better time to be implementing it.

I personally watched Arnaud dissect HTTP, OpenAPI, JSON Schema, and the other essential building blocks of the API economy between 2015 and 2019. As the API Handyman, he went deep down the API design rabbit hole and returned with the first edition of *The Design of Web APIs*. Now Arnaud has done it again, aligning his expertise in the pragmatic real-world design of the digital resources and capabilities that are shaping the global business landscape with what enterprises need to tame the API sprawl they face in 2025.

It does not matter if your teams are producing internal, partner, or public APIs. Your engineering teams are likely both design-first and code-first when it comes to delivering web APIs. *The Design of Web APIs* in 2025 is your guide to modernizing your digital supply chain, factory floor, and distribution channels. Most traffic on the World Wide Web is web API traffic, and this is the ubiquitous approach to defining digital resources and capabilities and delivering modern web, desktop, mobile, device, and now artificial intelligence application experiences. The second edition of *The Design of*

Web APIs is your handbook not just to design APIs but also to lay a solid foundation for the governance of those APIs at scale across teams.

There are two types of enterprise organizations today: those investing in ensuring that their product and engineering teams have the skills they need to design, develop, deliver, and sustain modern web APIs; and those outsourcing the heart of their businesses to cloud software vendors. *The Design of Web APIs* is the book you need to equip both your product and engineering teams to deliver consistent and standardized web APIs at scale. This book will help your teams master the most common standardized aspects of web APIs and will equip them with the skills they need to design and deliver the parts of your operations that are unique and proprietary to the way your enterprise does business.

I have been working exclusively with web APIs since 2010, and Arnaud—the API Handyman—and *The Design of Web APIs* are always my go-to for the API design knowledge I need to produce APIs and support my customers in producing the hundreds or thousands of APIs they need to conduct business each day. Without proper design of web APIs, each new application or integration inside or outside the enterprise becomes exponentially more expensive and time-consuming to make happen—increasing the total cost of ownership for every API. If you are just starting your business and are looking for guidance to make your business supply chain, factory floor, and distribution channels efficient, or you’ve been in business for many years and would like to get a handle on API sprawl, *The Design of Web APIs* has the answers you are looking for.

—KIN LANE, The API Evangelist

preface

My career has spanned more than two decades, mostly in finance, and during that time I have worked on connecting software across networks using technologies like FTP, Sun RPC, CORBA, Java RMI, SOAP, and web APIs. I have complained about terrible internal and third-party APIs (web services, RPC, etc.) and created awful ones myself. I've witnessed how flawed API design causes confusion, prolonged development, brittle code, rising technical debt, wasted resources, production crashes, and security breaches.

As technology evolved, connecting software became easier, especially with web APIs. The rise of “as a service” products and successful public APIs like Twilio and Stripe in the 2010s raised expectations for API design and developer experience; sending SMS messages and money with simple code was a game-changer. It transformed my approach to software and API design: why, I wondered, couldn't I always have such a fantastic experience? However, even after the API hype started, many private and public web APIs were neglected, just as their predecessors had been. They were often seen as mere technical plumbing, and well-meaning creators often fell into common traps that I also encountered.

Attending my first API conference (API Days Paris in late 2014) made me want to share all I had learned. I created my API Handyman blog, began speaking at conferences, and wrote the first edition of *The Design of Web APIs*, published in 2019. I originally intended to explore web API design with REST, GraphQL, and gRPC, but the book became too lengthy and complex. To teach design principles, I chose to focus on REST APIs, as they are widely used and rooted in solid principles; this helped me

address key design issues like meeting needs, usability, security, performance, and modifications, which apply to all APIs.

In 2025, the first edition of this book is still relevant. Web APIs remain vital across all industries and are essential for distributed systems, web apps, mobile apps, cars, kitchen appliances, other applications, and AI. REST APIs are still the most commonly used. And the numerous API design reviews, workshops, training sessions, and API analyses I've conducted make it clear that API design still needs to be taught. Many, including AI, struggle to understand what designing an API entails and how to create effective APIs.

So, I quickly agreed to work on a second edition when asked. Since the first edition, I've gained valuable insights and wanted to include my new and revised knowledge in the book. The first edition can be compared to *Terminator* and the second to *Terminator 2: Judgment Day*: bigger and better. The core story remains (teaching web API design and focusing on REST APIs, not saving us from Skynet), but it's a complete rewrite with more depth and tons of new content. For example, it features an API design process that ties the book together, emphasizes the importance of communication with other stakeholders, highlights interoperability as part of usability, simplifies design decisions, discusses building guidelines and their automation, and includes 70 exercises to reinforce key concepts. My past self would have avoided many problems if I'd had this book to use as a resource; I hope you find it useful, too!

acknowledgments

Writing this second edition was even more challenging than the first; I wouldn't have achieved such a great book alone. I want to thank everyone who made this odyssey possible.

First and foremost, I want to thank my wife, Cinzia, and my daughter, Elisabetta. You have always supported me and encouraged me. I love you so much.

Next, I want to thank everyone at Manning Publications, starting with my editor, Mike Stephens; thank you for believing in the book's first and second editions. To my development editor, Marina Michaels: thank you for your help, support, and feedback. Thanks also to my technical editor, Jeremy Glassenberg, for his invaluable feedback. And thank you to all the Manning production folks—production manager Aleksandar Dragosavljević, production editor Andy Marinkovich, graphics supervisor Azra Dedic, copyeditor Tiffany Taylor, proofreader Jason Everett, typesetter Dennis Dalinnik, and cover designer Marija Tudor—for their efforts in getting this book ready for publication.

Thank you to all the reviewers for their invaluable and detailed feedback: Adam Hörömpöli, AJ Bhandal, Akinwale Habib, Alceu Rodrigues de Freitas Junior, Amol Gote, Andrei Tarutin, Andres Sacco, Ashwini Gupta, Asif Iqbal, Becky Huett, Dileep Kumar Pandiya, Elias Rangel, Emmanouil Chardalas, Ernesto Cardenas Cangahuala, Gabriele Bassi, Harini Shankar, Jeremy Caney, José Alberto Reyes Quevedo, Kalyanasundharam Ramachandran, Kristina Kasanicova, Lakshminarayanan A. S, Lov Lalwani, Malik Novruzov, Mihaela Barbu, Mikhail Malev, Mwiza Kumwenda, Naga Rishyendar Panguluri, Nakul Pandey, Nghia To, Palak Mathur, Richard Meinsen,

Saidaiah Yechuri, Salahuddin Zaki, Shantanu Kumar, Sridhar Rao Muthineni, Sumit Bhatnagar, Tede Morgado, Walter Alexander Mata López, Werner Nindl, and Zorodzayi Mukuya. Your input helped make this a better book.

Thank you to everyone who provided invaluable encouragement and feedback at various stages. A very special *merci beaucoup* to Joyce Stack for her feedback on the early manuscript of the second edition. Also, thanks to all readers of the first edition, especially Isabelle Reusa and Mehdi Medjaoui, who field-tested and reviewed the early manuscript.

Finally, thanks to all the API practitioners I have met; I learned a lot from you. Thanks to Mike Amundsen, Kin Lane, and Mehdi Medjaoui (again) for their encouragement and help when I started the API Handyman blog in 2015 and after. A special thanks to Ivan Goncharov, who, in 2017, forwarded an email from Manning Publications seeking an author for an API book that later became *The Design of Web APIs*. This book wouldn't exist without all of you.

about this book

The Design of Web APIs, Second Edition was written to help you design new web APIs or modify existing APIs so that they do the right job; are versatile, secure, and efficient; address contextual constraints; and facilitate future changes. To do so, this book uncovers all aspects of API design and equips you with the mindset, processes, and tools to efficiently do your job in the long run and at scale, working on many APIs and with other API designers.

Who should read this book?

The Design of Web APIs, Second Edition, is, obviously, for anyone who directly designs web APIs, but also for people involved in their creation and use. They could be developers, business analysts, technical writers (involved in the creation of server applications, microservices, and backends for mobile applications or websites), tech leads, architects, API governance experts (working at scale on many APIs), or API product owners (who want to ensure the best possible developer experience). Additionally, developers using APIs, QA engineers testing APIs, technical writers documenting APIs, and security experts requesting modifications may be interested in understanding how APIs should be designed so they can give constructive feedback to their API providers.

How this book is organized: A roadmap

This book has 4 parts, 19 chapters, and an appendix. Chapter 1 is an introduction to the entire book. It establishes a shared understanding of web APIs and web API design and outlines the design process and practices we'll learn in the following chapters.

Part 1 teaches the fundamentals of designing a versatile API that does the right job:

- Chapter 2 explains how to identify the capabilities an API must offer to meet the requirements exhaustively and adequately.
- Chapter 3 introduces REST APIs and teaches how to observe API capabilities to identify the elements needed to design a REST API: resources, their relations, and their operations.
- Chapter 4 explores how to represent operations with HTTP, including resource path design, choosing HTTP methods and HTTP statuses, and selecting locations for data in HTTP requests and responses. It also discusses the REST architectural style and its benefits for API design.
- Chapter 5 discusses modeling data, including resources, path parameters, query parameters, and request and response headers and bodies.
- Chapter 6 shows how to describe HTTP operations using the OpenAPI Specification.
- Chapter 7 explains how to describe data in OpenAPI documents with JSON Schema.

Part 2 focuses on designing user-friendly, interoperable APIs that developers can use quickly and seamlessly without complex thinking and coding. It covers these concerns at the data, operation, sequence of operations, and API levels, with one chapter for each:

- Chapter 8 introduces the concepts of user-friendliness and interoperability for APIs and then focuses on data. It explains how to choose, define, type, organize, and name data so that it is ready to use, consistent, and standard.
- Chapter 9 explains how to make operations clear and guessable; have easy-to-provide inputs and ready-to-use outputs; enable pagination, filtering, and sorting; and handle errors gracefully.
- Chapter 10 shows how to design concise, error-limiting, flexible sequences of operations.
- Chapter 11 discusses creating one or multiple APIs, naming APIs, and enabling API browsing with HTTP and hypermedia.

Part 3 explains how security, efficiency, data, architecture, business, and modification concerns can constrain our ideal user-friendly, interoperable design that does the right job:

- Chapter 12 covers designing secure APIs, including data sensitivity, secure operation behavior, data integrity, and controlling access with scopes.
- Chapter 13 focuses on efficient API design that doesn't bother end users or negatively impact infrastructure. It discusses data volume optimization, caching, processing multiple elements, and considering separate optimized APIs.

- Chapter 14 explores how data, architecture, and business affect our design. It discusses handling files, long operations, webhooks, and types of APIs other than REST.
- Chapter 15 discusses modifying an API, including how not to break consumers, versioning, and extensible design.

Part 4 aims to make our API designer job easier and more sustainable in the long term and at scale, when we're working on many APIs with colleagues:

- Chapter 16 explains how to make design decisions confidently and create user-friendly API design guidelines.
- Chapter 17 discusses optimizing OpenAPI documents for consistency and simplified authoring, including defining reusable elements shared across APIs.
- Chapter 18 describes automating guidelines to ensure consistency and free our minds of details. It illustrates this with Spectral, an API linter.
- Chapter 19 discusses enhancing the API design artifacts we've created to build a design reference kit that streamlines our work, ensures accurate implementation, and supports the following steps of the API lifecycle.

The online appendix contains the solutions to the exercises in the book. I encourage you to solve them before reading their solutions, which include detailed explanations, references to relevant sections, and additional comments. It is available in the ePDF, ePUB, and liveBook versions of the book, as well as via download on the book product page at www.manning.com/books/the-design-of-web-apis-second-edition.

This book should be read from cover to cover, in order. Each new chapter expands on what has been learned in previous ones. But after you finish chapters 1–7, you can jump to any chapter that covers a topic you urgently need to investigate.

About the code

This book contains many examples of source code, both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text.

You'll find detailed examples of OpenAPI and JSON Schema in chapters 6, 7, and 11–19. Chapter 18 also contains examples of Spectral rules; you'll need NodeJS and Spectral version 6 to run them.

You can copy the listings' source code from the liveBook (online) version of this book at <https://livebook.manning.com/book/the-design-of-web-apis-second-edition>. The complete code for the examples in the book is available for download from the Manning website at www.manning.com/books/the-design-of-web-apis-second-edition and in this book's repository, available at <https://github.com/arno-di-loreto/design-of-web-apis-2e>. Both contain the following:

- An example of the API Capabilities Canvas from chapter 2 in Google Sheet (via a link), Excel, and Open Document formats

- All OpenAPI, JSON Schema, and Spectral code listings with complete comments and full versions of elements that are truncated in the book
- Spectral and OpenAPI files and magic `npm run <section number>` commands to quickly run the Spectral rules in each section of chapter 18

liveBook discussion forum

Purchase of *The Design of Web APIs, Second Edition* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/the-design-of-web-apis-second-edition/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website for as long as the book is in print.

Other online resources

If you want to learn more about API design and beyond, I recommend doing the following:

- Subscribe to the *API Developer Weekly* newsletter at <https://apideveloperweekly.com>.
- Read posts by Kin Lane, the API Evangelist, at <https://apievangelist.com>.
- Read my API Handyman blog at <https://apihandyman.io>.

about the author



ARNAUD LAURET is a French software architect and tech enthusiast with over two decades of experience. He has spent most of his career in finance, connecting systems in various ways, including web APIs. Currently, he is involved in API tooling and works as an API Industry Researcher at Postman. For the last decade, he has focused on web APIs—designing, building, and analyzing APIs, and guiding others in these areas while investigating the tools, practices, and challenges shaping the API world across industries. He shares his knowledge on the API Handyman blog and at conferences worldwide. Passionate about human-centered design, he strives to create systems that provide excellent experiences for everyone involved, from those who build and run them to end users.

about the cover illustration

The figure on the cover of *The Design of Web APIs, Second Edition* is captioned “Girl from Drniš, Dalmatia, Croatia.” The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian’s retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

What is API design?

This chapter covers

- What are web APIs?
- Why web API design matters
- When to design web APIs
- Who designs web APIs
- How to design web APIs

Think about your past experiences with web APIs. Have you struggled with cryptic or inconsistent naming and typing, poor data organization, or missing data or operations? Spent hours debugging a failed call that returned a vague “Bad Request”? Have you dealt with crashes due to data being removed in API responses? Maybe end users complained about slow performance, only for you to find an API call returning an entire database table. Or worse, have you discovered an API exposed sensitive data inappropriately? The root cause is often the same: poor API design.

The design of any web API is crucial, whether it is used by thousands of third parties or a few internal developers. Poorly designed APIs can negatively affect developers’ productivity, an IT system’s performance and integrity, end users’ experience, and an organization’s revenue. Well-designed APIs give developers and IT

systems superpowers and make end users happy, helping individuals, teams, departments, or organizations achieve their IT-system-dependent goals faster and more efficiently. In this book, we'll introduce you to techniques and tools that will help you design web APIs that developers will love to use.

1.1 **What is a web API?**

Our interconnected world relies heavily on web APIs (application programming interfaces), which enable communication between applications over a network. They are essential for websites, mobile applications, and IT systems with different software components working together. Web APIs can provide simple programmatic access to any server application with a few lines of code, enabling the remote triggering of any action the application can perform: for example, reading or updating a customer's address, getting a server's health information, sending an SMS, or detecting kittens in a video. APIs don't require knowledge of the underlying code and logic; as long as they are authorized to (API security is discussed in section 12.1), anyone can use them, not just their creators. This section clarifies what a web API is in the context of this book by discussing four key aspects. A web API is

- A remote interface for applications
- An HTTP protocol-based interface
- An interface to an implementation
- An interface we create for others

1.1.1 **A web API is a remote interface for applications**

A web API enables one application (server, backend, or provider) to expose functions or operations that other applications (clients or consumers) can use, call, or consume remotely over a network. Figure 1.1 illustrates typical scenarios with the fictitious SOCNET social network (replace with your favorite one).

To share a message on SOCNET, a user of its mobile application takes a photo, types a message, and taps the Share button. The mobile application then calls the “Share” operation of SOCNET's backend web API, sending the message and photo over the internet. The backend detects friends in the photo and stores the message by invoking internal server applications via their web API over the local network. Other users can view the shared message via the SOCNET application or website, which calls the “Timeline” operation. SOCNET also has a Health Check server application that runs in the cloud, regularly calling the “Timeline” operation to ensure that everything functions correctly.

This example shows that we can retrieve, send, or process data via a web API. However, web APIs can also affect the real world beyond data. For example, a web API call can hail a taxi or turn a smart lightbulb on or off.



NOTE A web API enables communication between mobile or web applications and their backends and between server applications (typically in micro-service or similar distributed architectures). A web API can provide multiple

operations. Multiple applications can access the same web API. An application that provides a web API can also consume others. An application consuming an API may or may not have end users.

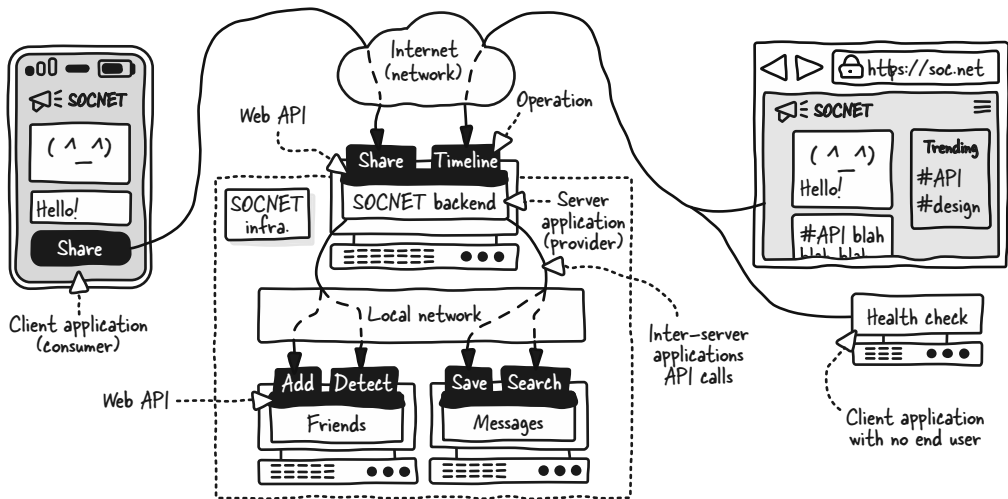


Figure 1.1 The SOCNET web API exposed by the backend has operations that can be called by any kind of application over the internet. The Friends and Messages internal applications also expose web APIs that can be called over the local network.

1.1.2 A web API uses the HTTP protocol

When an application calls a web API, it uses HTTP (Hypertext Transfer Protocol), the same thing a web browser uses to retrieve an HTML page of a website; that's the origin of the “web” in web APIs. Figure 1.2 shows a call to the “Share” operation of the SOCNET web API to illustrate what it looks like; we'll keep the details for section 3.1.

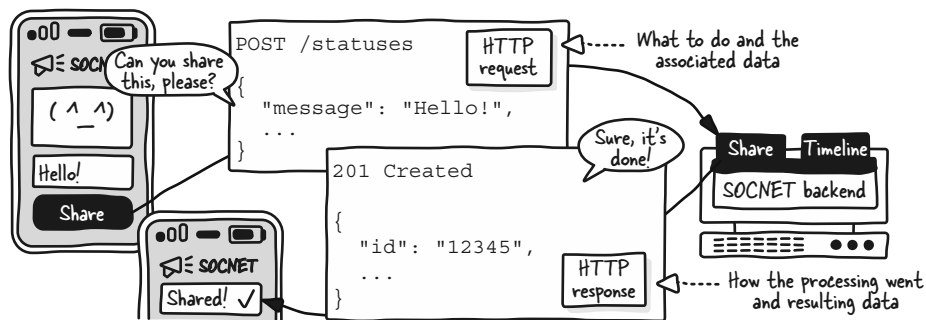


Figure 1.2 When calling a web API, the consumer sends an HTTP request indicating what to do and the needed data. Once the request is processed, the server returns an HTTP response indicating how the processing went and the resulting data.

When using an HTTP-based API, the consumer sends a request containing information that describes the action to be performed and the associated data needed to execute the action. The server returns a response indicating whether the processing went well and the resulting data. Different types of web APIs exist, such as REST, SOAP, GraphQL, and gRPC (no worries if those names mean nothing to you); they use HTTP differently. The figure illustrates a typical REST API call, the type of API we focus on in this book.



NOTE This book focuses on REST APIs because they are the most commonly used type of web API and rely on solid architectural principles (discussed in section 4.8). These principles can benefit any type of web (and non-web) API. Additionally, many design principles of this book apply to other APIs. However, a REST API is not always the adapted solution; alternatives will be briefly discussed in section 14.8.

1.1.3 A web API is an interface to an implementation

The term *Web API* is often used to refer to an application that exposes an API, but it represents only part of the application. A web API is an interface to an implementation and (ideally) conceals implementation details. As shown in figure 1.3, an application providing a web API can be compared to a restaurant and its consumers to the customers.



NOTE The term *API consumer*, or *consumer*, can refer to an application consuming the API, the developers who create it, and their organization. Similarly, the term *API provider*, or *provider*, refers to the application exposing the API, the developers who create it, and their organization.

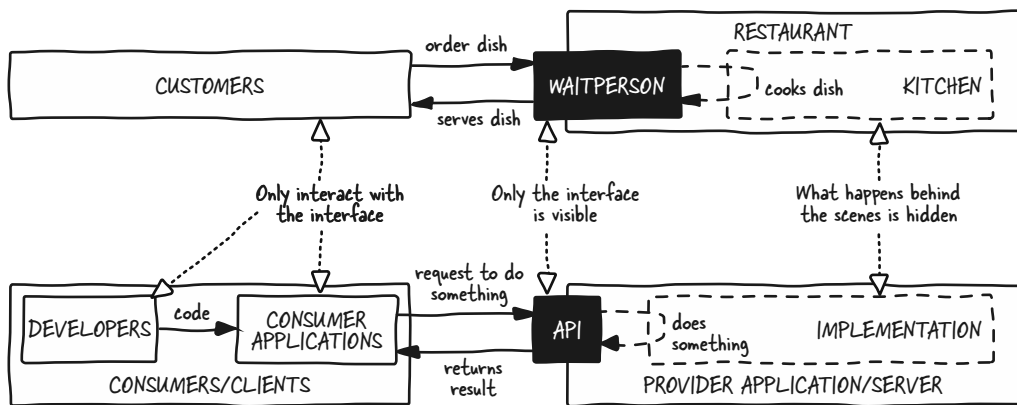


Figure 1.3 In a restaurant, we interact only with the waitperson without knowing what's happening in the kitchen. An API acts similarly, hiding what's happening in the implementation.

The API is the waitperson who takes our order and delivers it to us. The implementation represents everything that occurs in the kitchen. When we order our dish, we

only interact with the waitperson and don't need to know who is in the kitchen, the recipe, the ingredients, how the dish is cooked, or even if it's cooked there.

SOCNET mobile application developers blissfully ignore API implementation details when coding the API call to the “Share” operation triggered by the Share button's tap. They don't need to know if microservices are used under the hood, which database is used, how the data is organized in it, or how friends are identified in the photo. They only need to code an HTTP call (which all programming languages support) and send the relevant data (a message and an optional photo). The SOCNET backend API implementation handles the rest.



NOTE This book focuses on the design of web API interfaces (what is visible to consumers) and doesn't discuss their implementation (how they are architected and coded). This book's API design principles are applicable regardless of the architecture and programming language chosen for the implementation. However, what happens in the implementation may influence the design of an API (see section 14.1).

1.1.4 A web API is an interface for others

Web APIs can be used by others in addition to their creators; they foster seamless collaboration within an organization and with external users. As shown in figure 1.4, the SOCNET mobile and web teams use the API created by the backend team. To deliver a robust face-detection feature, the Friends API team uses the Face Detection API from the Image Processing as a Service (IPaaS) company (for which they pay a subscription). Additionally, SOCNET has set up a Search API for selected partners (willing to pay for access to more data).

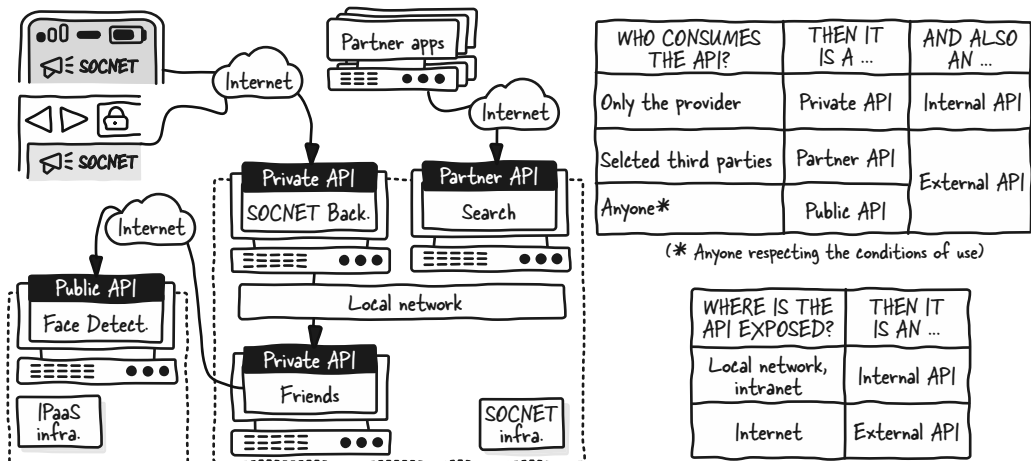


Figure 1.4 The terms *internal*, *external*, *private*, *partner*, and *public* API may need to be disambiguated so everyone involved understands each other.

The APIs involved here represent three levels of API openness or closeness: private (Backend and Friends APIs), partner (Search API), and public (Face Detection API). *Internal* and *external* may be used to qualify private and partner/public APIs, respectively. These terms may also indicate that an API is exposed on a local network or the internet.

Most web APIs are private, and hundreds of millions exist because any company with an IT system must connect its different applications. A private API may be used by several teams in the organization or only by the team that created it. Even APIs created for our team are used by “others”: newcomers, those who didn’t design them, or our six-month-old self.

Many organizations use partner or public APIs from commercial or open source software or “as-a-service” companies. These APIs are present in any domain or sector, such as payroll, retail, financial services, logistics, telecommunications, project management, software development, cloud infrastructure, or AI. Even not-so-digital companies and government agencies are offering partner or public APIs. Whatever we need, there’s an API for that.

1.2 Why does the design of any API matter?

What APIs can do, what they look like, and their behavior, and hence their design, can have terrible consequences or invaluable benefits for API consumers, their end users, and the API providers. To explain why the design of any private, partner, or public API matters, this section first illustrates fundamental API design flaws using an analogy to better realize how absurd APIs can be. Then we return to the software world to describe some consequences of poorly designed APIs and some benefits of well-designed APIs using quotes from internal or external users of the SOCNET APIs.

1.2.1 What if a terrible API was a kitchen appliance?

Transposing the flaws we may find in APIs to an everyday object can help us and others realize how absurd the design of our APIs can be and why. The Kitchen Radar in figure 1.5 has an interface that doesn’t help us determine what it is or how to operate it. Pressing the MAG. button appears to start it, but it stops when we release the button.

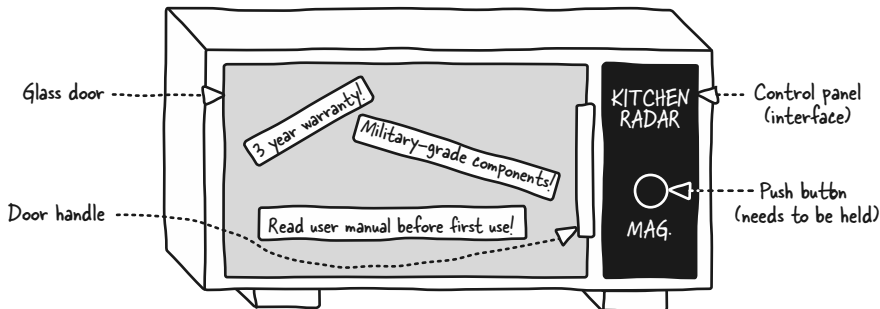


Figure 1.5 The cryptically named Kitchen Radar has an unclear purpose and a complicated user interface that is cumbersome to use.

According to the user manual shown in figure 1.6, the Kitchen Radar is named “Radar” for historical reasons; its actual purpose is to heat food. Holding the MAG. button turns on a magnetron, heating food with microwaves. To adjust the heating power, users repeatedly hold the button for a duration specified in the “Heating power cheat sheet” and then release it for the same duration until they feel it is enough.

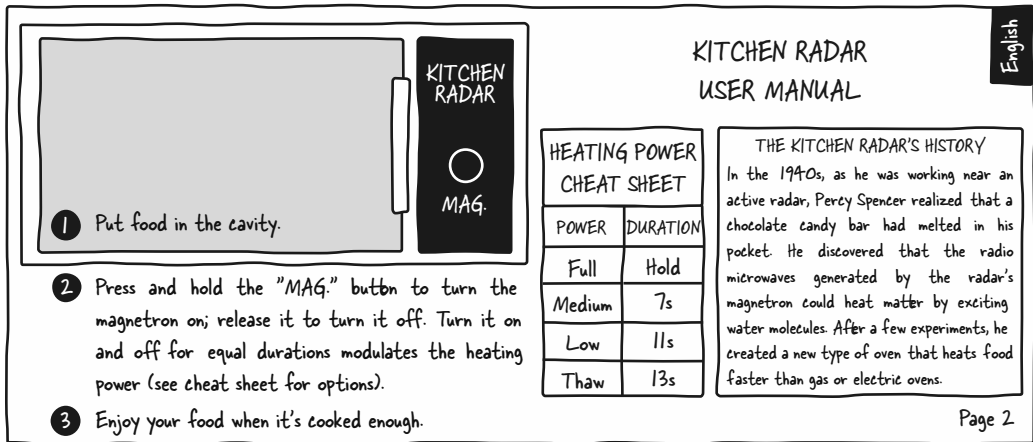


Figure 1.6 The Kitchen Radar's user manual clarifies its purpose and explains (poorly) how to operate it but doesn't make it easier to use.

The Kitchen Radar is a microwave oven that offers a frustrating user experience. Its main flaw is that it fails to address user needs; people want to heat food, not turn on a magnetron. The user manual doesn't make it user-friendly. The inside-out interface forces users to become magnetron experts and time themselves pushing and releasing the cryptic MAG. button. There are also potential reliability and safety concerns regarding the magnetron and circuitry when turned on and off at an erratic speed. Finally, we can wonder whether people would buy or use such a terrible product unless forced to.

We all agree that this appliance is absurd. Yet I've encountered private, partner, and public APIs whose designs resemble the Kitchen Radar. Such poorly designed APIs affect API consumers, their end users, and API providers.

1.2.2 Poor web API design affects developers and architecture

The design of APIs can make the work of application developers who consume them impossible or more complicated than it should be, possibly resulting in code and overall software architecture that are complex and brittle. If SOcNET neglected the design of its APIs, developers of applications consuming them might say

- "I can't list friends of friends!"
- "What contains the `sts` property?"

- “Why don’t `createdAt` and `fromDate` use the same date-time format?”
- “Identifying friends requires a `userId`, but storing a message requires a `username`! Can’t we use the same user ID in all operations?”
- “The ‘List friends’ operation is useless; to get useful data, I must call the ‘Read friend’ operation for each friend!”
- “The HTTP response indicates a success, but its data contains an error!”
- “How can I know what’s wrong with my API call if I only get an ‘Invalid request’ error message?”
- “Are you sure about the mobile and web applications taking care of friend identification with the Face Detection API before sharing a message with photos?”

Due to poor API design, developers will waste time deciphering cryptic operations, data, and errors or face unnerving problems due to inconsistencies or wrong HTTP usage. Incorrect or missing data or operations complicate code, possibly making it impossible to develop a desired feature. An API that exposes inner workings leads to error-prone and duplicated code across consumers and tight coupling between client and server, necessitating risky updates whenever what should have been inner logic changes. Missed milestones, fewer features, frequent bugs, increased technical debt, and revenue loss: poorly designed APIs affecting developers can cost an organization a lot.

1.2.3 *Poor web API design puts security and infrastructure at risk*

The design of an API can make it unsecured and can make an infrastructure unreliable and costly, as pointed out in the following feedback from consuming application developers:

- “Is it normal for user A to read the direct messages of user B?”
- “As I handle friend detection, I could send false identification information when sharing a message with a photo!”
- “Sometimes, calling the ‘Timeline’ operation takes ages.”
- “Sometimes, the ‘Timeline’ operation’s response is a Java stack trace indicating an out-of-memory error. I guess it’s because it returns all user messages; an active user can have thousands.”

Exposing inner workings increases the risk of corrupting data or processes if an essential API operation call is missed or the wrong data is sent. Unclear definitions of security rules during design can lead to an unsecured implementation that grants access to data that consumers or end users shouldn’t access. Leaking infrastructure information simplifies hackers’ work. API security problems can harm people and our organization, possibly leading to its bankruptcy.



NOTE Web APIs typically restrict access to authorized consumers, often allowing only a subset of operations (call “Search” message but not “List direct” message). Even if an operation is permitted, the end user’s privileges can limit execution or affect responses (“List direct” message returns only the end

user's messages). This book focuses on design-related security problems; see section 12.1 for more details.

Large data volumes—due, for example, to the absence of pagination in an operation design—can cause slow responses, server application crashes, or high cloud infrastructure egress fees (what cloud providers charge for transferring data). API design can also lead to inefficient use of server resources or third-party APIs, raising infrastructure costs or blocking the system because limits are reached.

1.2.4 Poor web API design affects end-user and third-party experiences

The design of an API can affect the end users of our application powered by our private APIs or third parties consuming our partner or public APIs, as illustrated by the following feedback:

- “The application crashes when showing the timeline of very active users!”
- “How can I help the end user fix the problem if the API error message is just ‘Invalid request’?”
- “Can’t we find a way to reduce the number of steps end users have to take to send a message in the mobile or web application?”
- “The application crashes when calling the ‘Timeline’ operation since the last API update!”
- “Send \$10 to a friend, and they’ll receive \$1,000!”
- “I’m fed up! I cancel my subscription to the Search API.”

The design of an API can affect end-user experience with unexpected crashes, unhelpful error messages, or complex or inflexible UI flows. Careless, non-backward-compatible modifications made to the design of existing APIs can cause unexpected errors (because `sts` has been renamed `status`) or, worse, silent behavior changes (turning dollars into cents when sending money to a friend). If end users are customers, they may switch to our competitors. If end users are internal, these problems will complicate their work and affect our customers. Developers potentially interested in our public or partner API may pass by or quickly cancel their subscriptions because of our API's complexity or unreliability. Less customers means less revenue for the organization.

1.2.5 Taking care of design unleashes the power of APIs

Taking care of API design not only prevents the problems described in the previous sections but also unleashes the true potential of web APIs. The following feedback from internal and external users of SOCNET's well-designed APIs illustrates this:

- “The Search API provides exactly the operations and data I need!”
- “It’s amazing! I was able to guess how the API works without reading the docs!”
- “I realized I could reuse many pieces of code when using different APIs!”
- “I developed the new SOCNET application for smart refrigerators in no time!”

- “I quickly aggregated different APIs into my server application!”
- “The API’s clever error handling helped us enhance the user experience!”
- “We’re far more responsive because we stopped returning all data on the timeline!”
- “We quickly improved the subscription funnel thanks to the API flexibility!”
- “Three partner integrations were done this week; it’s a new record!”
- “We replaced our in-house face-detection system with a third-party API without affecting our existing applications!”

Well-designed APIs give developers superpowers, improve their productivity, and help an organization achieve its goals faster. That’s because these APIs meet user needs, hide inner workings, and are user-friendly. They are also interoperable, facilitating data exchange and connection between systems; their operations and data are easy to use together. These APIs also foster the creation of modular, decoupled systems that are easy to evolve, simplifying developers’ work even further.

Well-designed APIs significantly enhance the overall experience for end users by contributing to efficiency and ease of use within applications. Their flexibility also enables improving client applications, especially the UI, without modifying the API.

In the case of a partner or public API, a good design that can be used instinctively without reading documentation contributes to an invaluable “Whoa!” effect and a faster time to value (the time necessary to create something meaningful for a consumer). All this increases acquisition and retention.



NOTE Well-designed private, partner, and public APIs help organizations generate more value by increasing developer productivity, making systems more modular and efficient, reducing the time to value, and contributing to outstanding user experience. However, don’t worry if your existing APIs look like the Kitchen Radar! It’s never too late to fix them; reading this book will help.

1.3 *When to design web APIs?*

Should we design all APIs? Should modifications to APIs also be designed? When is the best time to design an API? It’s essential to consider API design

- When creating any new API
- When modifying any API
- After deciding on an API creation and before its implementation

1.3.1 *Any new API must be designed*

You’ll often see the question, “Should we focus only on designing partner or public APIs, given that they are more visible than private ones?” The answer must be a firm “No!” Ignoring private API design leads to the problems highlighted in section 1.2, affecting developments, IT systems, and the entire organization. Additionally, it will undermine future partner and public API initiatives.

You will create more private APIs for your team than for others and far fewer for partners or public use. Designing many private APIs helps build invaluable skills via practice. Although it's possible to start from scratch with partner and public APIs, it may be risky; this book can help but won't work miracles. Partner and public APIs often rely on existing private APIs; neglecting these can lead to challenges. A public or partner API implementation may conceal a private API mess (like renaming `sts` to `status`). However, addressing deeper problems such as data, operations, security gaps, or missing pagination causing out-of-memory errors will likely necessitate changes to existing APIs, which may affect current consumers.

Designing private APIs facilitates creating partner and public APIs, even enabling instant transitions in some cases. In 2002, Jeff Bezos, then-CEO of Amazon, required all teams to communicate through “service interfaces” (they weren't called APIs then) designed with external use in mind, allowing customer access anytime after creation. This strategy was key to Amazon's success.



CAUTION Amazon's instant private-to-public switch is ideal but extreme, with deep architectural and organizational implications, and can be challenging for many organizations. One valid alternative is to create partner or public API façades on solid but less polished private API building blocks (see section 13.8).

1.3.2 Any modification of any existing API must be designed

API design doesn't only matter on creation; section 1.2.4 showed that carelessly modifying an existing API design can cause unexpected crashes or more fatal silent problems. It's essential to design any modification to, ideally, not introduce non-backward-compatible changes or to introduce them knowingly after carefully considering the consequences (see section 15.1). For example, a crash due to renaming `sts` to `status` can be avoided if we can synchronize updated consumers and API deployment.

Not everyone has the luxury of starting from a blank page. APIs may already exist and may not be as well-designed as they should be. The goal is not to shame what has been done before but to ensure that increases in API design technical debt are stopped. New APIs can use a new design mindset partly inspired by existing APIs (see section 16.5.2). However, rebuilding every existing API is often pointless unless it generates enough value to cover the update. Still, nothing prevents us from following new design principles learned in this book when modifying preexisting APIs.



NOTE This book also teaches how to create an extensible API design that reduces the risk of non-backward-compatible changes (section 15.6).

1.3.3 Design happens between choosing to create an API and coding it

API design, the book's focus, differs from API development (coding, implementation) or deciding to create an API for a specific purpose; it happens between them, as illustrated in figure 1.7. The figure shows a typical API lifecycle, which isn't strictly linear and applies to any software creation methodology (agile or waterfall, for example). It

outlines the stages or activities an API undergoes from inception to consumption but oversimplifies reality; activities can occur in parallel and involve back-and-forth interactions. Depending on our role in the organization, we may participate in various stages, wearing caps other than those of API designers.

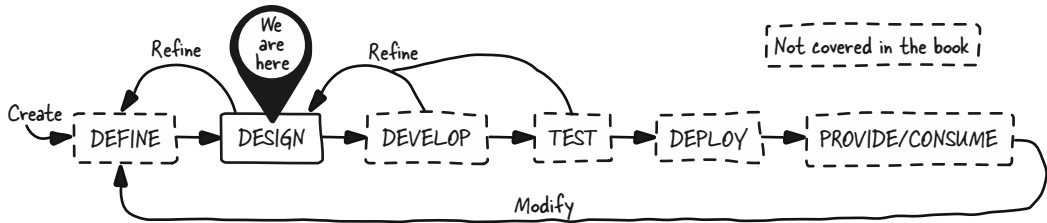


Figure 1.7 API design is an iterative activity distinct from deciding to create an API and implementing it.

Someone identifies the need for an API, such as creating a Search API to monetize SOcNET platform content or developing a microservice for friend identification in images and videos (Define). An API designer (that’s us) then designs a web API meeting these needs (Design). Developers code an application that exposes this API, which may start with an incomplete design (Develop). Throughout development, developers, QA engineers, and security experts verify that the API functions correctly and is secure (Test). An incomplete API can be deployed for testing or production, often involving exposure on an API gateway for securitization, consumption, and monitoring (Deploy). Once the final or incomplete API is deployed, consumers can use it, and it may be published to an API catalog or developer portal to make it visible (Provide/Consume).

API design is iterative. Questions and discoveries can require revisiting needs from the Define stage. Development, tests, and early consumption may help refine the API design. Once it’s in production, new needs may arise, necessitating API design modifications.



NOTE This book focuses on the Design stage of the API lifecycle. It doesn’t cover business or IT strategies for creating an API, defining its objectives, or desired business or IT outcomes. The book also excludes implementation code, architecture, tests, and deployment concerns, such as API developer portal resources like documentation. However, it explains how other stages can use the work and artifacts created during Design (section 19.1).

1.4 Who designs web APIs?

Once someone decides that an API needs to be created or modified, “we” can start working on the design. But who is “we”? It’s you, but maybe with some help. This section discusses the profiles needed to design an API and briefly lists the stakeholders who can influence API design.

1.4.1 The three profiles needed to design an API

Designing an API requires the following roles around the table; they can be one or different people:

- API designer
- Subject matter expert
- IT system expert

API designers can have various backgrounds. I've worked with, advised, and trained API designers with various profiles and experience in their fields, including developers, tech leads, architects, business analysts, tech writers, QA engineers, product managers, and product owners. The key for API designers is the ability, in the worst scenarios, to interpret vague user needs from the Define stage of the API lifecycle and address complex related subject matters (or fields, domains, business domains, topics) handled by obscure IT systems (where the API will run). The goal is to transform this complexity into an implementable HTTP-based API that meets user needs, conceals inner workings, and is user-friendly, interoperable, etc. (all that makes an API awesome, as seen in section 1.2). That's what this book will teach you.

APIs cover countless fields like banking, logistics, customer relationship management (CRM), product catalogs, and cloud infrastructure. With experience, you may become a subject matter expert (SME) in some fields, facilitating your API designer work. However, not being an SME is not a problem; as an API designer, you can effectively interview SMEs to gather information important to design an API.

This book covers essential software concerns for designing web APIs. For instance, some design patterns may drain smartphone batteries, whereas others enhance systems interoperability. However, this knowledge won't make you an expert in all the IT systems of our vast world's organizations. For instance, if you work at SOcNET, you may not know that the application responsible for detecting faces takes a minute to identify people in a 10-second video; this should be considered when designing the SOcNET backend API. Your API design skills will help you gather such information from developers, tech leads, and architects of systems you're not an expert on.

1.4.2 The stakeholders influencing API design

Some stakeholders directly or indirectly influence the design of an API. In addition to SMEs and IT experts whose information will shape the design of the API, we can add

- People in charge of the Define stage
- Consumers
- Security experts
- Peers

As an API designer, you'll discuss the design with the people who define the user needs and with consumers to ensure that the API design matches the (initially vague)

expectations. Security experts and peers may make recommendations to improve your design. You'll know how to integrate, adapt, or refuse all stakeholders' requests and feedback for the greater good of the API and your organization to design an API that satisfies all parties involved. Typically, you'll integrate security feedback without much discussion but carefully consider consumers' requests, which may lead to a highly specific API usable by only one of them.



NOTE The designer, SME, and IT expert profiles and collaboration with various stakeholders, including users, are similar to what we see when creating any application. The software design methodologies, tips, and tricks you know may help you design APIs.

1.5 How do we design web APIs?

Designing APIs mirrors any design process; it involves analyzing requirements and creating a blueprint for the final product. Figure 1.8 illustrates this book's methodology within the API lifecycle, breaking down the design process step by step while using a layered approach to address one main problem at a time. Similarly to the API lifecycle, don't view this as a strict linear waterfall; the activities and perspectives occur in parallel, with optional steps and back-and-forths within the Design stage and other lifecycle stages. Section 1.6 guides us through the design process, and section 1.7 summarizes the design layers.

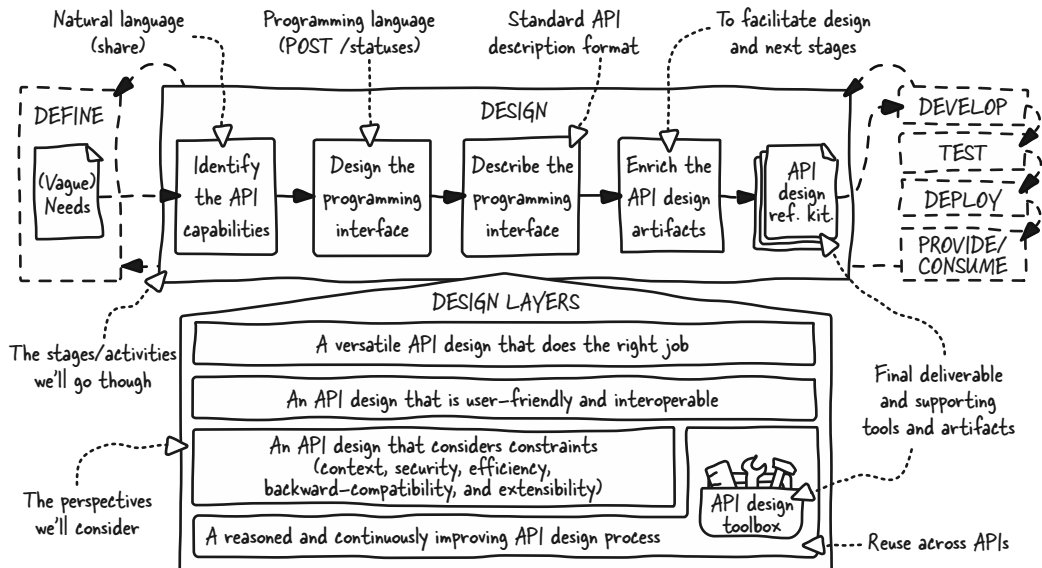


Figure 1.8 This step-by-step and layered approach aims to help us design APIs in various contexts and facilitate our learning.



NOTE Many software design and development methodologies can be applied to API design. Consider this book a toolbox; once you’ve learned its principles, adapt them to your context.

1.6 Designing APIs step by step

As we saw in figure 1.8, the Design stage starts once user needs or requirements are identified in the Define stage of the API lifecycle and is composed of four steps or activities:

- Identifying the API capabilities
- Designing the programming interface
- Describing the programming interface
- Enriching API design artifacts

1.6.1 Identifying the API capabilities

As API designers, our primary task is to analyze user needs and identify the required API capabilities to address them (discussed in chapter 2). User needs can range from broad objectives like “Social network” or “Database as a service” to specific intents such as “Enabling tagging friends in a photo on mobile and web.” Artifacts describing these needs can vary from brief sticky notes or tickets to detailed documents, including, for example, user experience research for public APIs. We express API capabilities in a stakeholder-friendly natural language like English or French. Capabilities encompass use cases, such as “Sharing a status,” which involves steps like “Upload message’s photos” and “Send the status.” We’ll identify the API operation needed for these steps, like “Upload a photo” and “Share a message.” Operations can apply to multiple use cases; for example, the “Changing user profile photo” could need the “Upload a photo” operation.

1.6.2 Designing the programming interface

We design the programming interface that represents the identified API capabilities. This book focuses on REST APIs, but the choice of API type should align with capabilities and context (section 14.8). We will prepare for our REST API design by identifying key concepts or business objects, such as User and Status (chapter 3). We’ll convert natural language operations into HTTP operations, like turning “Share a message” into `POST /statuses` (chapter 4). Additionally, we will model the input and output data for operations (chapter 5): for example, determining that a Status includes a message of type string.



NOTE Although this book focuses on REST APIs, many of its teachings can be applied to other types of APIs. We typically need to identify key concepts or model data when designing any API.

1.6.3 Describing the programming interface

Using a standard API specification format, we can efficiently describe the programming interface HTTP operations (chapter 6) and their data (chapter 7) in a blueprint document while we design them. Doing so has numerous benefits, including generating implementation code (Develop) or facilitating the connecting of the dots between the initial user needs and API capabilities in natural language and the resulting programming interface for all stakeholders (Design).

1.6.4 Enriching API design artifacts

The document listing capabilities and the API blueprint belong to the API design reference kit. This kit fully describes the API and is used during design and in the following stage of the API lifecycle, particularly to implement and test the API. In parallel with previous steps, we may consider enhancing the API blueprint with more detailed information (the Status message is 140 characters long, for example) or adding new artifacts to the kit (an API mock that simulates the yet-to-be-developed API, for example). Such enrichments help to describe the API design better and facilitate discussions and thinking (chapter 19).

1.7 Designing APIs layer by layer

As illustrated in figure 1.8, we'll split designing APIs into four layers to address one main problem at a time:

- A versatile API design that does the right job
- An API design that is user-friendly and interoperable
- An API design that considers constraints
- A reasoned and continuously improving API design process

1.7.1 Designing a versatile API that does the right job

Our first goal is not only to design an API that meets the needs identified in the Design stage of the API lifecycle but also to conceal its inner workings and ensure that the API is usable in different contexts (chapter 2). A well-designed API typically reflects a business domain, independent of the applications consuming it, although specific APIs may sometimes be required (see section 13.8). The SOcNET backend API is designed to allow the website and mobile application to deliver expected social network features, despite differing UIs, without burdening developers with face-detection algorithms or microservices architecture. SOcNET can also build a new smart refrigerator application without modifying the API. Similarly, SOcNET designs the Search partner API without knowing the types of applications that will use it.

1.7.2 Designing a user-friendly and interoperable API

In addition to doing the right job and being versatile, we must ensure that our API is user-friendly and interoperable; this affects data (chapter 8), operations (chapter 9), sequences of operations (chapter 10), and the entire API (chapter 11). Using the SOcNET APIs can be challenging if data pieces are labeled with cryptic names like `sts` and `someCrypticJargonThatMeansStatus` instead of clear and easily understood terms like `status`. An error message like “Invalid request” is insufficient; we need to provide actionable error feedback. If the Friends and Message APIs use different user IDs (username versus internal numerical ID), that compromises interoperability. The same applies if a backend API’s “List friends” operation requires a username but the `Send Message` function needs a numerical ID.

1.7.3 Integrating constraints in an API design

If we focus only on ensuring that our API design is effective, user-friendly, and interoperable, we may end up with a non-implementable, failure-prone, or, worse, dangerous API design. Designing an API requires integrating constraints dictated by security, efficiency, context, and modification-related concerns.

API security in the context of API design covers sensitive data management (like personal and banking information) and access controls (chapter 12). For instance, in section 1.2.3, a developer using the SOcNET backend API found that user A’s direct messages were accessible when they were logged in as user B. This common API security flaw often arises because “who can see what” is not clearly defined during API design.

An API’s design affects its efficiency (chapter 13). An inefficient API may be slow, consume excessive server resources, or drain smartphone batteries. For example, in section 1.2.2, developers reported the “Timeline” operation being slow and returning an out-of-memory error (it also affects end users). This is often due to attempting to return all data in a single call, which an appropriate design can prevent.

We may need to adapt API design to its specific context, including the subject matter, provider, consumer habits, and limitations (chapter 14). For instance, in section 1.4.1, we discovered that identifying friends in a video takes 1 minute for every 10 seconds of footage. This makes the identification process longer than the 100 milliseconds maximum SOcNET that allows for efficient API calls. Instead of a typical one-call design, a long operation is required, initiating a job with the first call, monitoring status with a second call, and retrieving results with a third call (section 14.7).

API design must ensure extensibility to facilitate future changes and carefully consider non-backward-compatible modifications that could break consumers (chapter 15). Following developer feedback from section 1.2.2, SOcNET renamed `sts` to `status`, which in section 1.2.4 led to crashes in unmodified applications due to missing expected data. Assessing the effect of this modification could have prevented the problem by SOcNET either deciding not to perform the modification or handling consumer updates better.

1.7.4 Using a reasoned and continuously improving design process

We must find ways to simplify our API designer's life and enhance our efficiency. Designing APIs involves numerous decisions, such as selecting names, data types, and pagination parameters, many of which recur consistently. Although we may not have all the answers, some choices can affect subsequent designs, and mistakes can happen. For instance, if we choose `fromDate` over `from_date`, we must remain consistent within and across APIs, but typos like `fromDate` may still occur. There are various pagination methods and ways to design them; which one should we adopt? Establishing a clear decision-making process is vital for guiding our research and ensuring confident, consistent choices (section 16.1). We can create and continuously enrich an API design toolbox with API design guidelines (section 16.3), ready-to-use design components (section 17.6), and automated guidelines (chapter 18). Such a toolbox can help us seamlessly and consistently apply the correct naming conventions and pagination without the need to remember everything. The API design toolbox elements can enrich the API design reference kit.



NOTE AI will likely become a vital asset in our API design toolbox. You'll find very few tips in this book, because it's not *The Design of Web APIs with AI*. The book focuses on core principles essential for humans and AI. Understanding the API design process and principles before depending on AI is crucial, as AI can produce wrong, inaccurate, or incomplete responses. By mastering API design from this book, you'll better see when and how to integrate AI in your design process, guide AI with essential information to enhance its response quality, and finally ensure that the resulting API does the job, is user-friendly and interoperable, and integrates all security-, efficiency-, context-, and modification-related constraints.

Summary

- Web application programming interfaces (web APIs) are software interfaces that allow communication between applications over a network using the HTTP protocol.
- Web APIs enable communication between mobile or web applications and their backends and between server applications, can expose multiple operations, and can be consumed by multiple applications.
- Different types of web APIs exist; they use the HTTP protocol differently. This book focuses on REST APIs.
- A web API is an interface to an implementation and ideally conceals implementation details.
- A web API can be used by internal applications and developers (private API), selected partners (partner API), or anyone (public API). A private API can be exposed on the internet.

- Consider API design because it can positively or negatively affect developer productivity, security, infrastructure efficiency, costs, end-user experience, and an organization's revenue.
- Design APIs that meet user needs, are usable in different contexts, hide inner workings, and are user-friendly, interoperable, and efficient.
- Design any new private, partner, or public API or modification of an API.
- Design an API after deciding why to create it and before its implementation.
- Design private APIs to build skills via practice and design partner or public APIs more easily.
- Carefully consider API modifications to prevent unexpected crashes or silent behavior modifications.
- Design APIs collaboratively and iteratively.
- Use your or your colleague's SME and IT expertise when designing an API.
- Discuss the design with the people who define the user needs and with consumers to ensure that the API design matches expectations.
- Integrate, adapt, or refuse stakeholders' requests and feedback to design an API that satisfies all parties involved.
- Analyze user needs, and identify the required API capabilities to address them.
- Design the programming interface that represents the identified API capabilities.
- Align the choice of API type with capabilities and context.
- Build an API design reference kit starting with an API capabilities list and standard API description to support the design process and the following stages of the API lifecycle; enrich it according to your needs.
- Consider security, efficiency, context (subject matter, provider, or consumer), backward compatibility, and extensibility when designing an API.
- Establish a clear decision-making process for guiding research and ensuring confident and consistent API design choices.
- Create and continuously enrich an API design toolbox with API design guidelines, automated guidelines, and ready-to-use design components.

Part 1

Fundamentals of API design

Fundamental web API design skills include identifying capabilities that meet user needs, representing capabilities in an HTTP interface, and describing the interface in a blueprint. For example, suppose the “Define stage” output is “SOCNET API”: we must clarify the expected capabilities to avoid creating a design that doesn’t meet user needs. Once we agree with stakeholders on capabilities like “Sharing a status” and “List friends,” we can design a REST API interface with operations such as `POST /statuses` and `GET /friends`, which requires understanding REST API principles and HTTP. Discussing the design with stakeholders requires a formal description of the API. Using a standard API description format instead of a wiki page is recommended for streamlining discussions. It will also facilitate accurate implementation.

To teach these fundamentals of web API design, the first part of this book focuses on the first layer of API design: designing a versatile API that does the right job (section 1.7.1). We go through the essential steps of the API design process (section 1.6): identifying API capabilities, and designing and describing the programming interface. Chapter 2 discusses identifying API capabilities that meet user needs from the Define stage of the API lifecycle. Chapter 3 introduces REST APIs and HTTP and discusses how to examine capabilities to spot elements required to design a REST API. Chapter 4 starts the programming interface design, showing how to represent capabilities with HTTP operations, and

chapter 5 ends it by looking at data modeling. Chapter 6 explains how to describe HTTP operations using the OpenAPI format, and chapter 7 discusses data description via JSON Schema in OpenAPI.

Identifying API capabilities

This chapter covers

- Analyzing use cases
- Identifying unique and versatile operations
- Ensuring alignment with user needs
- Avoiding integrating overly specific consumer requirements
- Avoiding exposing the provider's inner workings

API design begins by analyzing users' needs and identifying the API capabilities required to fulfill them. Identifying the appropriate API capabilities is crucial. As seen in section 1.2, an API with incorrect capabilities can make developing applications that consume it complex or impossible, wasting time and resources and potentially jeopardizing the API provider, even when the faulty API is private.

We'll describe capabilities using plain English or any other natural language rather than programming interface language. This is because form follows function, a design principle that applies to buildings, kitchen appliances, applications, and APIs. An effective API design requires analyzing users' use cases and identifying the generic operations to fulfill them and yet-to-be-discovered use cases before choosing the appropriate programming representations and data modeling. This

simplifies discussions, streamlines the design process, and avoids creating complex or incomplete APIs that don't meet user needs.

This chapter starts with an overview of the identification of API capabilities and when it happens in the API lifecycle and design process. Then we present the API Capabilities Canvas, the methodology this chapter uses to subsequently illustrate the necessary steps to effectively and accurately identify an API's capabilities while avoiding common pitfalls.

2.1 *An overview of identifying API capabilities*

As shown in the API lifecycle (figure 2.1) and in the microwave oven and SOcNET backend API examples (figure 2.2), our first task when designing an API is to analyze user needs from the Define stage to identify the necessary capabilities to address them (section 1.6.1). To simplify our work and learning, we focus on the “versatile API that does the right job” layer, addressing consumer needs, concealing inner workings, and ensuring usability in various contexts (section 1.7.1).

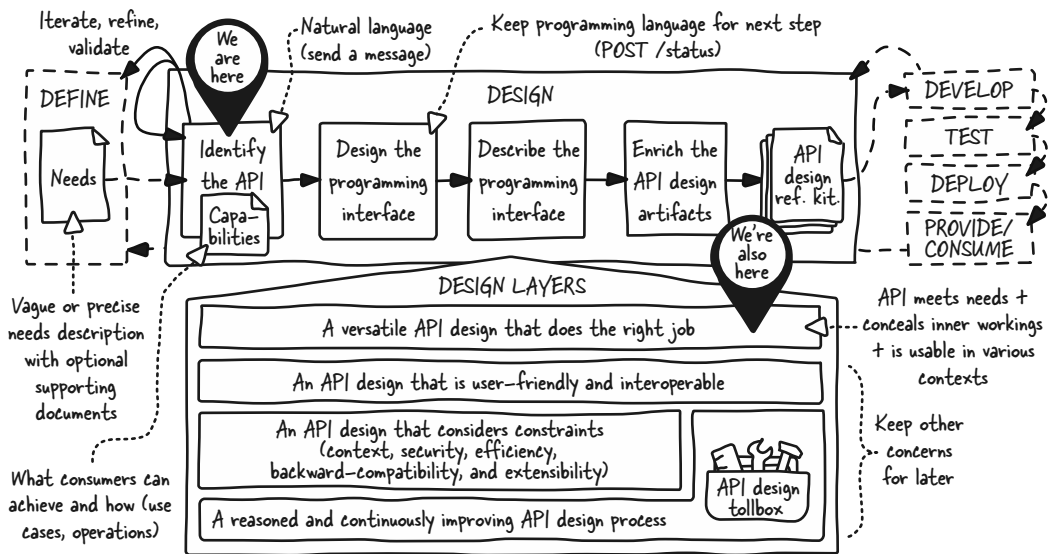


Figure 2.1 We analyze the needs from the Define stage of the API lifecycle to identify the versatile API capabilities that meet them and other unknown needs. We'll iterate with the Define and Design stages stakeholders.

The API capabilities we seek are use cases and operations. Use cases are like recipes, and operations are their generic ingredients. The API will expose the operations to consumers, enabling them to achieve existing and new use cases. We describe these capabilities in plain English or other natural languages (“Send a message”), leaving the programming language (POST /status) for the “Design the programming interface”

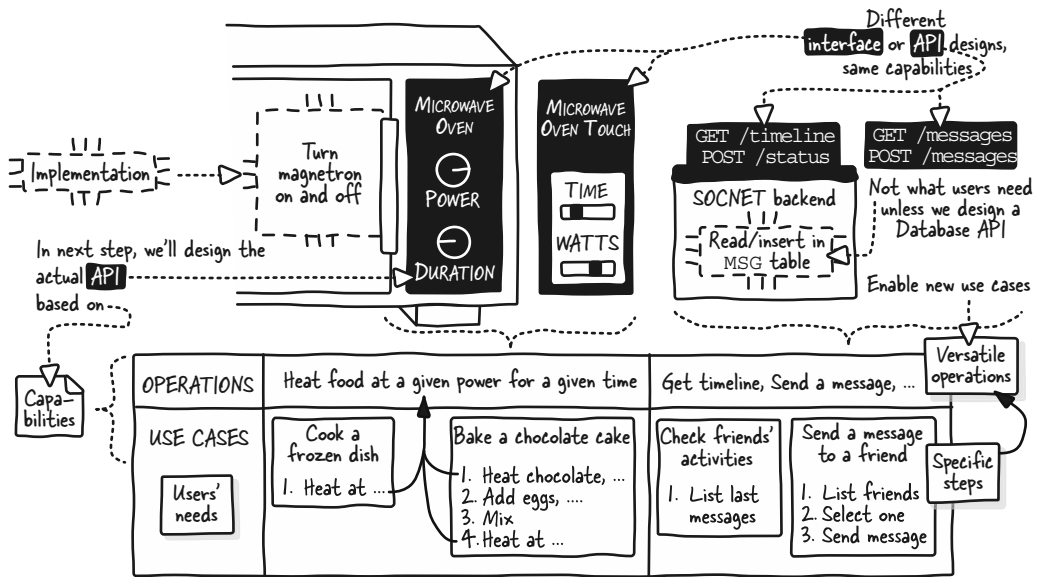


Figure 2.2 API capabilities are use cases and the operations needed to achieve them. We describe them in plain English and keep programming interface concerns for the next stage because form follows function.

step (see section 3.1). We'll iteratively discuss and clarify the subject matter and input and validate our findings with the stakeholders of the Define and Design stages to ensure alignment.

Contrasting a microwave oven and API, this section provides an overview of API capabilities identification, during which we'll

- Use the output of the Define stage
- Analyze what users need to achieve
- Identify versatile operations to achieve use cases
- Keep programming interface design concerns for later
- Clarify the subject matter and input

2.1.1 Starting with the output of the Define stage

We will analyze user needs to identify API capabilities but not determine the API's global purpose. This occurs during the Define stage of the API lifecycle and addresses strategic, product, and IT systems architecture concerns (section 1.3.3). Our input is the output from the Define stage, which describes user needs or problems to solve with more or fewer details.

API design input often fits on a sticky note, especially for a private API. It may range from vague descriptions ("Social network," "Online shopping," or "Database as a service") to precise ones ("Tracking order status on customer mobile and customer care applications"). Such brief input requires us to investigate users and their use

cases from scratch. The input may be more complete, typically for a partner or public API. It may include user experience research, personas, or use cases. We may need to refine the provided use case, as the descriptions may be high-level and not give all the details we need to spot all capabilities.



NOTE APIs may also be created based on nonfunctional requirements. For instance, a server application typically offers a health check API. Such a “technical” API is no different from a “business” API (like “Social Network” or “Database as a service”) and must also be designed using the principles of this book.

2.1.2 *Analyzing what users need to achieve*

When identifying API capabilities, we focus on the use cases users need to achieve, not the inner workings. We’ll look at them in section 14.1.

What do microwave oven users want? They don’t want to turn the magnetron on and off, which is the implementation’s job. Although it’s possible to cook frozen meals or bake cakes by turning the magnetron on and off, doing so requires considerable effort and can lead to half-frozen dishes, burned cakes, and damaged appliances (section 1.2). Users want to “Cook a frozen dish” or “Bake a chocolate cake”; these use cases can involve one or multiple steps. Cooking a frozen dish requires heating at a specified power for a set time. Baking a chocolate cake includes several steps: heating a mixture of chocolate, butter, and milk; incorporating eggs, sugar, and flour; mixing; and then reheating.

Users in the API context may refer to the consumers or their end users (if any). For the SOcNET backend API, users don’t want to read from the `MSG` database table. They want to “Check friends’ activities” or “Send a message to a friend.” Checking friends’ activities requires listing their last messages. Sending a message involves listing friends, selecting one, and sending the message.

Knowing users is vital for accurately identifying API capabilities. For instance, unlike people using a microwave oven, people building microwave ovens or radars want to turn magnetrons on and off, not bake cakes. Database API users need to access specific tables.



NOTE Identifying users and analyzing their use cases can be tricky. To avoid missing anything and to streamline the analysis, see section 2.2. Also, see section 2.8 to discover how inner workings can surface in API capabilities and jeopardize an API.

2.1.3 *Identifying versatile operations to achieve use cases*

We use case steps to identify API operations that meet user needs. Although steps may be specific, operations are versatile (usable in various contexts). For example, the microwave can handle heating steps for “Cook a frozen dish” or “Bake a chocolate cake” use cases with the operation “Heat food at a given power and time.” This operation also supports new cases like “Prepare hot chocolate.” Strictly mapping operations

to specific steps, like “Heat chocolate, butter, and milk,” complicates the interface and hinders new cases. Achieving new use cases would require many specific operations, such as “Heat milk and chocolate.” The interface doesn’t intend to cover all use-case steps; for instance, we let users mix ingredients when baking a chocolate cake because it isn’t the microwave’s role.

The SOCNET backend API’s “Get timeline” operation allows users to “Check friends’ activities.” Operations like “List friends,” “Upload media,” and “Send a message” enable the “Send a message to a friend” use case, although it’s up to users to choose whom to message. A specific operation like “Send a message to friends at Paris Olympics Games” fulfilling a “Contact friends at Paris Olympics Games” use case is unnecessary; generic operations for listing friends (with an “at event” filter) and messaging will suffice for contacting friends at any event.



NOTE Identifying unique and versatile operations is essential to make an API usable in various contexts; check section 2.5. The same goes for fulfilling users’ needs without being too specific; see section 2.7.

2.1.4 Keeping programming interface design concerns for later

When identifying capabilities, we focus on what consumers can do with the API, not its appearance, because form follows function. We describe capabilities in a natural language, such as English or French, rather than in an API programming language, to avoid complicating discussions and creating an API with incorrect capabilities.

Different designs may offer the same capabilities; a microwave may use power and duration knobs or a touchscreen with time and watts sliders. What truly matters now is that users want to “heat food at a given power and time,” not whether they will use “power” or “watts” and “time” or “duration” knobs or sliders to achieve it. The SOCNET backend REST API’s “Send a message” operation could be `POST /status`, `POST /messages`, or `POST /message`, among other options (section 3.1 explains this). Debating these options distracts from user needs and wastes subject matter experts’ time. Also, not all stakeholders understand API language, which complicates discussions and leaves doubts about whether the capabilities meet user needs.



NOTE Another reason we avoid the API programming language is the “Consider constraints” API design layer (section 1.7.3). To-be-discovered constraints could require us to adapt the API design (section 14.1) or even lead us to use a type of API other than REST (section 14.8).

2.1.5 Clarifying the subject matter and input

Identifying API capabilities requires questioning, rephrasing, and resolving contradictions in subject matter- and input-related discussions and documents.

It’s premature to argue over `POST /messages` versus `POST /message` (we discuss this in section 9.3.2), but clarifying the subject matter is crucial. If stakeholders, including us, interchangeably say “Sharing a status” and “Sending a message,” we must clarify: Is

status a message? Is it a private message? This is where our or our colleagues' expertise comes in (section 1.4.1).

We must collaborate with the Define stage stakeholders to align capabilities with their expectations. However, although we focus on Design rather than Define, our role isn't simply to accept every input uncritically. We must discuss unclear or impractical requirements. For example, an input such as "Create a Social Network API" is vague; does it involve content moderation? Similarly, "Create an operation that returns all user data, posts, and private messages" combines unrelated elements, risking performance problems (see section 13.1). Requesting more details uncovers the actual expectations and ensures that we design the appropriate API (see section 2.6.3).

This need for clarification emphasizes the importance of using a natural language that all stakeholders understand, like English, instead of an API language (section 2.1.4). In addition to streamlining discussions, natural language helps refine terms and build a common vocabulary, contributing to capability identification (if you're familiar with the domain-driven design methodology, that's the "ubiquitous language").



NOTE API design, particularly identifying capabilities, is often collaborative and iterative, even for experienced designers. The iterative process described in this chapter (see section 2.2) helps refine information into a comprehensive and accurate list of capabilities that all parties can agree on. Additionally, refer to section 2.6 to stay aligned with inputs and clarify obscure topics.

2.2 *Introducing the API Capabilities Canvas*

Now that we have a general idea, we can explore the steps to take and pitfalls to avoid to achieve the API capabilities identification concretely. The rest of this chapter illustrates this with the API Capabilities Canvas, a methodology and document I developed from years of software and API design experience. I didn't create anything entirely new with this canvas; many API designers, analysts, developers, and tech leads use similar processes to analyze user needs. However, the API Capabilities Canvas consolidates all necessary knowledge to identify API capabilities thoroughly. Experienced designers will find fresh insights and connections with their practice, and newcomers will learn to identify capabilities effectively. Consider the canvas a toolbox adaptable to your needs. With experience, you may not even need it for simple cases, like a small private API used only by yourself; you can process everything in your head. This section looks at how the API Capabilities Canvas works and discusses related tools.



NOTE Identifying capabilities is essential when creating or modifying an API; the API Capabilities Canvas can be used in both situations. However, modifications require more caution as we can break consumers; see section 15.1.

2.2.1 How does the API Capabilities Canvas work?

The API Capabilities Canvas template (figure 2.3) and sample (figure 2.4) show that we decompose the user needs determined in the Define stage of the API lifecycle into small steps in two passes (nominal paths first, then alternative and failure paths), identify unique operations for all steps, and ensure focus on the proper needs.

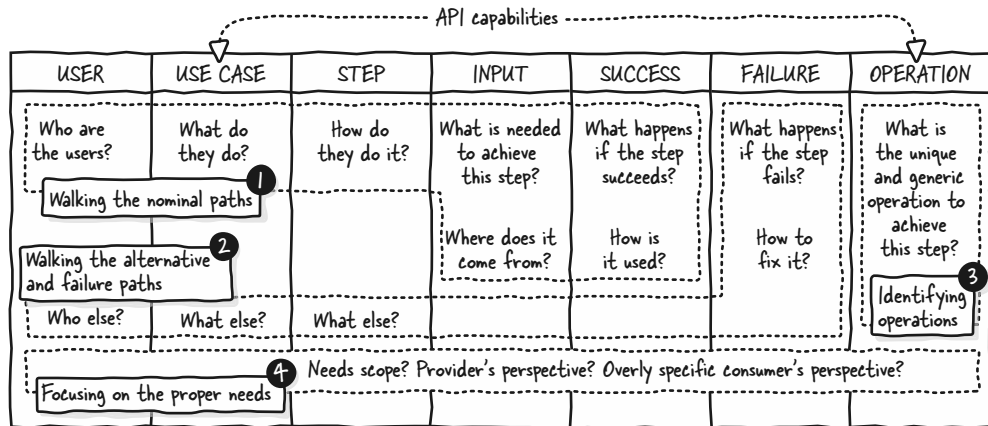


Figure 2.3 We walk through all use cases' nominal, alternative, and failure paths. Then we refine steps to identify unique operations. Finally, we ensure that capabilities fulfill expected users' needs without being too consumer-specific or exposing inner workings.

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE	OPERATION
End users	Buy products	Search for products to buy	Catalog, filters	Products matching filters	No product found	Search for products
		Add a product to the cart	Selected product	Product info.	Product doesn't exist	Add a product to the cart
		Check out	Cart	User gets an order	Empty cart	Create an order
Catalog admins	Fill catalog	Look for similar products	Catalog, characteristics	No product found	Products matching characteristics	Search for products
	

Figure 2.4 Different steps may share the same unique operation.

To exhaustively analyze user needs, we identify users (USER column); describe their use cases (USE CASE column); and list steps to achieve the use cases (STEP column), inputs to execute each step (INPUT column), outcomes and outputs when the step is successful (SUCCESS column), and context, outcomes, and outputs or errors when it fails (FAILURE column).



NOTE The input from the Define stage may include user and user case definitions (section 2.1.1). However, we'll likely need to investigate this input further to fill gaps, add additional information, and identify the needed API capabilities.

We proceed in two passes to separate concerns and simplify the decomposition process. We analyze the nominal, ideal, or happy paths in the first pass. They are the most common and straightforward use cases and sub-paths of steps that lead to successful completion (section 2.3). In the second pass, we investigate alternative sub-paths and use cases and the failures set aside during the first pass (section 2.4). We check each input source, outcome usage, and how to fix failures (second pass only) to ensure that nothing is missed.

To ensure that our API is versatile, we map each step to a unique, context-agnostic operation (OPERATION column). Different steps may share the same operation (see section 2.5).

Ultimately, we ensure that we stay within the Define stage's needs scope and check all elements contribute to fulfilling the proper needs (see section 2.6). We especially don't want to integrate overly specific consumer needs (see section 2.7) or expose inner workings (see section 2.8).



NOTE Identifying API capabilities correctly requires trial and error. Even experienced designers rely on an iterative process and need feedback. Finding the right level of detail can be tricky in the beginning, but keep going; it takes practice.

2.2.2 *Tools to use along with the API Capabilities Canvas*

You can draw and fill the API Capabilities Canvas on a virtual or physical whiteboard. However, a physical whiteboard may fall short for bigger APIs or when modifying multiple elements, and you'll likely need to rewrite everything in a digital document at some point.

A good-old spreadsheet is my go-to for a digital API Capabilities Canvas; examples can be found on my website (<https://apihandyman.io/the-design-of-web-apis>). You can screen-share during API design workshops. Adding or moving elements is easy, and the searching and filtering features are helpful. Pivot tables provide an overview of unique operations and their use, as illustrated in figure 2.5.

It can be helpful to model step flows with diagrams for complex use cases with optional subbranches and loops. Diagram-as-code tools like PlantUML and MermaidJS allow for easy creation and modification of diagrams.

Data is reorganized around operations.

Better view on operation info

OPERATION	INPUT	SUCCESS	FAILURE	STEP	USE CASE	USER
Add a product to the cart	Selected product	Product info.	Product doesn't exist	Add a product to the cart	Buy products	End users
Create an order	Cart	User gets an order	Empty cart	Check out	Buy products	End users
Search for products	Catalog, filters	Products matching filters	No product found	Search for products to buy	Buy products	End users
Search for products	Catalog, characteristics	No product found	Products matching characteristics	Look for similar products	Fill catalog	Catalog admins
...

Figure 2.5 A pivoted API Capabilities Canvas where data is reorganized around operations

2.3 Walking the nominal paths

We start identifying API capabilities by analyzing the nominal paths, which are the ideal and successful paths of the most common use cases. We keep alternative use cases, sub-paths, and failures for section 2.4. Treating nominal and other paths separately simplifies analysis and provides a quick overview of API capabilities for stakeholder discussions. As illustrated in figure 2.6, we use a subset of the API Capabilities Canvas from section 2.2. It focuses on

- Identifying who does what and how (users, use cases, steps)
- Identifying steps' inputs and successful outcomes
- Spotting missing elements

USER	USE CASE	STEP	INPUT	SUCCESS
Who are the users?	What do they do?	How do they do it?	What is needed to achieve this step?	What happens if the step succeeds?
Who does what, and how? ①		+	Inputs and success outcomes	②
			Where does it come from?	How is it used? ③
			Missing elements	

Figure 2.6 When walking the nominal path, we focus on the ideal and successful paths of the most common use cases. We wonder who (users) does what (use cases) and how (steps). We also identify steps' inputs and successful outcomes to spot missing elements.

In this section, we'll learn how to walk the nominal paths using a sticky note indicating "Online Shopping" as input.

2.3.1 Identifying users

Walking the nominal paths starts by investigating who the users are, what they do, and how they do it, as shown in figure 2.7. This section discusses users, section 2.3.2 discusses use cases, and section 2.3.3 discusses steps.

USER	USE CASE	STEP	INPUT	SUCCESS
End users	Buy products	Add a product to the cart		
Who are the users? ¹	What do they do? ²	Check out	How do they do it? ³	

Figure 2.7 Identifying users ensures exhaustively listing use cases. Breaking down use cases into steps will help us identify all operations.

Identifying users is essential for framing areas to cover and identify all use cases. Users, also called *profiles*, include consumers (applications, developers, organizations) and application end users. Although end users don't directly use the API, their needs shape its capabilities. We can either list all users and focus on significant ones or start with a shortlist of key users and revisit identification in the second pass (section 2.4.3). We can use the 80/20 Pareto principle to identify significant users, targeting the 20% of users who provide 80% of use cases or business value. Exact numbers don't matter; we seek key elements that cover the most ground. In our "Online Shopping" example, the key users are those shopping online through the mobile application or website (that consumes the API); these applications and their developers can also be seen as users. We may keep administrators and corporate end users for the second pass.



NOTE The API Capabilities Canvas also helps spot missing users; see section 2.3.6. Identifying users, profiles, or roles also has important implications regarding security; see section 12.1.

2.3.2 Listing use cases

Once we know the users and select the key ones, we can list what they do and hence their use case or the high-level actions, processes, and flows they perform. Using the 80/20 rule and subject matter knowledge, we can prioritize the 20% of use cases that address 80% of a user's needs. Again, these are not actual numbers; a single use case may give a good idea of what the APIs need to provide. We reserve the other users and use cases for the second pass (section 2.4.3). During workshops with subject matter experts (SMEs) unfamiliar with the methodology, you can encourage them to select a familiar use case for practice. In our case, what do most end users of an "Online Shopping" application do? As seen in section 2.1.2, these users likely don't want to select * from product—that is, read the product table in the database. They "Buy products." They can also "Manage their delivery preferences." Use cases can be more specific, like "Buy birthday presents."



NOTE Use prioritization methods that work for you and the context. We can pick the top five use cases, regardless of users. The idea is to cover the most ground with minimal effort and quickly get an overview of the API's main capabilities.

2.3.3 Decomposing use cases in steps

Stopping at the use-case level risks missing capabilities; it is crucial to decompose the use case into steps. How do the end users buy products? They repeatedly “Add a product to the cart” and then “Check out.” These are the two steps of the use case. Managing delivery preferences can be done with “View delivery preference” and “Update delivery preferences.”

Don't rush the process and think about generic operations; we'll work on this in section 2.5. Keep step descriptions as said by the people participating in this or as they come to your mind. For example, to “Buy Christmas presents,” users may “Add a gift to Santa's basket” and then “Check out.” Although we likely already see that the first step is similar to “Add a product to the cart,” we keep its original description.



TIP We can work iteratively to avoid analyzing unneeded areas. The user and use-case levels (without steps) can provide an initial overview of API capabilities, which can help get stakeholders' first confirmation that we're aligned. Then we can break down use cases into steps to identify accurate capabilities. We can proceed similarly with the second pass (section 2.4).

2.3.4 Determining inputs and success outcomes

We determine inputs and success outcomes for each step, keeping failures for later (section 2.4). Inputs are the information or business concepts necessary to achieve the step. Success outcomes indicate what happens when the step executes smoothly. They describe inputs' states after the step, what has been created or done, or an event. These elements are viewed from the users' perspective and remain coarse-grained; for instance, we don't need to detail a product's properties (discussed in section 5.1).

What do end users need to add a product to the cart? As shown in figure 2.8, they need a product and a cart. And what happens when a product is added to the cart? The success outcome description states, “Product added to cart.” Similarly, for “Check out,” the input is a cart, and the successful outcome is “User gets an order.” When viewing delivery preferences, we need a “User” and get “Delivery preferences” in return.

USER	USE CASE	STEP	What is needed? ¹	What happens? ²
			INPUT	SUCCESS
End users	Buy products	Add a product to the cart	Product, cart	Product added to the cart
		Check out	Cart	User gets an order

Figure 2.8 Inputs and success outcomes are essential for understanding user needs, identifying and designing operations, and spotting missing elements.

Updating these preferences requires “Modified delivery preferences” as input; the success outcome is “Delivery preferences are updated.”

2.3.5 Spotting missing elements with sources and usages

Analyzing input sources and success outcome usages reveals missing steps, use cases, or users. Inputs can be user-known, API-managed, or from prior steps. Success outcomes may also serve as inputs for other steps. Figure 2.9 illustrates this for the “Buy products” steps.

USER	USE CASE	STEP	INPUT	SUCCESS
End users	Buy products	Search products to buy	Where does it come from? ¹	How it it used? ²
		Add a product to the cart	Product (Search products to buy) cart (API)	Product added to the cart (Check out)
		Check out	Cart (API)	User gets an order (Manage orders)
	Manage orders			

Figure 2.9 Investigating the “Buy products” step’s input sources and success outcome usages allows us to spot a missed step and use case.

For the “Add a product to the cart” step, we check where the cart and the product come from. The API manages the cart (users don’t provide it or get it from another step). Users search for products before adding them to the cart; we missed an essential step. But it’s fixed by adding the “Search for products to buy” step at the beginning of the “Buy product” use case. The successful outcome of “Add a product to the cart” is “Product added to the cart.” It’s useful for the “Check out” step, but there’s nothing new here.

We proceed similarly with the “Check out” step. It needs a cart that is managed by the API. When a “User gets an order,” they may want to check its status, modify or cancel it, and even do that with all their orders. We have uncovered a new area to investigate: “Manage orders,” which we added to the use case list for the end users.



NOTE Some elements we spot may not be in the initial scope defined in the input; remember to validate with stakeholders whether the new aspects should be included in the API.

2.3.6 Analyzing the spotted elements

To investigate the newly identified elements, we proceed as before (identifying use cases, decomposing them in steps, and spotting missing elements). Figure 2.10 illustrates this analysis for the “Search products to buy” step. We identified a new step, use case, and user.

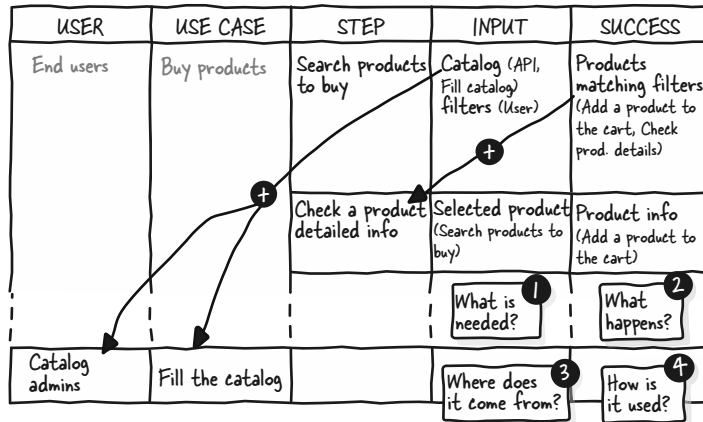


Figure 2.10 Investigating the “Search for products to buy” step’s input source and success outcome helps us spot new elements.

We identified the step’s inputs and success outcomes. Users need a catalog of products to search for products and would benefit from search filters (details aren’t needed at this stage). In return, they get the “Products matching filters.”

We analyzed input sources and the use of success outcomes to spot missing elements. End users provide filter values, and the API manages the catalog. However, catalog administrators “fill the catalog” with products. We identified a new type of user and one of their use cases. When end users get the “Products matching filters,” they may add products to their cart or check the product’s detailed information beforehand. We added this step to the “Buy products” use case after the search step.

2.4 Walking the alternative and failure paths

Focusing only on the nominal paths would result in an incomplete API design. Once we have walked them, we must explore the alternative (less common) and failure paths. As shown in figure 2.11, we continue using the API Capabilities Canvas introduced in section 2.2 and will

- Describe failures for each step
- Add alternative branches on use cases
- Add alternative users and use cases

We continue working with the “Online Shopping” example from section 2.3. We investigate failures for each step of the “Buy products” use case, add its alternative and failure branches, and then step back to see how to identify and analyze other alternative use cases.

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE
		What do they do if ...?	What happens in case of failure? How is it used?
			Alternative branches		
					Steps failures
Who are the users?	What do they do?

Figure 2.11 To complete the analysis, we investigate the alternative users, use cases, and branches set aside during the first pass and analyze what happens when the steps fail.

2.4.1 Analyzing failures for each step

Analyzing failures helps spot missing steps, use cases, or users and is essential for creating a user-friendly API (section 9.8) and developing the implementation (section 19.1.3). For each step, we list potential failures, errors, or problems from the user's perspective and explain why they occur and how to fix them. Missing or invalid inputs, data states, or business controls can cause failures. Figure 2.12 illustrates this analysis for the steps of the “Buy products” use case; the FAILURE column is filled, and a new step is identified.

USER	USE CASE	STEP	INPUT	Problems? ¹	Why? ²	FAILURE	How to fix? ³
End users	Buy products	Search products to buy	Catalog			No product matching filters (Retry with diff. filters) No product found when no filters (Fill catalog)	
		Check a product detailed info.	Product			Product doesn't exist (Search products to buy)	
		Add a product to the cart	Product, cart			Product doesn't exist (Search products to buy)	
		Check out	Cart			Cart is empty (Add a product to the cart) A product is unavailable (Remove unavailable product from cart)	
		Remove unavailable product from cart	Product, cart			Product not in cart (No fix)	

Figure 2.12 Investigating the failures of the “Buy products” steps, their causes, and how to fix them helps identify a new step.

What problems can occur when searching for products? No product can be found due to users providing filters with no corresponding product in the catalog or the catalog being empty. As a fix, end users may search again with different filters, or administrators may “Fill the catalog” (as identified in section 2.3.5).

What can go wrong if “Check a product details info” is executed without a prior search? No product details may be found because the requested product may not exist in the catalog. To fix that, users can “Search products to buy” to find products existing in the catalog and try again.

“Check out” can fail if the cart is empty, which can be fixed with “Add a product to the cart.” The cart may also contain an unavailable product; the user can “Remove unavailable product from cart” to fix this. It’s a new step we add after “Check out.”

As we did for other steps, we investigate what is needed to achieve it and what happens in case of success and failure. Users need the cart managed by the API and the product indicated in the failure of “Check out.” On success, the product is removed from the cart. It fails if a user tries to remove a product not in the cart; there’s no fix.

2.4.2 Adding alternative branches on each use case

We must explore other user actions within use cases to ensure comprehensive API capabilities. We can identify alternative paths by examining potential events before or after the identified steps. We can also add elements previously set aside to streamline the nominal pass. We analyze new steps as usual, similar to how we have decomposed and analyzed use cases. This section briefly explores alternative paths. Use your knowledge to fill in gaps by listing steps, identifying inputs and their sources, and noting success and failure outcomes and fixes. Figure 2.13 shows the completed “Buy products” use case with steps of an alternative branch.

USER	USE CASE	STEP	What if? 1	INPUT	SUCCESS	FAILURE
End users	Buy products	Search products to buy	
		Check a product detailed info	
		Add a product to the cart	
	Business as usual 2	Verify cart content	
		Remove unwanted product from cart	
		Check out	
		Remove unavailable product from cart	

Figure 2.13 Asking “What if ...” helps identify a non-nominal branch in the “Buy products” use case.

What if a user changes their mind about a product on “Check out?” They need to “Remove unwanted product from cart” (step of an alternative branch). To do so, they need the cart and the product (its input), which can be obtained with “Verify cart content” (new step spotted with input source). Other examples of alternative branches could be “What if a user doesn’t have an address defined for delivery?” and “What if the price of a product in the cart has changed?”

“What if?” is not always needed. Thanks to our subject matter expertise, we may know that users usually “Verify cart content” before “Check out” and may “Remove unwanted product from cart” afterward. We may have omitted this alternative branch to streamline the nominal pass on the “Buy products” use case.

2.4.3 *Analyzing the alternative users and use cases*

To ensure API capabilities’ exhaustivity, we must list (if we haven’t already) and analyze the alternative or secondary use cases and users we set aside for the first pass (section 2.3). During our first pass, we may not have covered the “Manage delivery preferences” use case. Alternative use cases can be edge cases that rarely happen or specific use cases dealing with problems, such as “Notify a problem with an order” or “Be notified when a product is back in stock.” Alternative users, for our example, could be catalog administrators or corporate end users. Whatever their nature, we analyze them like the other users and use cases, including checking with stakeholders if they are to be covered by the API.



CAUTION We prioritized nominal elements to get a quick overview and simplify our work, analyzing one aspect at a time. However, we must not neglect alternatives. For example, administrators may be less prominent than regular users, but the system may not work if their use cases are not fulfilled.

2.5 *Refining steps to identify operations*

After analyzing use cases, we identify versatile operations for all steps; different steps may share the same operations. As seen in section 2.1.3, these context-agnostic operations can be used in various situations, enabling consumers to address new use cases. We aim to grasp the fundamentals of the API’s subject matter, which is essential for creating a reusable, user-friendly API. We will use these operations to design the programming interface (see section 2.1). Using the “Online Shopping” example and API Capabilities Canvas, this section clarifies the distinction between steps and operations, showing how to refine steps to find operations.

2.5.1 *Differentiating steps and operations*

Differentiating steps from functions or operations is essential to creating a reusable and user-friendly API design. An API bloated with duplicates or highly specific operations is difficult to use and reuse.

If we turn each step into an API operation as identified, we’ll end up with many similar ones. For example, in the “Buy products” use case, the “Remove unwanted product from cart” and “Remove unavailable product from cart” steps are very similar. A unique “Remove product from cart” operation can fulfill them. Similarly, the “Add a product to the cart” step of “Buy Product” resembles “Add a birthday present to cart” for the “Buy a birthday present” use case.

Many steps won’t have lookalikes and need a specific operation. However, we still need to differentiate them from the operations that fulfill them to make these

operations usable in other yet-unknown contexts. For example, we can't use the "Search products to buy" step as an operation as it is specific to the "Buy products" use case. A context-agnostic "Search for products" operation can fulfill it and is reusable in other situations we may encounter long after API deployment like creating a birthday wishlist.

2.5.2 Identifying unique and versatile operations

As shown in figure 2.14, for each step, we look for similar steps in the API Capabilities Canvas and determine their true intent by using their description, inputs, and success outcomes to describe a unique, context-agnostic operation fulfilling them. Figure 2.15 shows the operations for two "Online Shopping" use-case steps. Steps marked with the same letter are similar and share the same operations. This example includes the unanalyzed "Fill catalog" use case from section 2.3.6, which involves searching for similar products, verifying their details, and adding them if they are not duplicates.

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE	OPERATION
Repeat for each step without operation *		Similar steps? 1	True intent? 2			Unique and context-agnostic operation 3

Figure 2.14 Checking whether there are similar steps or digging to find each step's true intent allows us to identify unique and context-agnostic operations.

USER	USE CASE	STEP	INPUT	SUCCESS	...	OPERATION
End users	Buy products	A Search products to buy	Catalog, filters	Products matching filters	...	Search for products
		B Check a product detailed info	Selected product	Product info.	...	Get product details
		Add a product to the cart	Selected product, cart	Product is in cart	...	Add a product to the cart
		Verify cart content	Cart	Products in cart	...	List products in cart
		C Remove unwanted product from cart	Unwanted product, cart	Product removed from cart	...	Rem. prod. from cart
		Check out	Cart	Order	...	Check out
		C Remove unavailable product from cart	Unavailable product, cart	Product removed from cart	...	Rem. prod. from cart
Catalog admins	Fill catalog	A Look for similar products	Catalog, filters	Product matching filters	...	Search for products
		B Verify if product is different	Found product	Product info.	...	Get product details
		Add product to catalog	Product, catalog	Product is in catalog	...	Add prod. to catalog

Figure 2.15 We identified unique and versatile operations for each step of the "Buy products" and "Fill catalog" use cases. Three operations appear in more than one step.

The “Search products to buy” step is similar to the “Look for similar products” step of the “Fill catalog” use case (operation A). Their descriptions resemble each other (“Search products ...” and “Look for ... products”), and they share the same inputs (Catalog, filters) and success outcomes (Products matching filters). Their true intent is to “Search for products”; we use it as their unique and context-agnostic operation description.

The same goes for “Check a product detailed info” and “Verify if product is different” (B). They have the same fundamental intent and operation, “Get product details.” The “Verify cart content” step has no similar steps. Its description is specific and doesn’t clearly express the actual intent. We can find it by looking at the success outcome; it returns the list of “Products in cart.” Its operation is “List products in cart.”

As with A and B, “Remove unwanted product from cart” and “Remove unavailable product from cart” (C) are similar. We can remove the context-specific “unwanted” and “unavailable” qualifiers from their descriptions to get their operation: “Remove product from cart.”

The “Check out” and “Add a product to the cart” steps have no similar steps, and their descriptions are context-agnostic; we can keep them for their operation.



NOTE The same operation can be used in various steps of a use case and across different use cases and users.

2.6 *Focusing on the proper needs*

Although we seek API capabilities that meet user needs from the Define stage, we can be off track. To ensure the API’s accuracy and versatility, we must carefully filter, transform, or accept the elements we add to the API Capabilities Canvas (users, use cases, steps, inputs, outcomes, operations). This section discusses staying within the scope identified during the Define stage, focusing on the proper perspectives, and using the “Why?” question.

2.6.1 *Staying within the Define stage’s needs scope*

We must stay within the scope of the requirements clearly or vaguely defined during the Define stage (section 2.1.1). To do so, we can request confirmation from the stakeholders of the Define stage. To streamline discussions, we can also evaluate whether what we find is within the scope of the input subject matter(s) and verify the usage of outcomes.

When user needs are unclear or coarse-grained, we can request confirmation from the stakeholders of the Define stage before investigating new areas related to the initial user needs we uncover or think of. For instance, does “Online Shopping” cover the administration of the product catalog?

Elements unrelated to the user needs’ subject matter(s) are highly questionable. For instance, “Check end user bank account balance” looks distantly related to “Online shopping,” so maybe we shouldn’t include it in the scope of the API. Still, it can be

OK; our analysis may uncover initially unidentified subject matters not explicitly identified in our input. It's up to SMEs and the stakeholders of the Define stage to decide.

Verifying outcomes usage can help streamline decisions; it is a direct follow-up to looking for missing elements (section 2.3.5). We can likely remove any steps whose outcomes are useless to users and are not inputs for other steps. For example, if the "Buy products" use case has a "List product suppliers" step, and users do not use this information, and it is not an input for another step, we should remove it.



TIP Take an iterative approach. Validate the nominal list of users and use cases before further investigation (section 2.3); proceed similarly with the alternatives (section 2.4). Confirm newly identified topics before investigating them (section 2.3.5).

2.6.2 Focusing on the proper perspectives

Although we design an API to fulfill consumer needs, integrating needs specific to a consumer or integrating more generic needs in a way specific to a consumer leads to less versatile and reusable APIs. It's up to us and SMEs to balance all consumers' needs by staying focused on the subject matter(s). Section 2.7 illustrates typical overly specific consumer-needs situations.

Our expertise in the subject matter, software architecture, or existing implementation may lead us to expose inner workings that are not the consumer's business. Helped by SMEs, architects, tech leads, or implementation developers, we must ensure that we do not expose the provider's perspective. Section 2.8 shows typical examples.



NOTE We'll see in section 14.1 that sometimes we must adapt our design to provider and consumer habits and limitations.

2.6.3 Asking why to investigate any problem

"Why?" is a powerful question that helps us better understand the user needs and investigate potential problems. For example, why should users deactivate their addresses when updating them? Because a user can have only one active address in the database, which is not the consumer's business but the provider's (see section 2.8.2). Asking why several times can help us get to the root of any problem and identify unnecessary elements or proper capabilities.

2.7 Avoiding integrating too specific consumers' perspective

Although we design an API from the consumers' perspective, we must be careful not to be too specific, or the resulting API may be usable by only one or a few consumers or may not be reusable in other contexts. This section uses the "Online Shopping" example to illustrate how we can be too consumer-specific by

- Mapping our API design to consumers' user interface (UI)
- Integrating consumers' business logic

2.7.1 *Avoiding mapping consumers' UI*

Designing an API based on an existing or wireframe UI can be helpful. Still, we must be careful not to create use-case step flows representing specific UI flows instead of context-agnostic subject matter flows. UI-specific flows make APIs hard to reuse in other contexts, such as a modified UI or another application. Typically, optimizing a UI flow to gain more customers could be hindered because it requires revising the API, adding development costs and delays. Modifying the API and the UI can be impossible because other applications rely on this specific API call flow.

Like the UI it is based on, the “Create a user account” use case (see figure 2.16) comprises four steps/operations: “Save the user’s email,” “Save the user’s first name and last name,” “Save the user’s address,” and “Validate the new user.” Four UI screens to create a user account may make sense. But for an API, that means four calls, making the use-case flow unnecessarily complex. Also, if the screen order changes, can we change the step order to match it? Additionally, not executing the final step could result in incomplete user accounts.

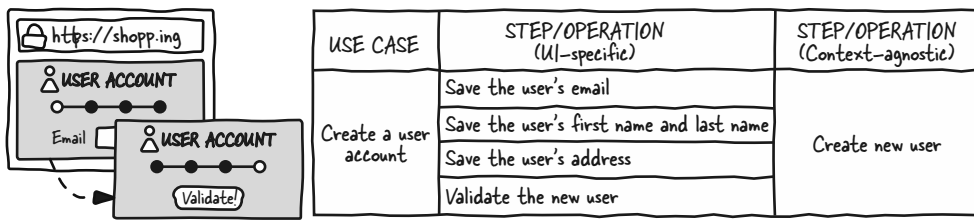


Figure 2.16 UI-specific flows are less reusable and flexible than context-agnostic flows.

Instead, a single “Create new user” subject-matter-focused step/operation is easily usable by any consumer (server application or UI). They are free to divide the information gathering into several steps. But ultimately, they will make a single API call to create a user account.



CAUTION If a use case’s flow mentions fine-grained information or can’t stand a UI flow modification, that’s a sign of a too-specific consumer perspective. We must replace it with a generic, context-agnostic flow focusing on the subject matter. See section 10.3 for designing flexible flows and section 10.4 for flexible flows that save data.

2.7.2 *Avoiding integrating consumers' business logic*

Consumers may try to delegate a specific job to the API, leading to tight coupling and reduced reusability.

For example, an application using our “Online Shopping” example needs to show a weather forecast pictogram based on the user’s address, leading to a “Get user’s

weather forecast” use case. We, and SMEs, can consider this highly specific to this application and unrelated to our primary subject matter, so we won’t include it in our “Online Shopping” capabilities. Still, weather-forecast-related features may make sense. For instance, the “Search for products” operation could have a filter to get products related to a weather condition like “winter,” “summer,” or “rain.” It would be up to the consumers to provide the conditions they think are interesting to their end users.



CAUTION An element (user, use case, step, input, outcomes, operation, or, later, data model) implying integrating concerns, business logic, or processing unrelated to or distantly related to the subject matter may be a sign of a too-specific consumer perspective. In case of doubt, check with an SME. Distantly related subject matters may also indicate that we need different APIs to fulfill all user needs; see section 11.2.

2.8 Avoiding exposing the provider's perspective

APIs reflecting the provider’s perspective expose the inner workings that consumers shouldn’t be bothered with. They are hard to understand and use and can harm the underlying systems. This section uses the “Online Shopping” example to illustrate three common ways of doing so:

- Exposing data organization
- Delegating business logic
- Exposing software architecture

2.8.1 Avoiding exposing the provider's data organization

An API design can mirror the underlying data organization (tables, databases), distancing it from the fundamental subject matter and making it complex. At the API level, consumers should view data (such as a customer or product) as cohesive business concept units; the implementation must manage data complexity. Figure 2.17 illustrates how data organization can affect or not affect a use case for retrieving customer information.

The first example has two steps/operations, “Read CUSD” and “Read CUSA,” mapping the customer data organization in two CUSD (customer data) and CUSA (customer address) tables. We hope they understand that CUSD and CUSA are customer data; it’s up to the consumer to aggregate the data retrieved with the two operations to get all customer data.

The second example is similar. It replaces the cryptic names with more meaningful ones: “Read customer data” and “Read customer address.” However, consumers still have to aggregate data from the two operations.

The third example doesn’t expose the data organization and focuses on the subject matter with a single step/operation: “Read custom information.” The implementation manages data aggregation, and the consumers get the needed data easily.

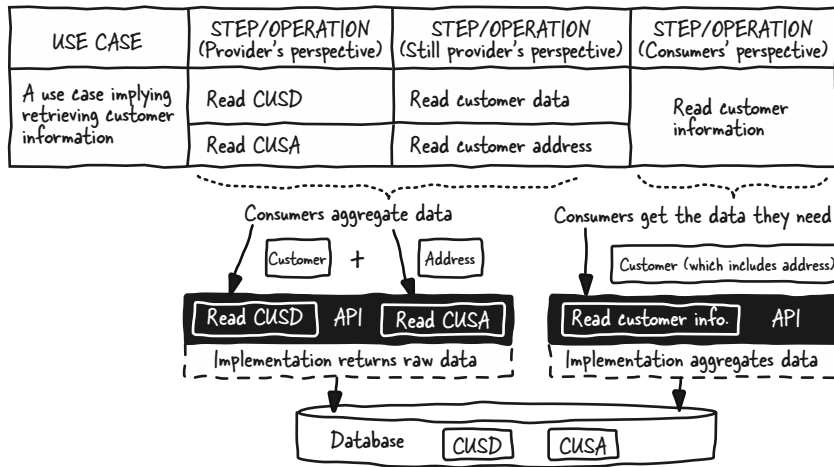


Figure 2.17 Exposing database structure through an API makes it complicated to understand and use.



CAUTION Table names present in step or operation names or steps that indicate how data is structured may be a sign of the provider's perspective.

2.8.2 Avoiding exposing the provider's business logic

An API design can mirror internal business logic, making it hard to use and potentially leading to underlying data and system corruption. Figure 2.18 illustrates this with an API relying on a system where older addresses are kept for security purposes; a customer's address is the one with an “active” status.

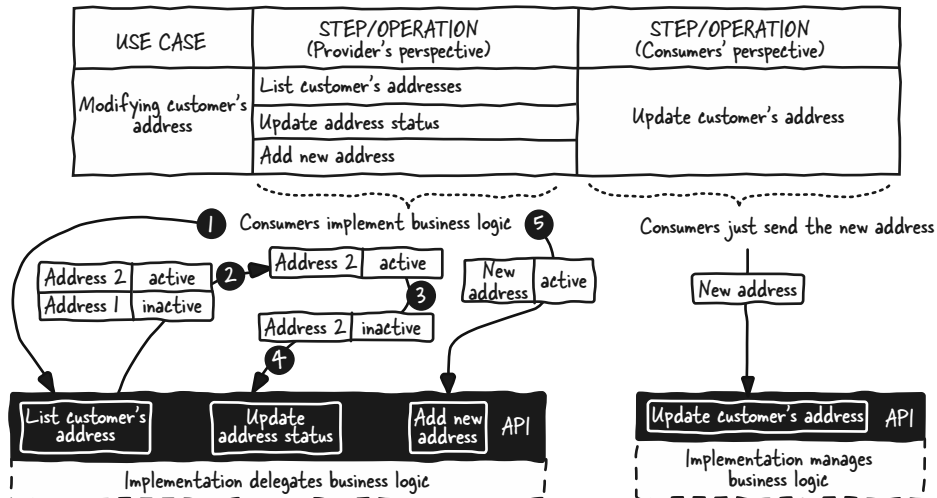


Figure 2.18 Delegating business logic to consumers can lead to system and data corruption.

Modifying a customer's address from the provider's perspective takes three steps. Users "List customer's addresses" to get the active one, "Update address status" to make this address inactive, and finally, "Add new address" with an active status. Going through these steps and data manipulation is complex, but more critical problems exist.

These steps will be executed by an uncontrolled consumer (developed by a third party, for example) or in an unsecured environment (a browser, for instance). Due to unexpected crashes, errors in code, or malicious intent, consumers may stop at the second step or add a new address without deactivating the active one, leading to data integrity problems. When thought of from the consumer's perspective, the use case has a single step, "Update customer's address," which ensures that the implementation we control manages the business logic securely and preserves data integrity.



WARNING If incorrect API steps or operation executions can compromise underlying data and systems integrity, we trust API consumers with business logic. It's solely the implementation's responsibility to handle such logic. For more secure API design considerations, see section 12.1.

2.8.3 Avoiding exposing the provider's software architecture

APIs enable building systems from various software pieces, but exposing the composition of an API's system can lead to complex and less performant APIs. Figure 2.19 illustrates this problem with an API relying on a system composed of two microservices (or small server applications). One handles most of the products' data, and the other manages their prices.

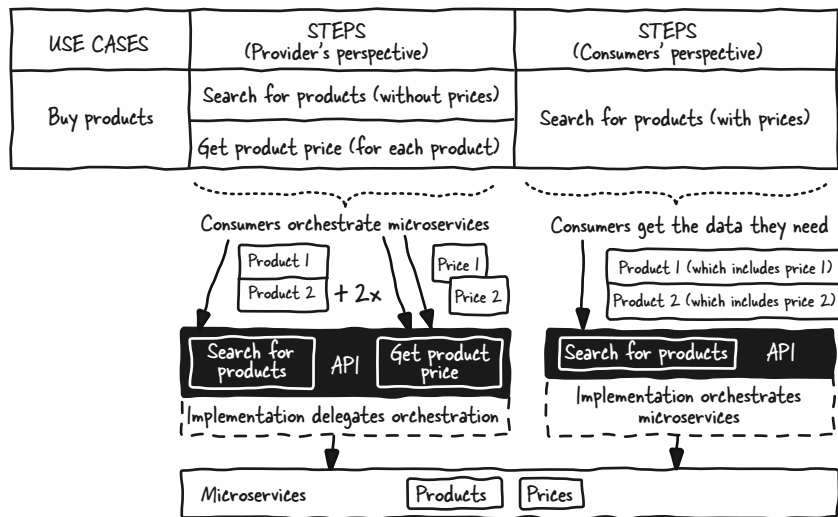


Figure 2.19 Exposing underlying system architecture can make APIs complicated to understand and use.

Conway's law

Conway's law states that any organization's system design will mirror its communication structure. This adage, first published in April 1968 in *Datamation* magazine, applies to APIs. They are influenced by the organization's communication structure and how it exchanges and processes data across its applications.

When buying products from the provider's perspective, users "Search for products" and loop on all products to "Get product price." A product without a price is irrelevant from the subject matter perspective of our API, so users will always do this inconvenient sequence, which also has performance concerns we'll discuss in section 13.1.

There can be excellent reasons for such an architecture, but that's none of the consumer's business, and it splits an API's business concept across operations. A single "Search for products" whose implementation handles getting their data, including their price, is preferable.



CAUTION If application names appear in steps or operations or retrieving data requires complex sequences, it's probably a sign of the provider's perspective. It's up to the API implementation to deal with the complexity of different applications handling a business concept.

Summary

- Analyze the user needs from the Define stage to identify API capabilities, which are the use cases the API must cover and the operations needed to achieve them.
- Describe API capabilities in natural language ("Send a message") instead of API language (`POST /status`) to ensure accurate identification and streamline discussions.
- Clarify subject-matter vocabulary variations or uncertainty to identify the correct API capabilities ("Are A and B the same thing?").
- Confirm with Define stage stakeholders that the found capabilities meet their expectations, and request clarification or challenge their inputs when necessary.
- List all users and their use cases to ensure that all capabilities are identified.
- Streamline use case analysis by focusing on nominal paths: the most common and successful cases. Save alternative use cases and failures for a second pass.
- Decompose use cases in steps to ensure exhaustive API capabilities.
- Identify the steps' source of inputs and success outcomes' usage to uncover missing steps, use cases, or users.
- Analyze alternative use cases, paths, and failures for exhaustive API capabilities.
- List potential failures, errors, or problems from the user's perspective, and explain why they occur and how to fix them to identify missed elements.
- Discover alternative paths by examining potential events before or after the identified steps ("What if?").

- Use steps' description, inputs, and success outcomes to identify similar ones (if any), and describe a unique, context-agnostic operation fulfilling them.
- Challenge elements distantly related to the user needs' subject matter(s) and those whose outcomes aren't used.
- Refrain from mapping use case flows to UI flows or integrating consumer-specific business logic; this makes an API less reusable and flexible.
- Refrain from exposing data organization, trusting consumers with business logic, or exposing software architecture; these make an API complex and can harm the underlying system.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 2.1

You're designing an API for an HR tool that manages time-off requests. List at least three potential users for this API (think about your experience taking time off, and remember that there are different types of users). Then choose which user you would prioritize for analyzing use cases, and imagine the use case you would analyze first. Briefly explain your reasoning.

Exercise 2.2

You're designing an API for a customer relationship management (CRM) tool for sales teams. Can you explain what's wrong and why with the following list of use cases for the sales representative user? Bonus question: what action could you take with what's wrong?

- Prepare a customer meeting
- Follow up on sales lead
- Verify data synchronization processes
- Identity and reach out to customers at risk of churn
- Track ongoing deals

Exercise 2.3

You're designing an API for a food delivery service and listing the steps for the "Place an order" use case for the "Customer" user. Investigate each step's input and successful outcome to find the missing step in the following list:

- Search restaurant
- Add dish to order
- Pay order

Exercise 2.4

You're designing an event management API. Your analysis reveals that event organizers can create events and offer event tickets, and customers can book tickets for an event. Using their description, inputs, and success outcomes, identify unique and versatile operations for the following use-case steps:

- (Create an event) Verify if the event already exists
- (Create an event) Create the event
- (Book a ticket for an event) Search for available events
- (Book a ticket for an event) Add tickets to order
- (Book a ticket for an event) Pay order
- (Offer event tickets) Add tickets to gift
- (Offer event tickets) Validate gift

Exercise 2.5

You're designing an API for a library management system. What's wrong with the following steps describing the "Return a borrowed book" use case? How do you fix the problem?

- Search for the borrowing record based on book ID
- Update the borrowing record to indicate the book has been returned
- Remove the borrowing record reference from the user's account

Observing operations from the REST angle

This chapter covers

- The basics of HTTP and REST APIs
- Identifying resources and their relations
- Identifying resources' actions and their inputs and outputs

Now that we have analyzed users' needs and identified the API capabilities required to fulfill them, especially the operations, we can start turning their plain English (or any other language) descriptions into the programming interface. In the context of this book, this means designing a REST web API. It requires knowing the basics of HTTP and REST APIs, which the sample without explanation in section 1.1.2 didn't teach us. We still have no clue how an operation like "Send a message" can be turned into a `POST /status`, `POST /message`, or `POST /messages` HTTP request. Additionally, designing such a programming interface covers different aspects we must know and separate to simplify our learning and work while ensuring that our API design is accurate and versatile.

This chapter examines the "Design the programming interface" step of the API design process by explaining the basics of HTTP and REST APIs and how to design a REST API. Then we focus on our first task within this step: observing the operations

we identified from the REST perspective. This consists of identifying the business concepts with which the API deals, called *resources* in REST APIs, how they relate, the actions that apply to them, and their inputs and outputs.

3.1 An overview of programming interface design

As shown in figure 3.1, we now enter the second step of the API design process, “Design the programming interface” (section 1.6.2). This step aims to design the REST API that represents the operations we described in plain English or any other natural language during the identification of API capabilities (section 2.1). We continue to focus on the “versatile API that does the right job” layer, addressing consumer needs, concealing inner workings, and ensuring usability in various contexts (section 1.7.1).

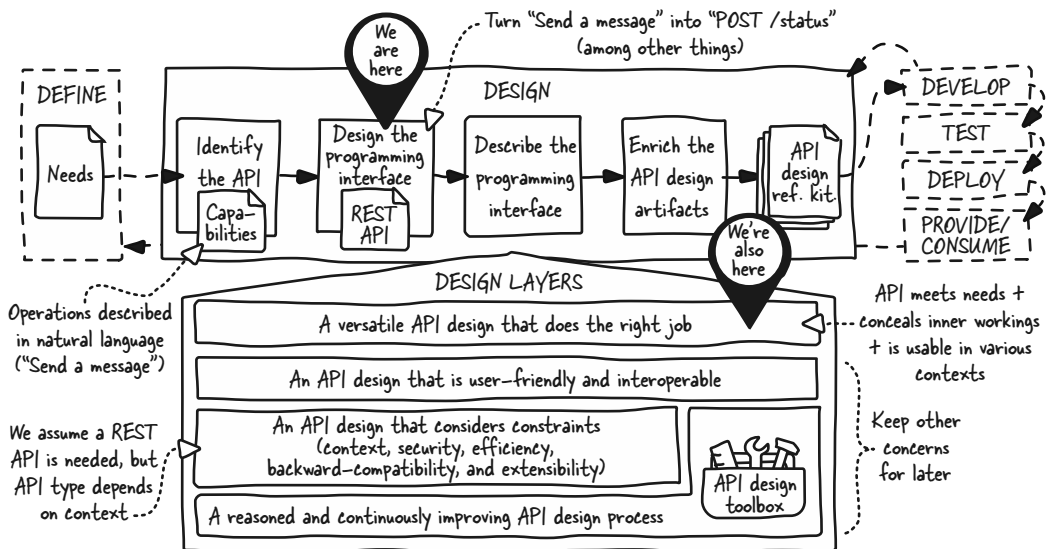


Figure 3.1 We design the REST programming interface based on the capabilities identified during the needs analysis. We focus on designing a versatile API that does the right job.



CAUTION We assume that a REST API suits our needs, but this API type may not be the most adapted option in our context; section 14.8 discusses typical alternatives and when to select an API type.

This section explains the basics of the HTTP protocol and REST APIs. We then provide an overview of how to design a REST programming interface and explain why we didn't discuss these concerns when identifying capabilities.

3.1.1 Introducing the basics of the HTTP protocol

As mentioned in section 1.1.2, REST APIs are web APIs that use Hypertext Transfer Protocol (HTTP), like web browsers. HTTP is a synchronous request–response protocol that enables interaction with resources via standardized methods. A resource can be anything and can be represented in requests or responses in various formats, such as HTML, videos, or PDFs. HTTP methods allow retrieving (GET) and sending (POST) resources, among other basic interactions. HTTP enables communication between clients and servers, independent of technologies and implementations. Figure 3.2 shows we can retrieve my blog posts using a web browser or the `curl` command by providing the <https://apihandyman.io/blog> URL.

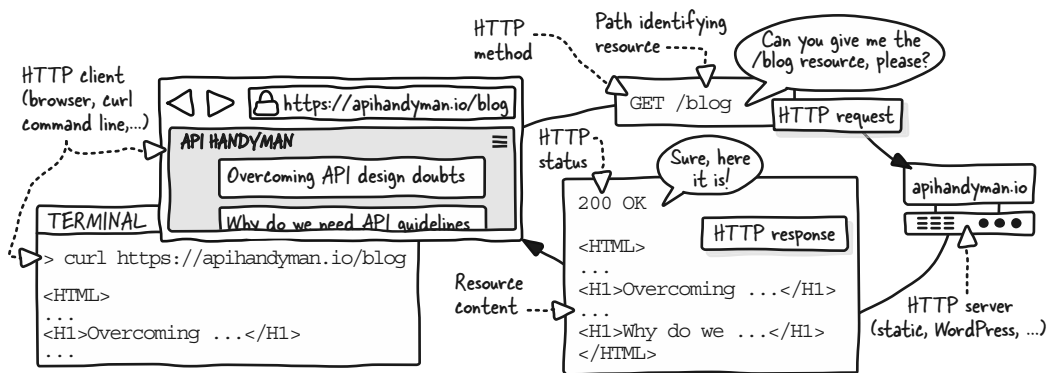


Figure 3.2 A browser or `curl` command loads a web page by sending an HTTP request containing a standard method and a path identifying the page. The server returns with an HTTP response confirming that the page is found and its content.

Both tools (clients) send a `GET /blog` HTTP request to the `apihandyman.io` server. The `GET` method requests a resource, and the `/blog` path identifies it. The server responds with a `200 OK` success status indicating that the resource is found, and returns the HTML code for the requested page. The browser parses this HTML, retrieving additional resources like JS, CSS, or images it references with the same mechanism. The `curl https://apihandyman.io` command outputs the returned HTML directly.

Listing 3.1 Retrieving a web page in Python

```
import requests
page = requests.get("https://apihandyman.io/blog")
print(page.text)
```

HTTP can be used in any programming language. Listing 3.1 shows a Python script acting like the example `curl` command. A Swift, Kotlin, or Java mobile application can send the same HTTP request. Whether the server is a WordPress PHP application or a static server, it will return the same HTTP response.

3.1.2 Introducing the basics of REST APIs

REST APIs can be used by applications with or without end users (section 1.1.1) and built in any technology, as any programming language supports HTTP (section 3.1.1). For now, we'll consider REST APIs as web APIs that extensively use HTTP and respect its semantics, although we'll see they're more than that in section 4.8. Figure 3.3 shows a call to the “Online Shopping” API made by a mobile application and the `curl` command-line tool to “Search for products.”

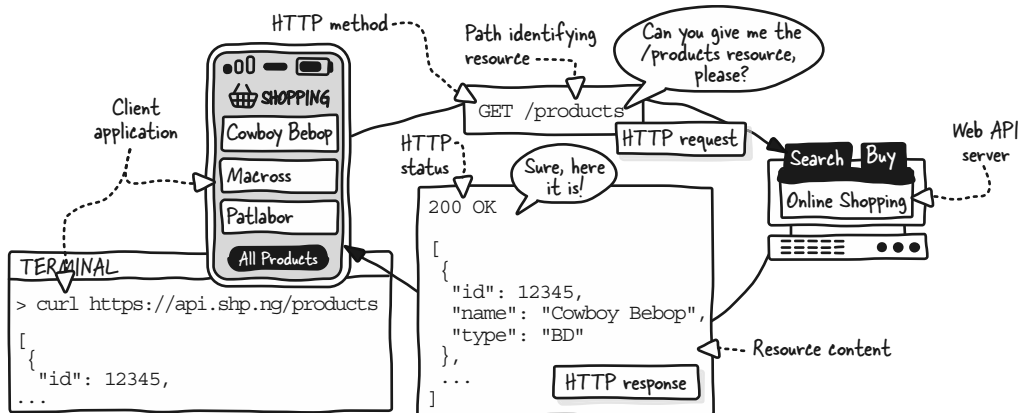


Figure 3.3 How a client application calls a REST API is identical to when a web browser loads an HTML page.

The application sends a `GET /products` HTTP request to the API server, which responds with a `200 OK` status and the product list, regardless of how data is stored and read. From an HTTP perspective, this REST API call is similar to the blog example in section 3.1.1. However, the resource represents a business entity or concept (“products”) related to “Online Shopping” instead of a web resource like HTML or CSS files, and the client gets structured data instead of HTML. The `curl` command line, a web application, a smart refrigerator application, or any application or program speaking HTTP can perform the same REST API call.



NOTE The data is in JSON format, but that doesn't matter now. Section 5.1.2 discusses this format, and section 9.7 demonstrates using other formats, such as XML and CSV.

The resource and method model may look familiar to those accustomed to object-oriented programming (OOP). An HTTP resource can be compared to an object or class, and the HTTP methods to the methods of an object or class. However, unlike OOP, HTTP is limited to standardized methods.

3.1.3 Contrasting REST with non-HTTP-compliant web APIs

Although all web APIs use HTTP, not all web APIs respect HTTP semantics like REST APIs. Figure 3.4 contrasts deleting a nonexistent product with a REST and non-HTTP-compliant web API.

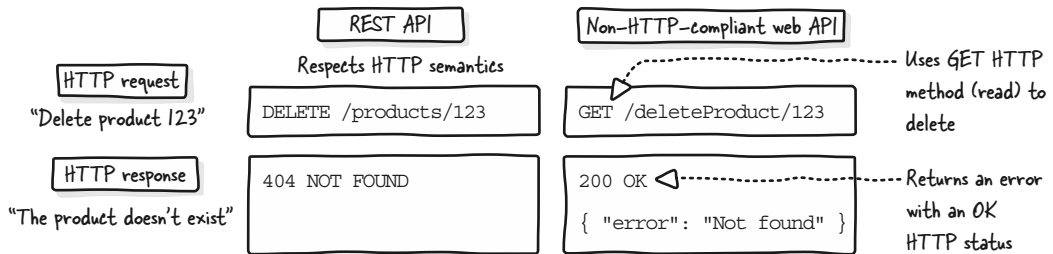


Figure 3.4 Some web APIs don't respect HTTP semantics. Their creator may wrongly call them “REST” simply because they use HTTP.

A REST API respecting HTTP semantics allows a client application to send a `DELETE /products/123` request to delete a specific product resource. If the product doesn't exist, the server responds with `404 Not Found`, similar to a nonexistent web page request. Conversely, an API ignoring HTTP semantics may use `GET /deleteProduct/123` and return `200 OK` with a “Not Found” error message, misusing the `GET` method for deletion and signaling an error with a success code, contrary to HTTP standards.



WARNING It may make sense for some web APIs to not use the HTTP protocol extensively and to use it as a transport layer. Still, it's best to stay within the protocol's definition to be interoperable and user friendly (see section 9.1.3).

3.1.4 How do we design a REST programming interface?

Designing a REST programming interface efficiently and accurately requires separating concerns, as illustrated in figure 3.5. We observe the operations found when identifying API capabilities (“Search for products”) from the REST angle (see sections 3.2 and following). We identify resources (“Catalog”), their relations (“Catalog contains Product resources”), the actions that apply to them (“Search”), their inputs, and success and error outputs.

Using identified elements, we represent operations with HTTP (see section 4.1). We design paths representing resources (`/products` for “Catalog”), choose standard HTTP methods to represent actions (`GET` for “Search”), pick HTTP status codes to indicate success or failure, and locate inputs and outputs data in requests and responses.

Ultimately, we design fine-grained data models (see section 5.1). We design the data of resources, operation inputs, and outputs. We identify, name, and type each piece of data (a product has a category of type `string`) and organize them in objects or arrays.

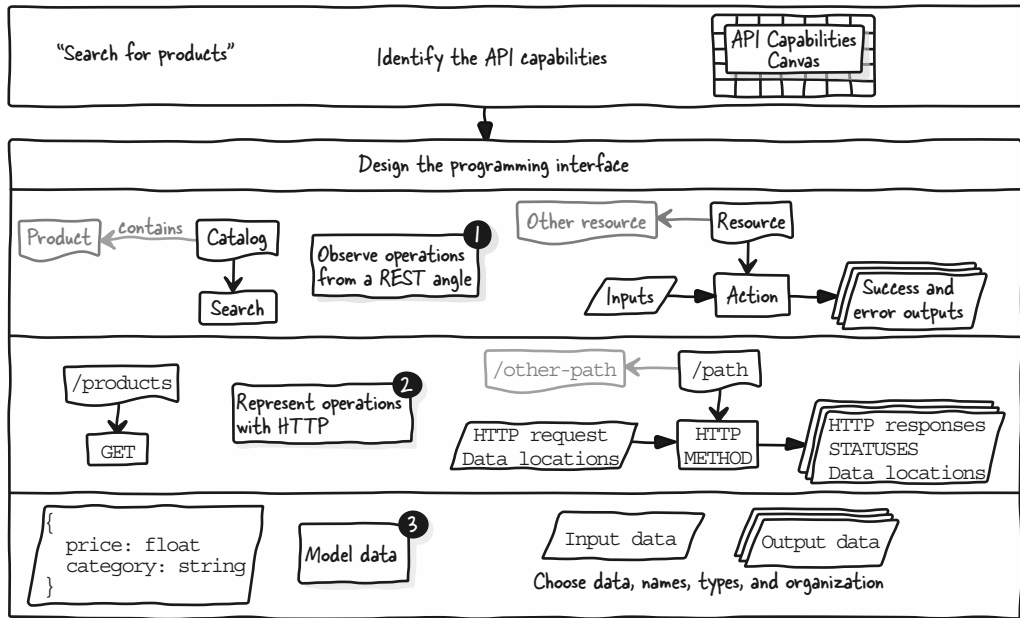


Figure 3.5 To efficiently and accurately design the programming interface representing the API capabilities, we observe the operations information of the API Capabilities Canvas from the REST angle, represent identified elements with HTTP, and, finally, model data.

3.1.5 Why not discuss HTTP and REST when identifying capabilities?

With experience, it may become evident that “Search the catalog” can translate to GET /products. However, we must always remember section 2.1.4: avoid discussing REST or HTTP when identifying API capabilities. Experience does not exempt us from assessing user needs to determine whether REST is suitable or whether another API type is required. We still face discussions around using /products, /product, or /catalog and deciding on appropriate HTTP statuses. This can complicate conversations with subject matter experts (SMEs), shifting focus from user needs and subject matter.

Our experience with the resource and standard methods model can bias our thinking, leading to flawed designs that do not meet user needs. Early attempts to identify HTTP representations can slow us down and increase error risks. For example, when designing an API for a library guided by our HTTP knowledge, we may focus only on the “Books” and “Book” resources with methods like GET /books (“search for books”) and GET /books/{bookId} (“read a book”). However, does DELETE /books/{bookId} (“delete a book”) make sense? Do these operations accommodate a library’s multiple book copies and handle the borrowing process? Speaking of the borrowing process, how do we represent it with HTTP? It’s more an action than a business concept (discussed in section 4.7). Identifying all capabilities without thinking

about HTTP and REST will help us meet user needs and be exhaustive and will provide us with information to help us overcome such challenges.



NOTE We avoid burdening SMEs with REST and HTTP concerns. However, their subject-matter expertise is essential for identifying resources (business concepts) and modeling data when designing the programming interface.

3.2 Observing the API Capabilities Canvas from the REST angle

The rest of this chapter focuses on observing the API Capabilities Canvas (section 2.2) from the REST angle to spot the REST elements needed to represent operations with HTTP (covered in section 4.1).

We use the API Capabilities Canvas in figure 3.6 to learn how to perform this task. It contains a subset of elements from chapter 2’s “Online Shopping” example, representing the five most typical API operations: searching, reading, creating, updating, and deleting things. These are often called CRUD operations; CRUD stands for create, read (also applies to search), update, and delete.

USER	USE CASE	STEP	INPUT	SUCCESS	FAILURE	OPERATION
End users	Buy products	Search for products to buy	Catalog, filters	Products matching filters	No product found	Search for products
		Check a product detailed info.	Selected product	Product info.	Product doesn't exist	Get product details
Catalog admins	Fill catalog	Look for similar products	Catalog, characteristics	No product found	Products matching characteristics	Search for products
		Verify if product is different	Found product	Product info.	Product doesn't exist	Get product details
		Add a product to the catalog	Product, catalog	Product is in catalog	Wrong product info. (not added to cat.)	Add a product to the catalog
		Modify product info.	Selected product, modified info	Product is updated	Product doesn't exist	Modify a product
		Remove a product from the catalog	Selected product	Product is removed	Product doesn't exist	Remove a product from the catalog

Figure 3.6 This API Capabilities Canvas showcases the five typical API operations: searching for elements and creating, reading, updating, and deleting an element (CRUD).

This section reorganizes the canvas’s information around operations and expands it to save findings. Then we provide an overview of how to observe operations from the REST angle to uncover what we seek.

3.2.1 Reorganizing and expanding the API capabilities canvas

To observe operations from the REST angle, we can reorganize the API Capabilities Canvas around operations and expand it to save our findings. Currently, the canvas is organized by use cases, with the same operation appearing in multiple steps, complicating our task. A spreadsheet can simplify reorganization. As shown in figure 3.7, we can create a pivot table with OPERATION as the main column, followed by INPUT, SUCCESS, FAILURE, STEP, USE CASE, and USER (check <https://apihandyman.io/the-design-of-web-apis/> for an example document). Alternatively, we can filter by OPERATION to select steps for a specific operation. Avoid sorting by operations to maintain the order of use-case steps.

OPERATION	INPUT	SUCCESS	FAILURE	STEP	USE CASE	USER
C Add a product to the catalog	Product, catalog	Product is in catalog	Wrong product info. (not added to cat.)	Add a product to the catalog	Fill catalog	Catalog admins
	Catalog, filters	Products matching filters	No product found	Search for products to buy	Buy products	End users
R Search for products	Catalog, characteristics	No product found	Products matching characteristics	Look for similar products	Fill catalog	Catalog admins
	Selected product	Product info.	Product doesn't exist	Check a product detailed info.	Buy products	End users
R Get product details	Found product	Product info.	Product doesn't exist	Verify if product is different	Fill catalog	Catalog admins
	Selected product, modified info	Product is updated	Product doesn't exist	Modify product info.	Fill catalog	Catalog admins
U Modify a product	Selected product	Product is removed	Product doesn't exist	Remove a product from the catalog	Fill catalog	Catalog admins
D Remove a product from the catalog						

Figure 3.7 Pivoting the API Capabilities Canvas to group information around operations simplifies our work. CRUD stands for create, read, update, and delete.

We'll save resource- and operation-related findings in the Operations and Resources tables in figure 3.8; we can add them as new sheets in our API spreadsheet. We can copy the OPERATION column of the pivot table to start filling the Operations sheet with the unique names we defined during API capabilities identification.

Creating a pivot table

To create a pivot table in Google Spreadsheet or Excel, select all cells in the sheet containing the capabilities. In the Insert menu, choose Pivot Table. Then insert it into a new sheet. Add the columns starting with OPERATION in lines (or rows) in your desired order. Finally, uncheck all the "totals" check boxes.

OPERATION	RESOURCE	ACTION	INPUT	OUTPUT			RESOURCE	RELATION
				Description	Type	Data		
Add a product to the catalog								
Search for products								
Get product details								
Modify a product								
Remove a product from the catalog								

Resources

Operations

Figure 3.8 We expand the API Capabilities Canvas with the Operations and Resources tables to keep track of our findings.

3.2.2 How to observe operations from the REST angle

As shown in figure 3.9, our input is the output of the API capabilities identification: the API Capabilities Canvas in our case. Going through each operation's information, we identify resources (or business concepts) manipulated by the operations and how they are related (section 3.3), which actions apply to them, and their inputs and outputs (section 3.4). SMEs can significantly contribute to observing operations from the REST angle; this task only requires using plain language and relies on subject matter expertise and vocabulary. Once done, we can move to the next task of programming the interface design, representing these elements with HTTP (see section 4.1).

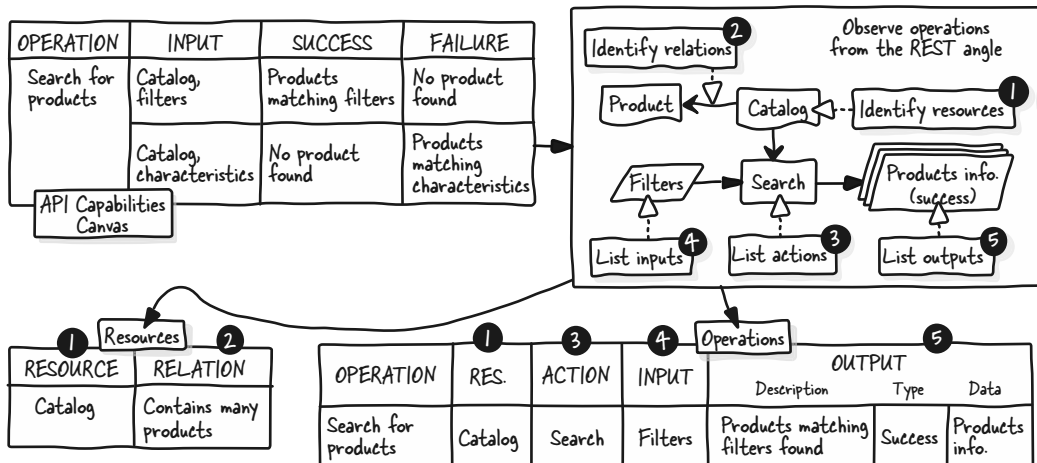


Figure 3.9 Using the operations information from the API Capabilities Canvas, we identify resources, their relations, and the actions that apply to them with their input and outputs.

3.3 Identifying resources and their relations

The observation of the API Capabilities Canvas from the REST angle starts with identifying resources manipulated by operations and how they are related (see section 3.2.2). This section first discusses what a resource is. Then we demonstrate how to identify resources and their relations using the five typical CRUD operations from section 3.2.1. Finally, we uncover patterns and recipes to simplify this task.

3.3.1 What is a resource?

In sections 3.1.1 and 3.1.2, we introduced the concept of resources when discussing HTTP and REST. In HTTP, a resource is virtually anything, such as an HTML file or an image, represented by a path (`/blog` or `/thumbnails/blade-runner.jpg`, for example). It can be manipulated with standard HTTP methods (such as `GET` or `POST`) and hence operations.

The same is true for a REST API, but before being represented by a path (we keep this for section 4.2.2), a resource is a high-level business concept or entity related to the API's subject matter. It has a noun or short description using domain terminology. A resource can exist independently and be manipulated alone. It differs from its properties, the small data pieces that compose it (see section 5.1). Typically, a resource is a class in object-oriented programming.

In our “Online Shopping” example, the “Product” is a key subject matter concept that appeared often when we identified the API capabilities (see section 2.1): this is a resource. Conversely, the “name” of the product is a piece of information belonging to it that can't exist independently. A counter-example is the product's price. It is a product property, but our analysis may reveal that we must also treat prices as resources to track their evolution over time.

3.3.2 Identifying an operation's resource

As shown in figure 3.10, we use an operation's description, input, success, and failure to identify the resource it manipulates. The operation's resource is often the target of

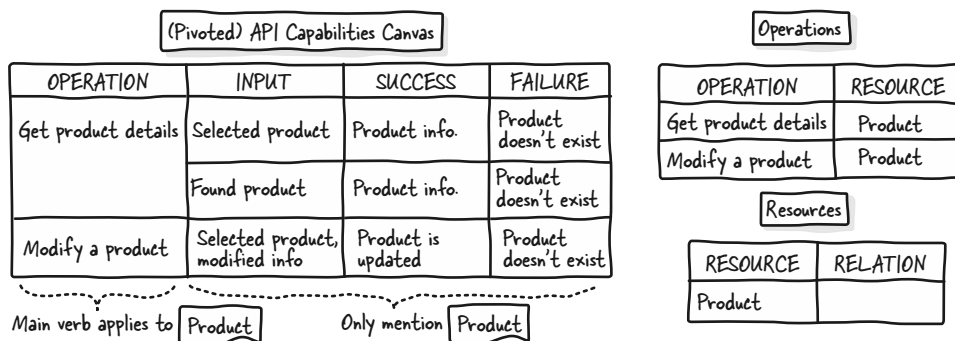


Figure 3.10 We use description, input, success, and failure to identify the resource manipulated by an operation.

its description's main verb. In “Modify a product,” the verb “modify” applies to “product.” We can assume the resource is “Product.” Looking at this operation's input, success, and failure confirm it; they all focus on the concept of “Product.” Similarly, in “Get product details,” the verb “get” applies to “product details,” and the inputs, success, and failure focus on “Product.” Both operations manipulate the same resource.



NOTE An operation manipulates only one resource, and a resource can be manipulated by different operations.

3.3.3 Tweaking an operation's description to identify resource

Shortening (figure 3.11) or expanding (figure 3.12) descriptions sometimes helps better identify an operation's resource. Shortening operation descriptions aids in identifying resources. For “Add a product to the catalog,” we may hesitate between the “Product” and “Catalog” resources. We can simplify the description to “Add to the catalog,” concluding that the resource is “Catalog.” Additionally, although both “Product” and “Catalog” are mentioned as input, success or failure relates to whether the

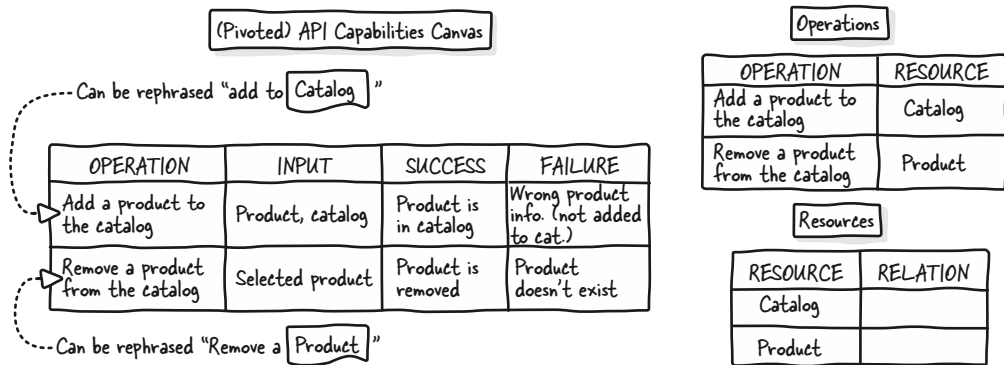


Figure 3.11 We can shorten the description to identify an operation's resource.

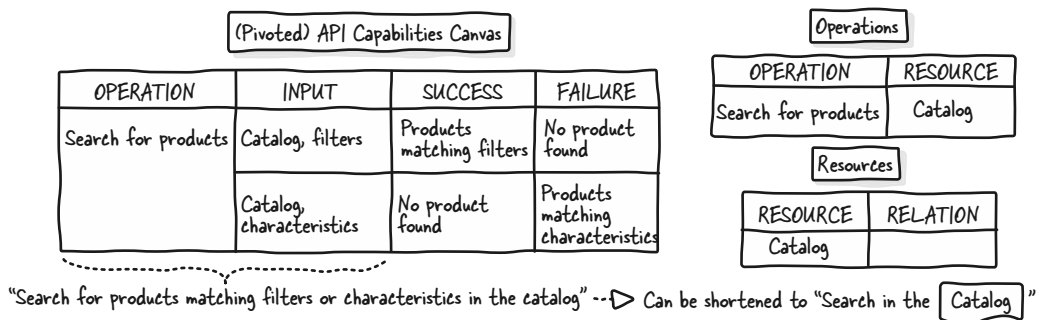


Figure 3.12 We can expand the description to identify an operation's resource.

product is added to the catalog, confirming that the main concept the operation deals with is “Catalog.” Similarly, we can condense “Remove a product from the catalog” to “Remove a product,” indicating that this operation manipulates a “Product,” as confirmed by the input and outcomes that focus solely on “Product.”

Alternatively, we can expand the operation’s description to identify its resource. For example, “Search for products” requires “Catalog” and “Filters” or “Characteristics.” We can expand its description to “Search for products matching filters or characteristics in the catalog” and shorten it to “Search in the catalog.” “Catalog” is the resource we search for, just like in the previous two operations.



NOTE If tweaking the description doesn’t help determine the resource’s identity, it may be because we’re dealing with a nested or hierarchical resource. For example, if an operation is “Add a new product to a merchant’s catalog,” we can shorten it to “Add to the merchant’s catalog,” but not “Add to catalog.” The resource is “merchant’s catalog,” not just “catalog.”

3.3.4 Identifying resource relations

Once we’ve analyzed all operations and determined their resources, we can identify their relations using our subject-matter knowledge and the API Capabilities Canvas information. We can also look for relations when adding a new resource to our list. Note that resources may not have relations, depending on the subject matter.

As shown in figure 3.13, from the “Online Shopping” subject matter perspective, it’s pretty evident that a “Catalog” (of products) contains many elements of type “Product,” and a “Product” belongs to a “Catalog.” In the API Capabilities Canvas, the “Search for products” and “Add a product to the catalog” operations and “Products matching filters” and “Product is in catalog” successes confirm this relationship.

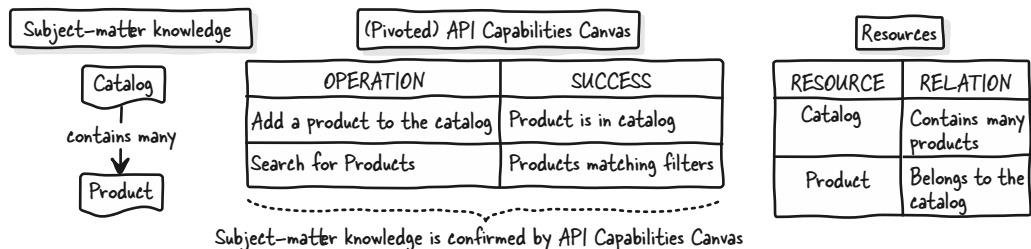


Figure 3.13 We can use subject-matter knowledge and the API Capabilities Canvas to identify resource relations.

3.3.5 Using patterns and recipes to identify resources and relations

Analyzing descriptions and using inputs and outcomes are fundamental for identifying resources and their relations. Still, we’ve discovered recipes applicable whenever we encounter typical patterns, such as CRUD operations.



NOTE Throughout the design process, recognizing typical patterns and applying proven recipes facilitates the design work, helps us be more confident in our design decisions, and contributes to creating excellent APIs. See section 16.1 to learn more about streamlining API decision-making.

The resource is the element when reading, updating, or deleting an element. The resource is the element's container when creating or adding an element to a container or when listing or searching for elements belonging to a container. How we describe relations between resources may depend on the subject-matter terminology, but we'll usually end with "X belongs to Y" or "Y contains X" relations.



CAUTION Identifying resources and their relations is similar to defining classes or tables. Use your preferred methods, but avoid being influenced by preexisting code or databases (see section 2.8).

3.4 Identifying resource actions

Consumers interact with resources through standardized HTTP methods when using a REST API. As seen in section 3.1.4, we need to identify the specific action an operation applies to the resource we identified in section 3.3 to be able to choose the appropriate HTTP method representing it (section 4.3). In the process, we'll list the action's inputs and outputs, for which we'll select locations in HTTP requests (section 4.4) and responses (section 4.6). We'll model the input and output data in section 5.1. This section explains what an action is and how to identify it, and lists its inputs and outputs.

3.4.1 What is an action, and how can it be identified?

Each operation applies an action to its resource, described by the main verb from the operation's description. This verb is the same one we used to identify the resource (see section 3.3.2), which is why we can simultaneously identify an operation's resource and action. Figure 3.14 shows the enhanced operation descriptions we used when identifying resources, allowing us to connect the two tasks.

Operations		
OPERATIONS	RESOURCES	ACTIONS
Add a product to the catalog	Catalog	Add
Search for products (in the catalog)	Catalog	Search
Get product details	Product	Get
Modify a product	Product	Modify
Remove a product from the catalog	Product	Remove

The main verb is the action

Figure 3.14 We use the operation description's main verb to identify the action.

In “Add (a product) to the catalog,” the main verb is “Add”; it is the action applied to the “Catalog” resource by this operation. Similarly, with “Search (for products) in the catalog,” the main verb/action is “Search.” The same goes for the three other operations: “Get product details,” “Modify a product,” and “Remove a product (from the catalog).” Their actions are, respectively, “Get,” “Modify,” and “Remove.”



CAUTION Don’t jump ahead when identifying actions (especially once you’ve learned to map them to HTTP methods in section 4.3). Use raw verbs from operation descriptions, and avoid replacing them with CRUD verbs or HTTP methods.

3.4.2 Listing an action’s inputs

Each operation’s action inputs merge the inputs of all steps using the operation. We describe them in a context-agnostic way to avoid duplicates, as when identifying operations in section 2.5.2.

When different steps use an operation, we merge inputs, as shown in figure 3.15. Two steps use the “Get product details” operation, whose inputs are “Selected product” and “Found product.” They both identify a specific product found with “Search for products,” regardless of its use. We discuss with SMEs what they usually use to identify a particular product; it’s a “Product reference.” We add it to the action’s inputs.

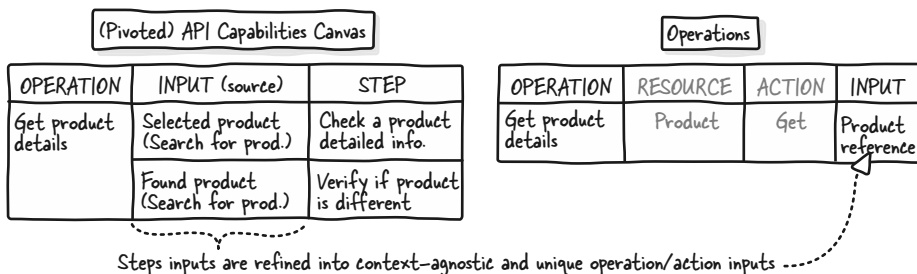


Figure 3.15 We can merge multiple context-specific step inputs into a unique, context-agnostic action input.



NOTE Naming can be difficult; if there are uncertainties, we can revise our choices during detailed data modeling. Section 8.8 covers the art of naming, and section 8.9.2 discusses selecting identifiers.

The task is more straightforward when the operation’s action is used on a single step, as shown in figure 3.16. Similar to the previous example, both “Modify a product” and “Remove a product from the catalog” expect a “Selected product,” so we add a “Product reference” to their action’s inputs. Modifying a product also requires “Modified information,” which we can make more explicit with the “Modified product information” description at the action level.

(Pivoted) API Capabilities Canvas			Operations			
OPERATION	INPUT	STEP	OPERATION	RESOURCE	ACTION	INPUT
Modify a product	Selected product, modified info	Modify product info.	Modify a product	Product	Modify	Product reference Mod. prod. info.
Remove a product from the catalog	Selected product	Remove a product from the catalog	Remove a product from...	Product	Remove	Product reference

Figure 3.16 No merging is needed when a single step uses the operation, but we may improve descriptions.



NOTE Remember, we don't need all the details at that stage; we'll look into fine-grained data, such as what goes into "Product information" when modeling data (section 5.1). For input parameters and output data related to pagination or filtering, it's not a problem if we miss them now; later, we'll work on API design efficiency to ensure that we don't miss them (section 13.1).

3.4.3 Dealing with the operation's resource when listing an action's inputs

The action inputs usually exclude the operation's resource, but exceptions may exist. They may help us spot elements we missed during the needs analysis.

Figure 3.17 shows that the "Add a product to the catalog" operation has a "Product" input, which we turn into "Product information" in the action inputs to avoid confusion with the product resource. The "Catalog" operation input source is the API; it's the operation's resource. If there are multiple catalogs, we add a "Catalog reference" to the action inputs to identify the catalog to work with and investigate a new use case, "Manage catalogs." If there's only one catalog, we don't add it to the action's input. After discussing this with SMEs, we decided to keep the one catalog option.

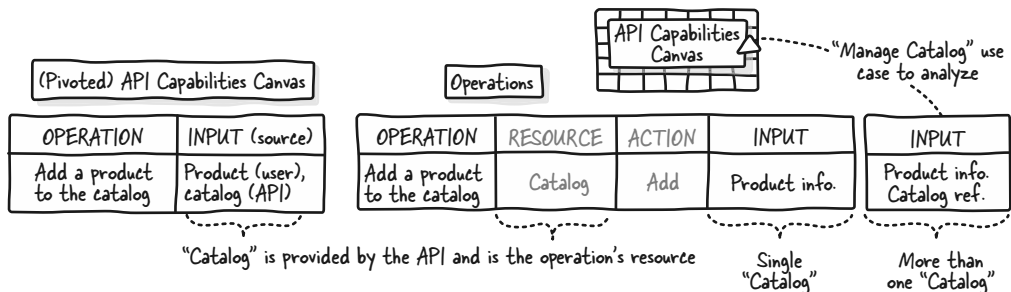


Figure 3.17 We can uncover a new use case when including the operation resource in the inputs.

Figure 3.18 shows the "Search for products" operation in two steps: "Search for product to buy" and "Look for similar products." It also has the "Catalog" input, but we

excluded it because it is the operation resource with only one catalog. After discussing this with SMEs, we merged “Filters” and “Characteristics” as “Filters,” both of which are search criteria that allow users to find specific products.

(Pivoted) API Capabilities Canvas			Operations			
OPERATION	INPUT (source)	STEP	OPERATION	RESOURCE	ACTION	INPUT
Search for products	Catalog (API), filters (user)	Search for products to buy	Search for products	Catalog	Search	Filters
	Catalog (API), characteristics (user)	Look for similar products				

Figure 3.18 We merge the inputs of the different steps and exclude the operation resource from the action’s inputs.



NOTE Listing an action’s inputs may reveal inputs with different names that are the same. If unsure, keep them separate, and reevaluate during data modeling.

3.4.4 Listing an action’s outputs

For each operation’s action, we build a single output list containing all successes and failures. Each case has a description, type (success or error), and data (if any). As for inputs, we merge elements from different steps and describe them in a context-agnostic way.

Figure 3.19 shows that “Get product details” is used in the steps “Check a product detailed info” and “Verify if product is different,” with the same success and failure outcomes. The success outcome is “Product information,” added to the actions outputs list with the description “Product found,” type “Success,” and data “Product information.” The failure outcome is “Product doesn’t exist,” added to the list with the description “Product not found,” type “Error,” and no data.

(Pivoted) API Capabilities Canvas				Operations			
OPERATION	SUCCESS	FAILURE	STEP	RESOURCE (ACTION)	OUTPUT		
Get product details	Product info.	Product doesn't exist	Check a product detailed info.	Product (Get)	Description	Type	Data
	Product info.	Product doesn't exist	Verify if product is different		Product found	Success	Product info.
					Product not found	Error	

Merged into a single success output Merged into a single error output

Descriptions can be rephrased

Figure 3.19 We merge different steps’ outcomes and rephrase descriptions when listing the action’s outputs.



CAUTION Mixing apples and oranges is often a problem. An operation returning heterogeneous content, particularly in lists, often indicates incorrect operation or data identification. For example, if an operation returns “Books” and “Toothbrushes,” these could be grouped as “Products.” If it returns “Products” and “Providers,” it may need to be separated into two operations. However, a product detail with a provider summary makes sense. Refer to section 8.7 for more information.

We proceed similarly for “Add a product to the catalog,” “Modify a product,” and “Remove a product from catalog” operations, all of which are used by a single step (figure 3.20). The “Add” action’s outputs are “Product added to the catalog” (Success, no data) and “Wrong product information” (Error, no data). The “Modify” action’s outputs are “Product modified” (Success, no data) and “Product not found” (Error, no data). The “Remove” action’s outputs are “Product removed” (Success, no data) and “Product not found” (Error, no data).

(Pivoted) API Capabilities Canvas				Operations			
OPERATION	SUCCESS	FAILURE	STEP	RESOURCE (ACTION)	Description	Type	Data
Add a product to the catalog	Product is in catalog	Wrong product info.	Add a product to the catalog	Catalog (Add)	Product added to the catalog	Success	
					Wrong product information	Error	
Modify a product	Product is updated	Product doesn't exist	Modify product info.	Product (Modify)	Product modified	Success	
					No product found	Error	
Remove a product from the catalog	Product is removed	Product doesn't exist	Remove a product from the catalog	Product (Remove)	Product removed	Success	
					No product found	Error	

Figure 3.20 No merge is necessary when a single step uses the operation.

3.4.5 Dealing with contradictory successes and failures when listing outputs

Defining success or error for an API operation must be independent of the use case and consumer. It relies on the operation’s nature, input and output data, and subject matter. When calling the API, the consumer will interpret the response as success or error according to its specific context.

In the “Search for products” operation shown in figure 3.21, the success and failure outcomes of the two steps contradict each other. Finding products to buy is a success, and finding none is an error. However, looking for similar products to avoid duplicates in the catalog is the opposite.

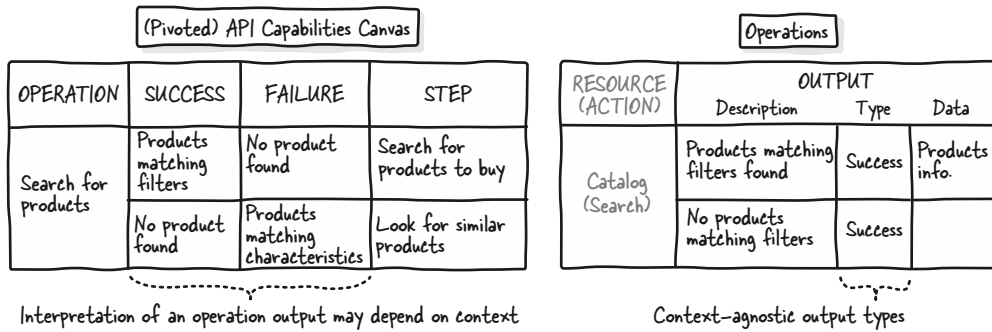


Figure 3.21 We must turn consumer-context-specific steps' outcomes into context-agnostic operation/action outputs.

We add both “Product matching filters found” (with “Products information” data) and “No products matching filters” (with no data) to the action outputs. Then we choose their type from a context-agnostic perspective. We set the “Product matching filters found” type to “Success” because that is how search operations usually behave. We can find both options for “No products matching filters” when looking at other APIs. Still, we choose “Success” because that’s the most common and user-friendly option (discussed further in section 9.8). For now, that’s a pattern to remember: search and list operations do not error when they find nothing.



NOTE Design decisions, such as making a search operation that finds nothing successful, should be applied consistently to all future designs to make our APIs user-friendly. Check section 16.3 to discover how defining API design guidelines can help you achieve consistent designs.

Similarly, suppose that reading a product returns product information indicating an empty stock. In a use case where an end user wishes to buy it, this can be interpreted as a failure. However, it’s a success in another use case where an admin wishes to verify that this product is no longer available. In both cases, the operation success output is the product information.

Summary

- HTTP is a synchronous, request–response protocol that enables interactions with resources through standardized HTTP methods.
- An HTTP resource can be anything, such as an HTML file or data in any format.
- REST APIs use HTTP extensively and respect its semantics.
- To design a REST API, observe operations from the API Capabilities Canvas from the REST perspective, represent the identified elements with HTTP, and model data.
- Observe operations from the REST perspective to identify resources, actions, inputs, and outputs, continuing with plain languages like English.

- The five typical API operations are searching for elements and creating, reading, updating, and deleting an element, also called CRUD operations.
- A resource is a standalone business concept distinct from properties.
- An operation uses a single resource, which several operations can use.
- The resource is the target of the main verb in the operation description.
- When creating/adding and listing/searching elements, the resource is the container of the elements; when reading/updating/deleting, the resource is the individual element within the container.
- An action is the main verb that applies to the resource manipulated by an operation.
- An action inputs list merges the inputs of the steps using it; use context-agnostic descriptions to avoid duplicates.
- The operation's resource should be removed from the action's input list unless multiple instances exist.
- An action's outputs list merges the success and failure of the steps using it. Each output has a description, type (success or error), and optional data.
- Choose an action's output type from a context-agnostic perspective. Consumers may interpret success and error differently based on their context.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 3.1

You're designing an API for an online course platform where teachers and students interact in various ways. Following are descriptions of some of the platform's operations for teachers. For each, identify the resources, relations, and actions.

- 1 Set up a new course offering, including a syllabus, difficulty level, and a list of topics to cover.
- 2 Look for available courses by topic or difficulty level.
- 3 Verify information about a course, including syllabus and topics.
- 4 Modify a course difficulty level, syllabus, or topics.

Exercise 3.2

You're designing an Air Travel API and have identified the following operations and step inputs. What is the resource, and what are the actual operation inputs for each operation?

- Operation: Search for flights. Step inputs: airport, destination, flights, departure date.

- Operation: Search for flights. Step inputs: date, airline, flights.
- Operation: Book a flight. Step inputs: flight, date, passenger, flights.

Exercise 3.3

You're designing an API for a fitness and wellness application. By identifying and analyzing the following operation outputs, can you spot and explain what's wrong and why?

- View meal plan logs and sleep logs
- Search for trainers
- List workout history and nutrition advice logs
- Get today's dashboard (today's workouts and meals and last night's sleep logs summary)

Exercise 3.4

You're designing an API for a library system and have found that steps of different use cases using the "Read a book" operation have contradicting success and failure outcomes. What should be the operation's success output?

- The step fails if a reader discovers the book has been borrowed.
- The step is a success when a librarian can confirm a book has been borrowed.

4

Representing operations with HTTP

This chapter covers

- Designing paths for resources
- Mapping typical actions to HTTP methods
- Representing successes and failures with HTTP status codes
- Choosing data locations in HTTP requests and responses
- Representing “do” operations with HTTP
- Using the REST architectural style

After reviewing the API Capabilities Canvas from a REST perspective and identifying resources, actions, inputs, and outputs, we can translate “Send a message” into a `POST /messages` or `POST /message` request. However, challenges can arise in representing the resource (“message”) with a path (`/messages` or `/message`) and mapping the freely defined action “Send” to standard HTTP methods like `POST`. These examples reveal only a few aspects of a request; more considerations are needed for accurately mapping all the identified elements to HTTP requests and responses. Additionally, not all operations fit neatly in typical create, search, read, update, and

delete categories; some are simply verbs, such as “like.” These verb-based “do” operations can complicate HTTP representation.

We’ll break the work into steps to overcome these challenges for easier execution and learning. We’ll dive into HTTP and look at common practices to understand our decisions. This will help us identify patterns and recipes that are applicable to typical operations to streamline design. While doing this, we may fill gaps in our analysis, as it’s expected to overlook some data or error cases in previous steps. We’ll learn enough to discuss the origin of REST APIs: the REST architectural style. Its principles are essential for creating REST APIs and can also be used for other types of web APIs and remote APIs.

This chapter first outlines how to represent operations with HTTP, detailing the resulting HTTP request and response and the steps taken to achieve them. We then explore each step in depth. The chapter also covers representing “do” operations with HTTP. Finally, we introduce the REST architectural style and how to use its principles in API design.

4.1 *Representing operations with HTTP*

As shown in the zoomed API lifecycle (section 1.6) in figure 4.1, we continue designing the programming interface, as discussed in section 3.1.4. After observing the API Capabilities Canvas operations from the REST angle (section 3.2), we can represent

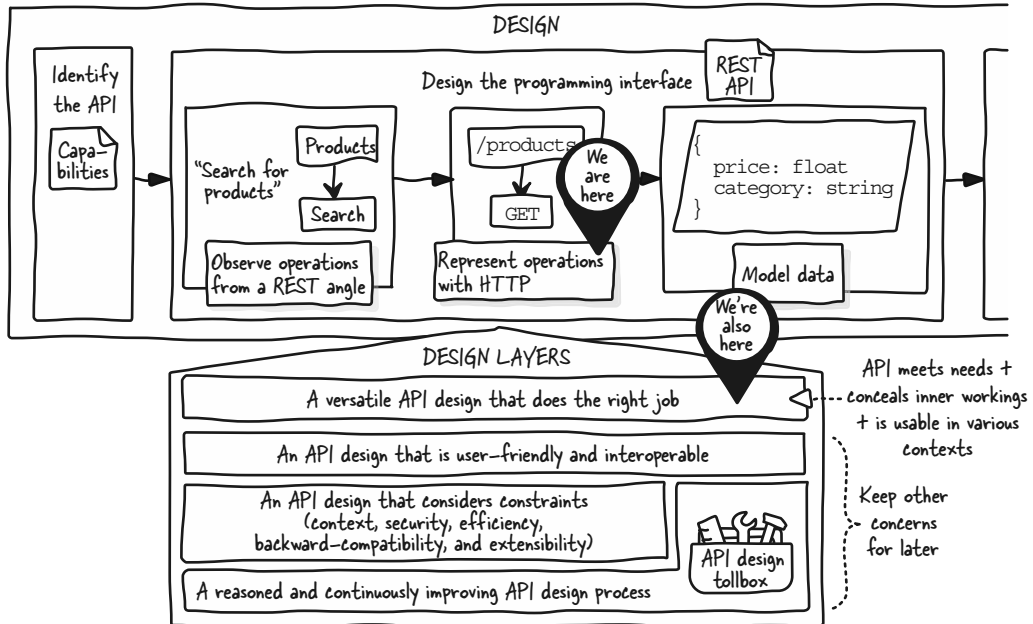


Figure 4.1 After identifying API capabilities, including operations, we observed them from the REST angle. We identified resources, actions, inputs, and outputs that we can now represent with HTTP.

the identified resources, actions, inputs, and outputs shown in figure 4.2 with HTTP. This section provides an overview of the final HTTP representation of an operation and what we focus on in this chapter. The next step, modeling data, is discussed in section 5.1. We continue to focus on the “versatile API that does the right job” layer, addressing consumer needs, concealing inner workings, and ensuring usability in various contexts (section 1.7.1).

Capabilities identified by analyzing needs

Information collected when observing the operation capabilities from the REST angle

OPERATION	RESOURCE	ACTION	INPUT	OUTPUT		
				Description	Type	Data
Add a product to the catalog	Catalog	Add	Product info	Product added to the catalog	Success	Product info.
				Wrong product information	Error	
Search for products	Contains many	Search	Filters	Products matching filters found	Success	Products info.
				No products matching filters	Success	
Get product details		Get	Product reference	Product found	Success	Product info.
				No product found	Error	
Modify a product	Product	Modify	Product reference, Modified product info	Product modified	Success	
				No product found	Error	
Remove a product from the catalog		Remove	Product reference	Product removed	Success	
				No product found	Error	

Figure 4.2 When observing the operations in the API Capabilities Canvas of the Online Shopping API from the REST angle, we identified resources, the actions that apply to them, and their inputs and outputs.

4.1.1 What an operation looks like with HTTP

Before discussing how to represent operations with HTTP, we look at the final result and how an application uses it. Figure 4.3 shows a possible HTTP representation of the “Search for products in the catalog” operation of the Online Shopping API. This operation accepts search filters and returns the products matching them.

The mobile application searches for Blu-ray (BD) products by sending a `GET /products?type=BD` HTTP request to the API server. The `GET` HTTP method represents “search,” and the `/products` path is the “catalog of products.” The `type=BD` is a query parameter used as a search filter. The HTTP response has a `200 OK` status, which indicates that no problems were encountered. The response body contains the list of products matching the filters.

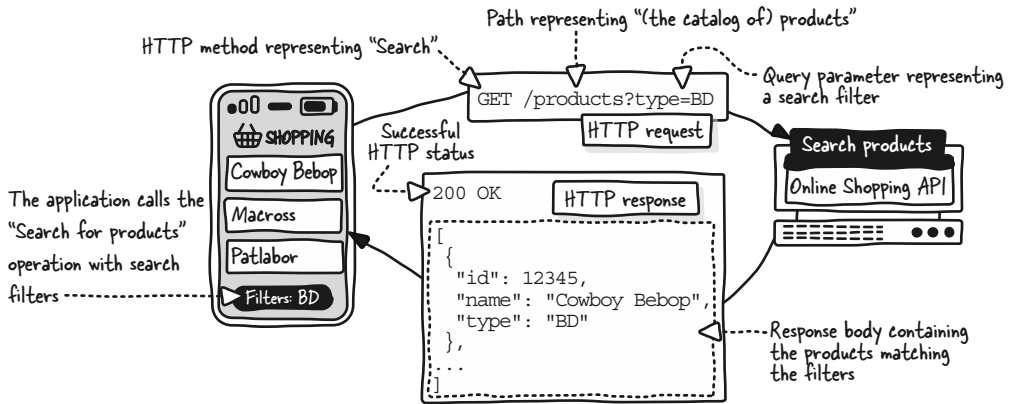


Figure 4.3 The mobile application searches the catalog for BD products. The API server returns the products with type BD.

4.1.2 How to represent operations with HTTP

When observing the operations from the REST angle, we identified resources, actions that apply to them, and their inputs and outputs. Figure 4.4 shows how we'll represent them with HTTP.

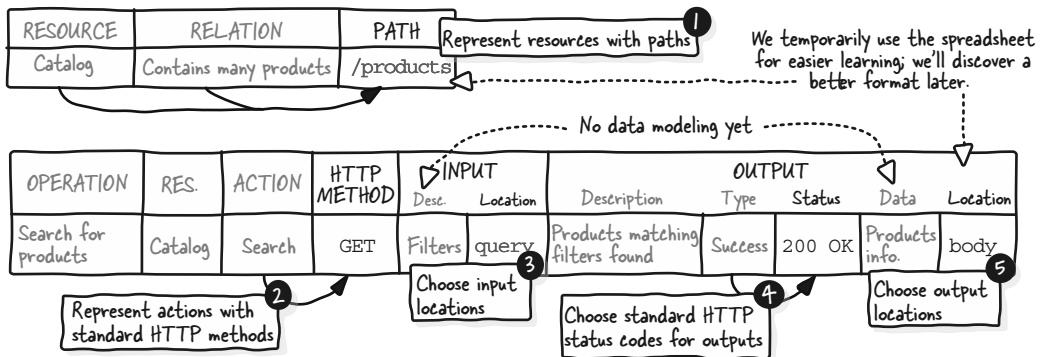


Figure 4.4 We focus on paths, HTTP methods, HTTP statuses, and the locations of input and output data in HTTP requests and responses.

We start with the HTTP request, representing the resource ("Catalog with many products") with a path (`/products`). We select an HTTP method (`GET`) for the action ("Search") and choose the appropriate locations for each piece of input data (query for "Filters"). Next, we address the HTTP responses, choosing status codes representing the output types and descriptions (200 OK for the successful "Products matching filters found") and locations for output data (body for "Products info").

Modeling the detailed data of elements such as “Filters” input (products can be filtered by type, price, etc.) and “Products info” output (a product as an ID, name, price, etc.) is done in the next step (see section 5.1). We temporarily store our findings in our API spreadsheet to separate learning concerns; we will discover a more suitable format when describing the programming interface (see section 6.1).

The following sections detail this process for the five typical operations of the Online Shopping API and teach recipes that will help us streamline representing operations with HTTP and use HTTP correctly.



NOTE Although it’s not our focus yet, the recipes will help us proceed similarly across operations and APIs, making them interoperable and user-friendly (see section 1.7.2). For HTTP-related questions not covered in this book, refer to the documentation (starting with RFC 9110, www.rfc-editor.org/rfc/rfc9110.html); it may even help you find solutions to design web APIs. See section 16.1 for searching design solutions.

4.2 Representing resources with paths

As seen in section 3.1.2, REST APIs rely on resources (such as the catalog business concept) identified by a path (`/products`, for example). This section first explains the basics of resource paths. Then, using the catalog and “Product” resources of the Online Shopping API identified in section 3.3.2, we cover the following considerations:

- Designing meaningful resource paths
- Targeting specific elements with path parameters
- Materializing the relations between resources
- Representing lists and their elements

4.2.1 What is a resource path?

The resource path in `https://api.server.com/path` is `/path`, identifying a unique resource. Two resources can’t share the same path on a web API or HTTP server, similar to files on a filesystem. However, different paths may lead to the same resource.



NOTE We’ll discuss other path elements that may precede the actual resource path (`/shopping/v1/path`). Section 11.3.3 covers the API name (`shopping`), and section 15.4.3 discusses its version (`v1`).

Although a path identifies a resource, it doesn’t reflect the underlying data or its organization. For instance, the `/something` path doesn’t correspond to the `/something` filesystem folder or `something` database table. Remember the provider’s perspective in section 2.8.

A path (`/this/is/a/path`) can have segments (`this`, `is`, `a`, `path`) separated by slashes (`/`). It may include path parameters denoted by curly braces (`/segment/{someParameter}`) or prefixed with a colon (`/segments/:someParameter`) in API documentation or code. Path parameters are filled with values (`/segment/123abc`) in an

HTTP request. Multiple path parameters can appear anywhere in the path, inside segments, and combined with characters (`{a}/{b}-{c}12/{d}text{e}`).



TIP A resource path with a trailing slash, such as `/path/`, can cause routing bugs at the implementation code or network infrastructure levels, leading to hours of debugging. Use `/path` instead.

4.2.2 Designing meaningful resource paths

Our first concern when designing paths is to ensure that they best represent the resources. Figure 4.5 shows different paths uniquely identifying the “Catalog” resource; some choices are more meaningful than others.

RESOURCE	PATH			
A collection of products Catalog	Random /xyz	Abbreviation /cat	Resource name /catalog	Resource content /products
	Unique but not meaningful		Unique and meaningful	

Figure 4.5 From a REST perspective, a path only needs to be unique. However, we should ensure that our paths mean something to our API users.

We can randomly choose `/xyz` for the “Catalog” resource path; it can uniquely identify a resource. However, it’s not evident that it represents a catalog. It’s best to call a spade a spade; `/catalog` is a good option that is unique and meaningful. Abbreviated names are common in programming, but `/cat` is less clear than `/catalog`. A name based on the resource’s content works, too. Because a catalog is a list or collection of products, we can also consider the `/products` path. Before deciding, we’ll discuss possible paths for the “Product” resources the catalog contains.



CAUTION Users won’t easily understand cryptic or abbreviated names; we may not even remember their meaning six months later. See section 8.8 to learn the art of naming.

4.2.3 Targeting specific elements with path parameters

Some resources can’t be uniquely identified by name, as shown by the “Product” resource paths in figure 4.6. This section discusses the first three paths, and section 4.2.4 covers the others, focusing on the “Catalog” and “Product” relationship.

The `/product` path is meaningful but can’t uniquely identify a product. During our REST angle analysis, we found that all operations using the “Product” resource require a unique product reference input (see figure 4.2 in section 4.1). We can use it as a path parameter, giving `/ {Product reference} (/123456)`.

RESOURCE	PATH (123456 is a {Product reference} path parameter)	UNIQUE	MEANINGFUL	HIERARCHICAL
A single Product	/product Resource name	X	✓	n/a
	/123456 Resource identifier	X	X	n/a
A single Product in the Catalog collection	/product-123456 Resource name and identifier	✓	✓	n/a
	/123456/catalog Identifier and parent name	✓	✓	X
	/catalog/123456 Parent name and identifier	✓	✓	✓
	/products/123456 Resource type and identifier	✓	✓	✓

/{Product reference} may collide with /{Supplier reference}

Figure 4.6 The “Product” resource path requires a unique resource identifier path parameter. Multiple segments can show the relationship between “Catalog” and “Product” resources.

Although unique, `/123456` lacks meaning, and a future `/123456` could conflict with it (`/123456` versus `/123456`); even noncolliding IDs would complicate implementation. Combining the resource name and reference solves this problem. The `/product-123456` path suggests, “I’m the unique product 123456.” However, we should explicitly show the relationship between “Catalog” and “Product” resources.



NOTE A *resource identifier* is an ID, reference, or code that appears in all the resource operations; add it to the resource path to ensure unique identification. The `{Product reference}` name is temporary; we’ll specify the actual path parameter name when modeling data (see section 5.3.1). Section 8.4.4 covers resource identifier selection, and section 9.3.3 discusses using multiple identifiers and path parameters for a resource path.

4.2.4 Showing resource relationships with a hierarchy

By taking advantage of HTTP paths’ hierarchical nature, we can indicate the relationship between different resources in a path, making our paths more meaningful. In `/parent/child`, the parent contains one or more children, and the child is an element of a parent.

The last three paths in figure 4.6 try to materialize that a product is an element of the catalog. The `/123456/catalog` (`/123456/catalog`) path could be read as “Product 123456 belongs to the catalog,” but its hierarchy is reversed (child comes before parent). The `/catalog/123456` (`/catalog/123456`) path fixes this, saying, “The catalog contains product 123456.” We can also use `/products/`

{Product reference} (/products/123456), which states, “Product 123456 belongs to the list of products.”



NOTE The last segment of a hierarchical path defines its fundamental meaning; /w/h/a/t/e/v/e/r/products refers to products. Segments before the last one influence interpretation; /merchants/{merchant id}/products/{product reference} represents a specific product in a merchant’s catalog. A nonhierarchical path, such as /products/merchant/{product reference}/{merchant id}, is more complicated to interpret, especially at runtime (/products/merchant/123/456). Section 9.3 further discusses path structure, hierarchy, length, and number of path parameters.

Now that we have different options for the catalog and its products, we can decide which paths to choose.

4.2.5 Representing lists and their elements

All of the previous catalog and product path combinations are valid from a REST perspective. However, the common approach for a list or collection resource with elements is to use an /elements path for the collection and /elements/{element resource identifier} for child elements, as shown in figure 4.7. Such paths are meaningful and establish resource relationships. Consumers can simply concatenate the element ID to the collection path to form the element path, like when working with a filesystem. A singular collection noun is also valid (/element and /element/{identifier}).

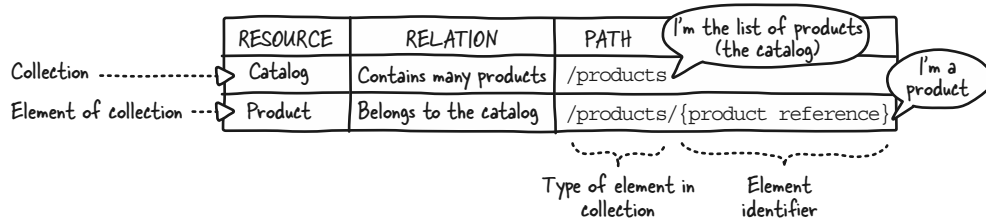


Figure 4.7 A list is represented by a noun that indicates its element type (usually plural, but can also be singular). The element identifier is added to the list path, forming the element path. This pattern is widely adopted in REST APIs.

“Catalog” is a collection resource containing “Product” resources, so we represent them as /products and /products/{Product reference}. These paths uniquely identify each resource (thanks to the product reference path parameter for the product), describe the relationship between the two (thanks to being hierarchical), and are meaningful (each clearly states what the resource is).



NOTE Although it’s not our focus yet, this pattern or recipe makes our API user-friendly by simplifying its use and making it look like all the others. Read

section 9.3 to learn more about user-friendly paths. We also streamline our work by using a reasoned, established practice. See section 16.1 to learn more.

4.3 Representing actions with HTTP methods

As seen in section 3.1.2, REST APIs represent operations with standardized HTTP methods applied on resource paths (GET /products, for example). This section introduces the HTTP methods that are commonly used and then illustrates how to choose one for each operation's action identified in section 3.4.1. Finally, we generalize these learnings with recipes for choosing an HTTP method for the five typical operations (create, read, search, update, and delete).

4.3.1 Determining which HTTP methods to use

REST APIs use five HTTP methods to represent actions: POST, GET, PUT, PATCH, and DELETE. Figure 4.8 summarizes their meaning and usage.

HTTP METHOD	MEANING (HTTP)	MEANING (CRUD)	ACTION EXAMPLE	
POST	Create	C	Create, add, start, save, send	
	Process	non- CRUD	Do, execute	
GET	Read	R	Read, get, search, filter, select, retrieve, show, download	
PUT	Replace	U	Replace, modify, update, change, edit	From the consumer perspective, both allows partial update
	Create	C	Create, add, start, save, send	
	Upsert	C U	Create if not present, update otherwise	
PATCH	Update	U	Replace, modify, update, change, edit	
DELETE	Delete	D	Delete, cancel, close, finish, stop	

Figure 4.8 Refer to this mapping when in doubt about choosing an HTTP method for an action. **CRUD** stands for create, read, update, delete.

POST usually represents a creation (C of CRUD), adding an element to the targeted resource. Use it for actions such as “create,” “add,” “start,” “save,” and “send.” However, its real meaning is broader. It means “process according to resource’s significance” and can be a fallback when no other method works.

GET reads the resource (R of CRUD). Use it for actions such as “read,” “get,” “search,” “filter,” “select,” “retrieve,” “show,” and “download.”

PUT is for resource replacement or update (U of CRUD), creation (C of CRUD), and upsert, which updates an existing resource or creates a new resource if one doesn’t already exist (CU of CRUD). Use it for actions like “modify,” “update,” “change,” “replace,” and “edit,” or the same actions as POST for creation.

PATCH also updates a resource (U of CRUD) and can be used for the same update actions as PUT. PATCH is generally used for partial updates (modifying a subset of

resource data); however, from the consumer perspective, a partial update can be performed with either `PUT` or `PATCH`. With `PUT`, all of the resource data containing the few needed modifications is sent; with `PATCH`, only the modified data is sent (see section 5.3.4).

`DELETE` represents a deletion (D of CRUD). Use it for actions such as “delete,” “cancel,” “close,” “finish,” and “stop.”



NOTE What actually happens internally when processing an HTTP request depends on subject matter and implementation choices. A `DELETE /something` request can hard- or soft-delete data by updating a flag. Likewise, `POST` or `PUT /something` can also trigger data deletion. However, the implementation must comply with the “idempotent” or “safe” nature of HTTP methods. No worries if these terms are unfamiliar; we’ll discuss these concerns in section 12.5.2.

4.3.2 Choosing HTTP methods to represent actions

We must select the HTTP methods that best represent resource actions. However, an HTTP method can be defined only once for each resource. Conflicts are rare and usually indicate a wrong resource or operation identification; reevaluate them in such cases.

Figure 4.9 shows the HTTP methods for the five typical REST API operations of the Online Shopping example (we designed the resource paths in section 4.2). The following sections explain these results from simple to complex cases. In practice, you’ll work on one resource after another, which helps detect HTTP method conflicts.

OPERATION	RESOURCE	ACTION	HTTP METHOD	RESOURCE PATH
Add a product to the catalog	Catalog	Add	POST	/products
Search for products		Search	GET	
Get product details	Product	Get	GET	/products/{Product reference}
Modify a product		Modify	PUT or PATCH	
Remove a product from the catalog		Remove	DELETE	

Figure 4.9 We added the action and HTTP method mapping to the operation table of the API spreadsheet.

4.3.3 Representing search, read, and delete actions

The “Catalog” resource has a “Search” action, but it’s fundamentally a “Read” action we can map to the `GET` HTTP method. To “Search for products,” consumers send a `GET /products` HTTP request.

The “Product” resource has a “Get” action, which we can easily map to the `GET` HTTP method. To “Get product details,” consumers send a `GET /products/{Product reference}` HTTP request.

The “Product” resource has a “Remove” action of type “Delete,” so we choose the `DELETE` HTTP method. To “Remove a product from the catalog,” consumers send a `DELETE /products/{Product reference}` HTTP request.



NOTE Without refining the use-case steps in section 2.5.2, we would have “Search for products” and “Look for similar products” operations, leading to “Search” and “Look” actions. This results in conflicting `GET` methods on the “Catalog” resource and makes us realize we can merge these two operations.

4.3.4 Representing update actions

The “Modify a product” operation of the “Product” resource has a “Modify” action, which is an update; we can map it to `PUT` or `PATCH /products/{Product reference}`. At this learning stage, `PUT` and `PATCH` can be used interchangeably. I recommend choosing `PUT` as it fits most cases and is simpler to implement. It is possible to have both methods defined. We’ll keep both to demonstrate the five usual HTTP methods.



NOTE Later sections will show other perspectives and help us make a reasoned choice between `PUT` and `PATCH`. Section 5.3.4 covers basic data formats, section 13.5 discusses efficiency concerns and advanced data formats, section 14.2.5 covers implementation questions, and section 12.5.1 discusses expected implementation behavior.

4.3.5 Representing create actions

The “Catalog” resource has an “Add” action to add a product. It’s a creation we can represent with `POST` or `PUT`. Using `POST /products` creates a product and returns the reference usable with `GET /products/{Product reference}`. A `PUT /products` would replace the entire catalog, which is not what we want. We must use `PUT /products/{Product reference}` to create a single product with `PUT`.

Choosing the `PUT` option requires combining “Add a product to the catalog” and “Modify a product” into “Adding or modifying a product” and allowing the consumer to provide the product reference. Creating a resource based on user-provided IDs works with predefined and globally unique IDs, such as book ISBNs. However, if the reference is a random number, it’s better to use `POST /products` and let the system generate the reference to avoid errors from colliding IDs. At this stage of learning, I recommend choosing `POST` for creations; it’s the most commonly used pattern.



NOTE Learning more about resource IDs (section 8.4.4) and the HTTP-induced implementation behavior for `POST` and `PUT` (section 12.5.2) will help you decide between the two for creation operations.

4.3.6 Mapping typical operations to HTTP

We've uncovered new patterns and recipes that apply whenever we need to map one of the typical create, search, read, update, or delete operations to HTTP.

Given that `/elements` represents a list or collection of elements and `/elements/{element identifier}` represents one of its elements, the five typical REST API operations can be mapped to HTTP as follows:

- *Create an element*—POST `/elements` (default choice at this stage) or `PUT /elements/{element identifier}`
- *List or search for elements*—GET `/elements`
- *Read an element*—GET `/elements/{element identifier}`
- *Update an element*—PUT (default choice at this stage) or `PATCH /elements/{element identifier}`
- *Delete an element*—DELETE `/elements/{element identifier}`

4.4 Choosing input data locations in HTTP requests

After choosing a path and an HTTP method, we can decide on the locations of inputs in the HTTP request. This section discusses possible locations for data in an HTTP request and explains how to choose locations based on the nature of the input data when designing an API. We also generalize what we learn in recipes that are applicable to the five typical operations (create, read, search, update, and delete).



NOTE See section 5.1 for data modeling. Section 9.4.1 discusses the effect of input data location on usability, and section 12.6 covers security considerations.

4.4.1 Where to put input data in an HTTP request

This section explains the possible data locations in an HTTP request from a pure HTTP perspective. As shown in figure 4.10, these locations are the path, query parameters, and header fields and the body, which only specific methods support.

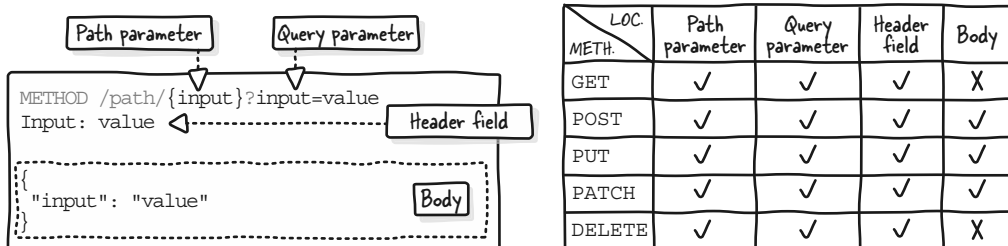


Figure 4.10 An HTTP request can contain data in path parameters, query parameters and header fields as well as the body, which is only available for POST, PUT, and PATCH.

HTTP doesn't explicitly define path parameters; as seen in section 4.2.1, path parameters are located in the resource path on the first line of the HTTP request. Their value can be anything that fits in a (usually short) string. For example, the path `/resources/{Resource identifier}/sub-resources/{Sub-resource identifier}` could become `/resources/12/sub-resources/ab` in an HTTP request.

The resource's path can be completed with query parameters containing nonhierarchical data; their order doesn't matter. They participate in resource identification. They're after a `?` in `name=value` format, separated by `&` for multiple parameters. Names can be chosen freely, with values fitting in strings like path parameters. For example, `/resources?a=1&b=no&c=true` is equivalent to `/resources?c=true&b=no&a=1` and contains three query parameters: `a`, `b`, and `c`, using number, string, or Boolean values.

Header fields come after the first line and contain metadata about the request, such as its origin, content, security, or cache concerns. Over 200 standard headers are defined (see the IANA HTTP Field Registry at www.iana.org/assignments/http-fields/http-fields.xhtml). Their format is `name: value`. Standard headers will usually meet our needs, but we'll discuss using custom ones in later chapters. As for query and path parameters, the value fits in a string; for example, `Content-length: 345` shows the request body size in bytes.

The request body follows the headers and is used in `POST`, `PUT`, and `PATCH` but not `GET` or `DELETE`. It can contain text or binary data, such as HTML, JSON, XML, or images. The body is a “representation” of the resource to create or update. The server may not store it exactly as sent; it can be converted for storage in an SQL database, for example. The representation concept is further discussed in section 9.7.

4.4.2 An overview of input data natures

The HTTP method and the data's nature influence the input data's location. We can categorize inputs into

- *Resource identifiers*—Data identifying the resource the operation interacts with (product reference)
- *Resource representations*—Resource data to create or update (product information and modified product information)
- *Resource modifiers*—Parameters tweaking the data (filters)

The following sections discuss the nature and location of the input data for the typical operations of the Online Shopping example shown in Figure 4.11. In the field, you'll work on one resource at a time, like HTTP methods.



NOTE We'll learn more about the “why” behind these results in section 9.4.1, which compares all location possibilities and their effects on API usability.

OPERATION	RESOURCE	ACTION	HTTP METHOD	Description	INPUT	Location
Add a product to the catalog	Catalog	Add	POST	Product information	Resource representation	body
Search for products		Search	GET	Filters	Resource modifier	query
Get product details	Product	Get	GET	Product reference	Resource identifier	path
Modify a product		Modify	PUT or PATCH	Product reference	Resource identifier	path
				Modified product info	Resource representation	body
Remove a product from the catalog		Remove	DELETE	Product reference	Resource identifier	path

Location is influenced by HTTP method and input data nature

Figure 4.11 We added the input data location to the operations table of the API spreadsheet based on HTTP methods and the nature of the input data (resource identifiers, representations, or modifiers).

4.4.3 Choosing a location for resource identifiers

The three operations of the “Product” resource (GET, PUT or PATCH, and DELETE `/products/{product reference}`) share the same “Product reference” input already identified as a resource identifier and path parameter (section 4.2). Still, we’ll investigate it again to learn about data location in HTTP requests for REST APIs.

This input cannot be a header because it doesn’t match any of IANA’s 200 standard headers. It shouldn’t be in the body, as GET or DELETE cannot have a body. Only PUT (or PATCH) allows body input, but it identifies rather than represents the resource to update. Hence, the input may be a path (`/products/{product references}`) or a query parameter (`/products?reference={product reference}`).

Both are valid for HTTP because the concatenation of path and query parameters identifies a resource. However, the pattern `/products` and `/products/{Product reference}` is most common from an API perspective. It shows a hierarchy, with the last segment indicating the resource we interact with. In contrast, `/products?reference={product reference}` lacks hierarchy and can be confusing: `/products` suggests either a list or a single element.



NOTE A resource identifier contributes to identifying the operation’s resource; it goes in a path parameter. If it doesn’t assist in identification, it becomes a resource modifier; refer to section 4.4.6 for an example.

4.4.4 Choosing a location for resource representations

We need “Modified product information” to modify a product (PUT or PATCH). This isn’t a header, as it doesn’t match standard headers; nor is it a path parameter, because it doesn’t identify the resource. This leaves the query parameter and body. Although we could stringify the data for a query parameter (which may cause problems;

see section 14.2.4), it must be in the body according to HTTP because it represents the resource's new state. This also applies to the production information needed to add a product to the catalog (POST).



NOTE Following HTTP terminology, we can qualify the data needed to create or update a resource as a resource representation. It goes in the request body.

4.4.5 Choosing a location for resource modifiers

To search for products (GET /products), consumers can provide “Filters” to get a subset of all products. Because the HTTP method is GET, a request body isn't allowed, and such input doesn't fit in a standard HTTP header. This leaves path and query parameters as options. For example, with filters like “type” and “description,” we could use /products/{type}/{description} (path parameters) or /products?type={type}&description={description} (query parameters).

According to HTTP, nonhierarchical data like type and description best suits query parameters, not hierarchical paths. Also, including them in the path makes them mandatory. Consumers can only search for products matching a type and description, excluding retrieving all products, products matching a specific type, or products matching a description.

We can achieve the same conclusion from another perspective. Although GET /products, GET /products?type=book&description=design and GET /products/book/design access different HTTP resources because path and query participate in resource identification, they target the same API resource (the “Catalog” entity). Therefore, type and description shouldn't be part of the catalog path; they should be query parameters.

The type and description filter inputs modify response data but don't change its nature; the operation still returns products, making the filters resource modifiers. If an input enables the return of the products in XML instead of JSON, the needed parameter also acts as a resource modifier. However, it fits in a standard Accept HTTP header (covered in section 9.7).



NOTE Resource modifiers tweak data, but the returned business entity is the same. They are included in query parameters unless they fit standard HTTP headers.

4.4.6 Hesitating between resource identifiers and modifiers

The difference between a resource identifier (path parameter) and a resource modifier (query parameter) isn't always clear, which can lead to unnecessary operations. To decide whether an element is a resource identifier or modifier, we can check the operation behavior without this input or revisit consumer needs and capabilities.

A resource identifier in one context can act as a modifier in another. For instance, a “Category reference” may identify a category when reading it (path parameter). Or

it can be a resource modifier (query parameter) to filter product searches because it isn't needed for catalog identification. Products can still be searched without it.

We can argue that a category is a clear entity in our API, making `GET /categories/{Category reference}/products` useful alongside `GET /products`. But is “Searching for a category’s products” an identified capability? Also, aren’t we mapping the UI? A category page displaying products doesn’t necessitate a specific operation (section 2.7.1). A versatile `GET /products` with necessary filters suffices (see section 9.6 for more about searching, sorting, and paginating lists). This may differ for `GET /suppliers/{supplier reference}/products`, where the “Supplier’s products” resource may fundamentally differ from products that users purchase, reflecting a different perspective on product data. However, this operation makes sense only if it’s an identified capability.



CAUTION Return to consumer needs and capabilities before creating operations derived from newly discovered possible paths. Otherwise, you risk complicating the API with unnecessary operations.

4.4.7 *Choosing input data locations for typical operations*

We’ve uncovered new patterns and recipes that are applicable any time we need to identify input locations of the typical create, search, read, update, and delete operations to HTTP:

- The data needed to create or update a resource is the resource representation and goes in the body.
- The resource identifiers participating in the operation’s resource identification go in path parameters.
- The resource modifiers that tweak the returned data go in query parameters unless they fit in a standard HTTP header.

4.5 *Representing output types with HTTP statuses*

We’re done with the HTTP request after designing resource paths and selecting HTTP methods and input locations. We can proceed to HTTP responses, starting with statuses. This section explains HTTP statuses and how to choose them for the success and error outputs discussed in section 3.4.4. We discuss how to ensure comprehensive error-handling. Finally, we summarize our findings in recipes for the five typical operations.

4.5.1 *What is an HTTP status?*

An HTTP response’s status indicates the outcome of an HTTP request. You likely already know the status `404 Not Found`, which appears when you access a nonexistent web page. An HTTP status has a three-digit code (`404`) and a descriptive reason (`Not Found`), as shown in figure 4.12.

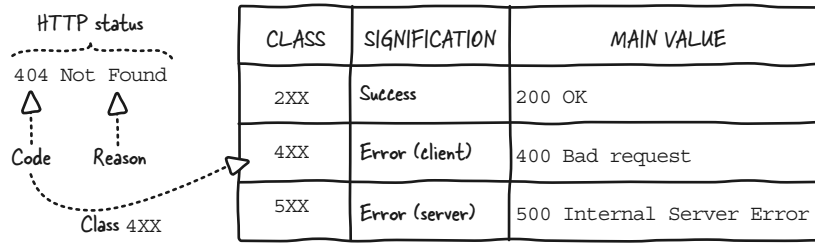


Figure 4.12 An HTTP status has a numeric code and human-readable reason. The class system allows for easily identifying whether the code signifies a success or an error and who is at fault.

The codes ranging from 100 to 599 are grouped in five classes: 1XX to 5XX, each with a specific meaning. We'll focus on 2XX (success), 4XX (client error), and 5XX (server error), which most APIs use:

- 2XX class codes 200 to 299 indicate that the server has successfully processed the request.
- 4XX class codes 400 to 499 indicate client/consumer errors. The server can't process requests due to problems such as unparsable data, unhandled HTTP methods, missing properties, business logic checks, or insufficient rights.
- 5XX class codes 500 to 599 indicate server/implementation errors caused by unexpected problems (a bugged implementation throwing a null pointer exception or an inaccessible database server, for example) or planned unavailability.

The class system simplifies interpreting unknown code. Even if you don't know the 413 HTTP status code, its 4XX class indicates that it's a consumer error.



NOTE Some APIs also use 3XX (redirection); section 10.4.7 uses it for operations flow optimization and section 14.5.1 for file uploads. The 1XX (Informational) class is uncommon and can also be used for file uploads (section 14.4.3).

4.5.2 Choosing HTTP statuses for outputs

We can proceed operation by operation to choose the HTTP statuses best representing each output listed in section 3.4.4, using what we discover from one to the next. At this stage, selecting an HTTP status code depends on the following:

- The type of output: success (2XX) or error (4XX, 5XX)
- In case of error, who caused it: consumer (4XX) or provider (5XX)
- The HTTP method of the request

This book showcases commonly used HTTP statuses in REST APIs, covering most cases. The HTTP methods documentation may provide recommendations (see the IANA HTTP Method Registry at www.iana.org/assignments/http-methods/http-methods.xhtml). For more codes, refer to the IANA HTTP Status Code Registry (www.iana.org/assignments/http-status-codes/http-status-codes.xhtml). Use caution

when using unusual codes not referenced in this book; they could surprise users. When in doubt, use the `x00` main value of a class.



WARNING HTTP classes facilitate interpretation, but don't use unassigned codes to create custom statuses. Doing so will only cause compatibility problems and confusion; see section 9.5.1.

In the following sections, we'll treat successes (2xx) and then errors (4xx or 5xx) to facilitate learning. In reality, proceed operation by operation and output by output.

4.5.3 Choosing successful HTTP statuses for read operations

Figure 4.13 shows the five typical operations of the Online Shopping example and their success outputs with corresponding HTTP statuses from the 2xx (success) class. This section and the following ones explain the results for each typical operation.

OPERATION	RES.	ACTION	HTTP METHOD	OUTPUT			
				Description	Type	Data	HTTP status
Add a product to the catalog	Catalog	Add	POST	Product added to the catalog	Success	Missing data!	201 Created
Search for products	Catalog	Search	GET	Products matching filters found	Success	Products info.	200 OK
				No products matching filters	Success		200 OK
Get product details	Product	Get	GET	Product found	Success	Product info.	200 OK
Modify a product	Product	Modify	PUT or PATCH	Product modified	Success	Missing data!	200 OK
Remove a product from the catalog	Product	Remove	DELETE	Product removed	Success		204 No Content

Figure 4.13 We added the HTTP status corresponding to each successful output and spotted missing output data in the process.

Consumers can “Get product details” by sending a `GET /products/{Product reference}` HTTP request. We could look at each 2xx code documentation to find the best one to represent this success. However, it is faster to check the `GET` HTTP method documentation, which says a `GET` request usually returns a `200 OK` response.

4.5.4 Choosing successful HTTP statuses for delete operations

The “Remove a product from the catalog” operation uses the `DELETE` HTTP method, whose documentation gives three options: `200 OK`, `202 Accepted`, and `204 No Content`. We can use `200 OK` if the action has been executed and the response contains data

describing its status. We can use 202 Accepted if the action will likely succeed but has not yet been executed. 204 No Content is similar to 200 OK, but the response contains no data. For our case, where deletion is instantaneous and doesn't return data, 204 No Content is the best option. Read section 14.7.1 to see 202 Accepted in action.

4.5.5 Choosing successful HTTP statuses for update operations

The “Modify a product” operation uses PUT or PATCH methods. Navigating the documentation, we found that the HTTP status options are the same as those for DELETE. As the update is instantaneous, we have to choose between 200 OK and 204 No Content. The output description, which says “Product modified,” isn't clear about returning data (200) or not (204). Both options are OK; we choose the most common one, 200 (section 13.5 discusses the pros and cons of these options). That means we need to add output data; see section 4.6.2.

4.5.6 Choosing successful HTTP statuses for search operations

Based on what we've seen with GET and DELETE, “Search for products” (GET /products) can return 200 OK when “Products matching filters found” and 204 No Content when “No products matching filters.” We can also use 200 OK for both; we choose this option because it is the most commonly used and enables having the same behavior when finding products or not (see section 9.6 for a detailed explanation).

4.5.7 Choosing successful HTTP statuses for create operations

Following the POST HTTP method documentation, “Add a product to the catalog” (POST /products) returns 201 Created as we create a resource. It is the same if we use PUT /products/{product reference}, which allows the consumer to differentiate between an update (200 OK) and a creation (201 Created).

The HTTP documentation reveals a missing piece: a creation should return the information of the created resource or at least minimal data so that it can be retrieved later. Section 4.6.2 shows how to detect and fill such a gap.

4.5.8 Choosing error HTTP statuses

Section 4.5.1 taught us that error HTTP statuses fall into two classes: 4xx (client) and 5xx (server); determining who is responsible for the error allows us to identify the correct one. As shown in figure 4.14, we identified two errors across all operations of the Online Shopping example: “No product found” and “Wrong product information.”

The “No product found” error is caused by a consumer providing the wrong product reference, resulting in a 4xx HTTP status class. Based on our web browsing experience and the HTTP documentation, 404 Not Found is the relevant code to indicate that the /products/{Product reference} resource doesn't exist. It's important not to confound 404 and 204 No Content. In this case, returning a 204 without data would be incorrect, meaning the product exists but has no data.

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT		
				Desc.	Location	Description	Type	HTTP status
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Wrong product information	Error	400 Bad Request
Search for products	Catalog	Search	GET	Filters	query			
Get product details	Product	Get	GET	Prod. ref.	path	No product found	Error	404 Not Found
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	No product found	Error	404 Not Found
				Mod. prod. info.	body			
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	No product found	Error	404 Not Found

Figure 4.14 We added the HTTP status corresponding to each error output.

The “Wrong product information” error occurs when a consumer provides incorrect or incomplete data while adding a product, resulting in a 4xx HTTP status class. The IANA list contains several options discussed in section 9.8. At this stage of our learning, we can use the most common one: 400 Bad Request.

4.5.9 Ensuring exhaustive error-handling

Errors found during the needs analysis may not be exhaustive; this is normal as the focus is on business needs. Choosing HTTP statuses is the perfect moment to identify and fill the gaps. Figure 4.15 shows what we can detect at this stage of our learning; we’ll learn to discover more gaps throughout the book.

Some operations expecting inputs are missing errors related to improper and missing inputs: 400 Bad Request. This is the case for “Search for products” and its “Filters” and “Modify a product” and its “Modified product information.” We can check that any operation with path parameters has a “Resource not found” error and 404 Not Found; we aren’t missing any errors here.

All operations must have an unexpected server error output returning 500 Internal Server Error. Such errors can arise from implementation problems, such as accessing a null value in Java (`java.lang.NullPointerException`), or infrastructure problems, such as an inaccessible database server or full storage.



NOTE We’ll discover more error cases when discussing user-friendly errors (section 9.8), security (section 12.10), and planned interruptions (section 14.2.3).

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT		
				Desc.	Location	Description	Type	HTTP status
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Wrong product information	Error	400 Bad Request
Search for products	Catalog	Search	GET	Filters	query	Wrong filters	Error	400 Bad Request
Get product details	Product	Get	GET	Prod. ref.	path	No product found	Error	404 Not Found
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	No product found	Error	404 Not Found
				Mod. prod. info.	body	Wrong product information	Error	400 Bad Request
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	No product found	Error	404 Not Found
All operations						Unexpected server error	Error	500 Internal Server Error

Figure 4.15 Analyzing inputs helps to identify errors missed during the needs analysis. Additionally, all operations need a 500 error.

4.5.10 Choosing HTTP statuses for typical operations

We've uncovered new patterns and recipes that are applicable any time we choose the HTTP status codes representing the outputs of the typical create, search, read, update, and delete operations and that can help us detect gaps:

- Choosing HTTP statuses for successful outputs:
 - A successful creation returns 201 Created.
 - A successful read returns 200 OK.
 - A successful search returns 200 OK.
 - A successful update returns 200 OK when the updated resource is returned and 204 No Content when it's not.
 - A successful delete returns 204 No Content if no status data is returned and 200 OK if status data is returned.
- Choosing HTTP statuses for error outputs and spotting missing errors:
 - An operation expecting input query or body data must handle missing or invalid data errors and return 400 Bad Request.
 - An operation whose resource path contains one or more path parameters must handle resource not found errors and return 404 Not Found.
 - Each operation must handle unexpected server errors and return 500 Internal Server Error.



NOTE We'll discover more options throughout the book, especially when discussing user-friendly errors (section 9.8), security (section 12.10), and long operations (section 14.7).

4.6 Choosing output locations in HTTP responses

Our last task is determining the locations of output data in the HTTP response, which affects data modeling (section 5.1). This section covers data locations in HTTP responses, filling output data gaps with HTTP, and choosing locations for data identified in section 3.4.4. Finally, we generalize these learnings in recipes that are applicable to the typical API operations.

4.6.1 Where to put data in an HTTP response

Figure 4.16 shows that an HTTP response has a structure similar to an HTTP request, starting with the HTTP status in the first line instead of a method and path. Headers, an empty line, and the body follow. The headers and body have the same characteristics as in the HTTP request (section 4.4.1). Headers are formatted as `name: value`, and standard headers from the IANA Header Field Registry (www.iana.org/assignments/http-fields/http-fields.xhtml) usually suffice; we'll discuss using custom ones in later chapters. The body can include any data.

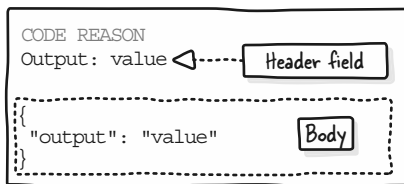


Figure 4.16 There are only two locations for data in an HTTP response: headers and body.

4.6.2 Filling the output data gaps

As seen in section 4.4, we may have missed some output elements when listing them in section 3.4.4, and HTTP can help us fill the gaps (see figure 4.17 for the final output list). The “Add a product to the catalog” operation uses `POST`, whose documentation recommends returning the representation/data of the created resource or minimal information (the product reference) to retrieve it later (with “Get product details”). It also uses `201 Created`, which requires the response to contain the URL of the created resource in a `Location` header, which allows for later retrieval with a `GET {created resource URL}` even when no data is returned.

We chose to return `200 OK` (with data) instead of `204 No Content` (without data) on the “Modify a product” operation, so we should add the updated product data as output. Similarly, for the “No products matching filters” `200 OK` output of “Search for products,” we must return data like “Empty products list.”

HTTP status codes can hint at what is happening but often fall short in case of errors. Thus the HTTP documentation recommends returning “Error information” data on all errors (see section 9.8 for more on errors).

4.6.3 Choosing output locations

Once we have an exhaustive output data list, we choose data locations (header or body) similarly to the way we did for input data in section 4.4. Figure 4.17 shows the final HTTP representation of the typical operations in the Online Shopping example, including output data locations.

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT				
				Desc.	Location	Description	Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info.	body	Product added to the catalog	Success	201	Product info.	body
									Product URL	header
						Wrong product information	Error	400	Error info.	body
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Products info.	body
						No products matching filters	Success	200	Empty products info.	body
						Wrong filters	Error	400	Error info.	body
Get product details	Product	Get	GET	Prod. ref.	path	Product found	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	Product modified	Success	200	Product info.	body
				Mod. prod. info.	body	No product found	Error	404	Error info.	body
						Wrong product information	Error	400	Error info.	body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204		
						No product found	Error	404	Error info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 204 No Content, 400 Bad Request, 404 Not Found, 500 Internal Server Error

Figure 4.17 This is the finalized operations table of the API spreadsheet, completed with all output data and its locations.

The “Product information” returned on creating, reading, or updating a product, as well as the “Products information” and “Empty products list,” are unfit for standard headers. They are representations of the resource the operation interacts with, so, as the HTTP documentation says, they go in the response body (as done in the request). HTTP also recommends putting “Error information” in the body. Strictly following the 201 Created status documentation, we return the “Product URL” of the “Add a product to the catalog” operation as a standard `Location` header.

4.6.4 *Choosing output data locations for typical operations*

We’ve uncovered new patterns and recipes that are applicable any time we need to identify output locations of the typical create, search, read, update, and delete operations in HTTP:

- A successful “read” operation (`GET`) returns the requested resource in the body.
- A successful “search” operation (`GET`) returns found elements or an empty list in the body.
- A successful “create” operation (`POST` or `PUT`) returns the created resource in the body and its URL in a standard `Location` header.
- A successful “update” operation (`PUT` or `PATCH`) returns the modified resource in the body.
- A successful “delete” operation (`DELETE`) returns nothing.
- Errors return error data in the body.



NOTE Section 13.5.2 discusses tweaking the return of created or modified resources for network efficiency concerns.

4.7 *Representing a “do” operation with HTTP*

Not all operations fit the typical create, search, read, update, and delete operations. For instance, mapping the “Check out” step of the “Buy products” use case (see section 2.3.1) to an HTTP operation is not straightforward. The challenge is figuring out the resource and the corresponding HTTP method.

As shown in figure 4.18, this section examines three ways to handle such a “do” or non-CRUD operation:

- Using an action resource
- Turning the action into a business concept
- Focusing on the action’s result.

OPERATION	RESOURCE	ACTION	HTTP METHOD AND PATH
Check out cart	Cart's checkout Action resource	Check out	POST /check-out
Create a checkout (from cart)	Checkouts Action to business concept	Create	POST /checkouts
Create an order (from cart)	Orders Action's result as business concept	Create	POST /orders

Figure 4.18 To represent a “do” action, we can create an action resource, turn the action into a business concept, or focus on the action’s result.

4.7.1 Using an action resource

The POST HTTP method signifies “Process according to resource’s signification” (section 4.3.1). A “do” operation can be represented by POST /do, where /do is the action resource path. The request body may contain data needed to execute the action, and the response body may include resulting data and input data. When it makes sense, the path should reflect the relationship between the action resource and a business concept resource, similar to the relation between a method and a class.

We can represent the “Check out” operation with a POST /check-out (or POST /carts/{Cart identifier}/check-out if multiple carts exist). It takes no input and returns 201 Created with the created order information (body) and URL (Location header, /orders/123456, for example).

Creating something is not mandatory; we can use action resources for volatile processing, such as summing two numbers with a POST /sum that expects two numbers in its body and returns 200 OK with the result.



NOTE Action resources are often misclassified as non-REST. Section “3.1 Resources” of RFC 9110 (HTTP semantics) states, “HTTP does not limit the nature of a resource; it merely defines an interface for interaction.” Thus, a resource can be a business concept, process, or action. Therefore, POST /do is valid in a REST API. However, although I keep action resources in my toolbox, I usually turn them into noun-based resources in my API designs for greater possibilities, as detailed in the following section.

4.7.2 Turning the action into a business concept

If it makes sense from a subject matter perspective, we can turn an action resource into a business concept by nominalizing its verb. Doing so also allows us to add more features than just “do.”

We can consider “Checkouts” (“check out” nominalization) as an essential business concept independent from the cart (though it uses the cart under the hood). POST /checkouts takes no data and returns checkout-related data (including an order reference) along with the /checkouts/{Checkout ID} URL to retrieve it later (“Get

checkout details” operation). Also, we can “Search for checkouts (with filters)” with `GET /checkouts`.

If you struggle to find a noun for a verb, try adding a suffix such as -ing, -ance, -ence, -ment, -tion, or -sion to the verb. For instance, “do” will become “doings,” and “execute” will become “executions.”



NOTE Turning an action into a business concept is perfect for handling long processes or operations, as covered in section 14.7.

4.7.3 *Focusing on the result*

If the action is not interesting from a subject matter perspective, we can work directly with its result and create a resource based on it. We might decide that the resulting “Order” is the crucial business concept and so represent the “Check out” operation with `POST /orders` (“Create an order from the cart”). This would return the created order (body) along with its URL (Location header, `/orders/{Order reference}`).

4.8 *Using the REST architectural style principles for API design*

To simplify learning, we’ve considered REST APIs as mapping capabilities to HTTP. However, reducing them to just that overlooks key principles. REST APIs are based on the REST architectural style, offering a foundation for efficient, scalable, and reliable remote API-based systems. This section introduces the REST architectural style, its core principles, and their importance in API design. We also discuss the often sterile debates that may arise in REST discussions.

4.8.1 *Introducing the REST architectural style*

Designing a web API involves working on a distributed system composed of software communicating over a network. A mobile application and its backend, microservices working together, and the internet are distributed systems.

The REST architectural style enables building distributed systems that are efficient (fast network communication and request processing), scalable (capable of handling more and more requests), reliable (resistant to failure), simple, portable (reusable), and modifiable. Roy Fielding developed it in his 2000 dissertation, “Architectural Styles and the Design of Network-based Software Architectures,” while working on HTTP 1.1. A REST software architecture needs to conform to the six following constraints:

- *Client/server separation*—Clients and servers must have separate and balanced responsibilities.
- *Statelessness*—Requests contain all necessary information; no client context (session) is stored between them.
- *Cache*—Responses to requests specify whether they can be reused and for how long (to avoid repeating the same call)

- *Layered system*—Clients only see and interact with servers and are unaware of the underlying infrastructure.
- *Uniform interface*—Interactions are performed via the manipulation of resources through representations of their state/data with standard methods (this is the origin of the REST acronym: representational state transfer) and the help of meta-data, enabling representation interpretation and knowing resource capabilities.
- *Code on demand* (optional)—A server can transfer executable code to the client (JavaScript, for example).

More about the REST architectural style

Fielding's dissertation is available at www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. REST is defined in chapter 5. It gives no guidance on APIs and API design.

The world has evolved, and REST has been used, misused, and abused since 2000. “Reflections on the REST Architectural Style and ‘Principled Design of the Modern Web Architecture,’” by Fielding et al., <https://dl.acm.org/doi/10.1145/3106237.3121282>, describes the history, evolution, and shortcomings of REST as well as several architectural styles derived from it.

4.8.2 Applying REST principles to API design

This book has already used or uncovered some REST constraints and will continue to do so. Note that you can use these principles for other types of remote APIs.

Section 2.6 discusses the provider and consumer perspectives, which are related to the *client/server separation* constraint. This constraint is an essential foundation for usability (section 8.1), security (section 12.1), and extensible designs (section 15.6).

The *statelessness constraint* is hidden behind the context-agnostic operation of section 2.5.2. Section 10.4 shows an example of a stateless API call flow that can support execution across different sessions.

Section 13.4 discusses enabling cache and conditional requests related to the *cache constraint*. Section 1.1.3 mentions that consumers are only aware of the API, which is the first layer of the system; it is an example of the *layered system constraint*.

In this chapter, we represented API capabilities with resources and HTTP methods, following the *uniform interface constraint*. The importance and benefits of this approach will be better understood when we discuss the interoperability of operations in section 9.10.

The *code-on-demand constraint* is mainly used in HTML and JavaScript apps but not often in REST APIs. Section 10.3 uses its spirit by providing flexible data and operation and data, enabling API call flow optimization.

4.8.3 *Debates about what is (or is not) REST*

The “What is or is not REST/RESTful?” question has sparked many heated and sterile debates that are often unrelated to REST or due to misunderstanding of REST and HTTP. Some people argue that “POST /do is not RESTful because /do is an action, not a resource.” Using an action resource is valid for HTTP, so it is valid for REST APIs. Similarly, others declare, “A collection must have a plural (or singular noun) in a REST API.” But naming conventions do not determine whether an API is RESTful.

This book has your back to help you understand the principles, apply them seamlessly, know when to make trade-offs, and evaluate the consequences of following only some principles. Beyond helping you create robust APIs, this will help you assess what to do with preexisting so-called RESTful APIs that don’t follow REST principles and barely use HTTP correctly. The focus must be not on shaming or engaging in sterile debates about the past but rather on recognizing the need for evolution in practice, addressing critical components, and discontinuing actions that may negatively affect our systems, users, and organizations.



NOTE Section 16.1 shows how to find reasoned solutions backed with sourced information (like this book) and reduce the risk of endless arguing over the same questions.

Summary

- A resource path must uniquely identify each resource (using path parameters when needed: `/resources/{identifier}`), clearly state the resource, and indicate any parent-child relations (`/parent/child`).
- A collection (list) resource path indicates the element it contains (`/elements`, for example), and its children’s resources concatenate their unique identifier and parent’s path (`/elements/{identifier}`).
- The five typical REST API operations are searching for elements and creating, reading, updating, and deleting an element (CRUD).
 - Map searching for elements to GET `/elements`.
 - Map creating an element to POST `/elements` or PUT `/elements/{element reference}`.
 - Map reading an element to GET `/elements/{element identifier}`.
 - Map updating an element to PUT or PATCH `/elements/{element identifier}`.
 - Map deleting an element to DELETE `/elements/{element identifier}`.
- The data needed to create or update a resource is the resource representation that goes in the body.
- The resource identifiers participating in identifying the operation’s resource go in path parameters.

- The resource modifiers (such as search filters or resource identifiers that don't identify the operation's resources) go in query parameters unless they fit a standard HTTP header.
- Only standard HTTP headers defined in the IANA registry should be used (at this stage).
- The success of an operation is represented by a 2xx class HTTP status code, an error caused by the consumers by 4xx, and an error caused by the provider by 5xx.
- A successful creation returns 201 Created; other operations may return 200 OK if data is returned (search, read, update) or 204 No Content otherwise (delete).
- Operations with input data (query, body) must handle invalid input with 400 Bad Request.
- Operations using resource path with path parameter(s) must handle a not found resource error with 404 Not Found.
- All operations must handle unexpected server errors with 500 Internal Server Error.
- Output data goes in the response body unless it fits in a standard header defined in the IANA registry.
- A “do” non-CRUD operation can be mapped to POST /do (action resource), /doings (nominalization of action), or /results (resource based on result). The latter two are preferred.
- An API adhering to the REST architecture style is efficient, scalable, reliable, simple, portable, and modifiable. It must respect the client/server separation, statelessness, cache, uniform interface, and optional code-on-demand constraints.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 4.1

Following is a list of calls to a library API and the context in which they occur. Analyze each call to fix the resource path, path parameters, and query parameters if needed.

- 1 Read a book's review: GET /books/12345/reviews?reviewId=678
- 2 List a borrower's books: GET /books/borrowers/7890
- 3 Search for available books written in English: GET /books/available/en
- 4 Get book information: GET /book/12345
- 5 Search for science fiction authors: GET /genres/science-fiction/author

Exercise 4.2

You're designing an event ticket booking API. For each of the following operations and their inputs, indicate the HTTP method and the location of each piece of input data.

- 1 Search for events. Inputs: event type, date range.
- 2 Book a ticket. Inputs: event, number of seats, attendee information.
- 3 Modify a booking. Input: booking, new number of seats, attendee information.
- 4 Cancel a booking. Input: booking.
- 5 Fetch a user's bookings. Input: user, event type, date range.

Exercise 4.3

Following are scenarios describing API responses for a restaurant reservation system. Each is paired with an HTTP status code. Based on the HTTP statuses mentioned in this chapter, fix the HTTP status codes if needed or indicate alternatives.

- 1 404: A client searches for available tables, but no tables match the date, time, or party size.
- 2 200: A new reservation is successfully created in the system.
- 3 500: A client tries to update reservation details using an invalid reservation ID.
- 4 204: A client successfully cancels an existing reservation.
- 5 455: A client submits a reservation request with an impossible date or negative party size.

Exercise 4.4

In an API managing magazine subscriptions, creating a new subscription operation instantly returns the created subscription data in the response body with a 200 OK status. How would you modify this based on what you learned from this chapter?

Exercise 4.5

You need a "translate text to language" operation in a translation API. It takes input text and the target language for translation. Which HTTP method and resource path can you use to represent it?

5

Modeling data

This chapter covers

- Designing resource data models
- Designing operations data from resource models
- Spotting missing capabilities with data
- Completing business errors
- Ensuring a versatile API that meets consumer needs

After giving an HTTP representation to operations and locating coarse-grained input and output data in HTTP requests and responses, such as “Product information” in the response body of “Read a product,” we can model the data. This implies deciding that a product is an object with a required product reference (integer), category (string), price (float), and optional keywords (array of strings). Additionally, the “Product reference” path parameter will be an integer, and we will break down the “Filters” of “Search for products” into “category” and “keywords.”

Data modeling involves selecting data, names, types, and organizations in objects or arrays, which can be error-prone. It’s easy to end with incomplete and inconsistent input and output models. It can also be laborious, especially when wasting time in arguments such as `available` versus `isAvailable` at the wrong

moment. Our goal is to efficiently model versatile data that meets consumer needs. Although this is a good first draft, we will refine our design later to avoid managing too many concerns at once; user-friendliness, security, performance, and implementation constraints are crucial in data modeling and will be addressed in later chapters.

This chapter starts by clarifying which data we model and provides an overview of how we'll model it, inspired by the JSON portable data format. Then, using the Online Shopping example from previous chapters, we demonstrate how to model resources and derive them into operations inputs and outputs, building new recipes along the way. We also show how to use the models to ensure capability completeness and exhaustive business error listings and make sure our API is versatile and meets consumer needs.

5.1 *An overview of data modeling*

As shown in the zoomed API lifecycle (section 1.6) in figure 5.1, we continue designing the programming interface, as discussed in section 3.1.4. After observing the API Capabilities Canvas operations from the REST angle (section 3.2), we have represented the identified resources, actions, inputs, and outputs; figure 5.2 shows the identified resources and their paths, and figure 5.3 lists the operations and their HTTP representations. We now enter the third step, which consists of modeling previously identified input and output data. We continue to focus on the “versatile API that

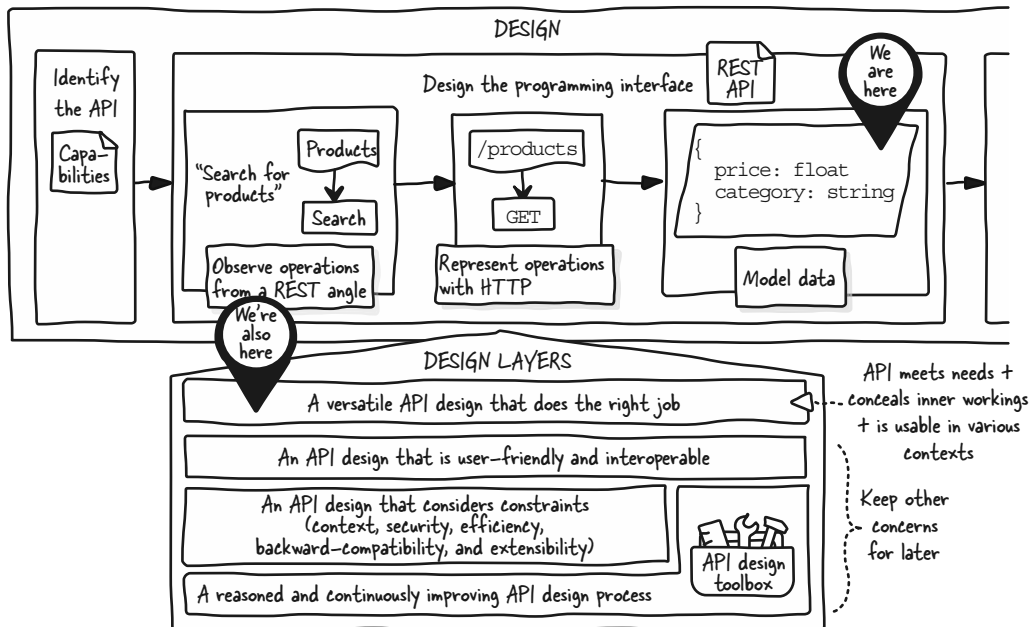


Figure 5.1 Once we have represented operations with HTTP, including choosing data locations in HTTP requests and responses, we can work on data modeling.

does the right job” layer, addressing consumer needs, concealing inner workings, and ensuring usability in various contexts (section 1.7.1). This section clarifies which data we model, introduces the JSON data format we must consider when designing data, and outlines how to model data.

RESOURCE	RELATION	PATH
Catalog	Contains many products	/products
Product	Belongs to the catalog	/products/{product reference}

Figure 5.2 We'll model the data of the resources of the Online Shopping example.

OPERATION	RES.	ACTION	HTTP METHOD	INPUT		OUTPUT			
				Desc.	Location	Description	Type	Status*	Data Location
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Product added to the catalog	Success	201	Product info. body Product URL header
						Wrong product information	Error	400	Error info. body
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Products info. body
						No products matching filters	Success	200	Empty products info. body
						Wrong filters	Error	400	Error info. body
Get product details	Product	Get	GET	Prod. ref.	path	Product found	Success	200	Product info. body
						No product found	Error	404	Error info. body
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	Product modified	Success	200	Product info. body
				Mod. prod. info.	body	No product found	Error	404	Error info. body
						Wrong product information	Error	400	Error info. body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204	
						No product found	Error	404	Error info. body
All operations						Unexpected server error	Error	500	Error info. body

Status* 200 OK, 201 Created, 204 No Content, 400 Bad Request, 404 Not Found, 500 Internal Server Error

Figure 5.3 We'll model the data of the input and output of the Online Shopping example's operations.

5.1.1 Which data are we modeling?

We'll model the data for which we identified locations in HTTP requests (section 4.4.1) and responses (section 4.6.1). This data is what we put in the path parameters, query parameters, header fields, and bodies shown in figure 5.4; figure 5.5 shows examples.

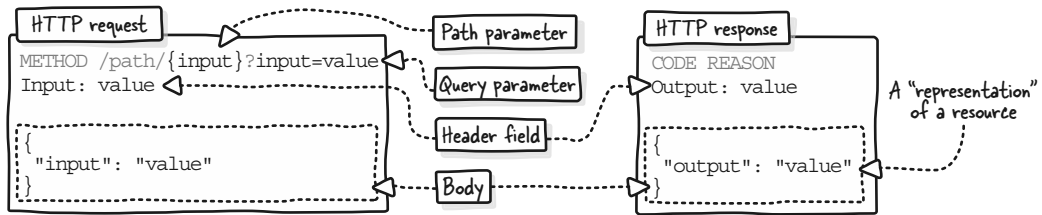


Figure 5.4 We model all HTTP request and response data (path parameters, query parameters, headers, and bodies).

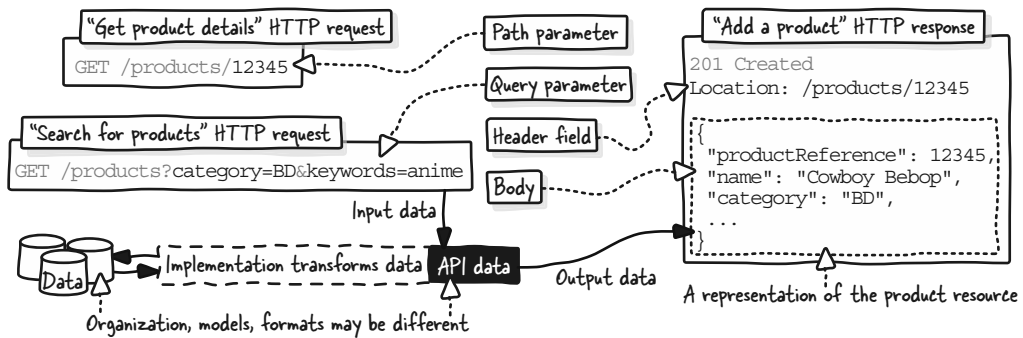


Figure 5.5 The data we model is the data exchanged between the consumer and the provider; it may differ from the underlying data stored in the database.

Path parameters appear in HTTP request paths and are resource identifiers pinpointing a unique resource; an example is the 12345 product reference necessary to get a product's details. Query parameters are resource modifiers that may appear at the end of the path after a ? and separated by & in the name=value form. The category and keywords product search filters are examples.

HTTP header fields contain metadata about requests and responses. For instance, the Location header indicates the created product URL when a product is added to the catalog.

HTTP request and response bodies contain a representation of a resource's desired or current state. For example, when we add a product to the catalog, the request body represents the product to create, and the response body represents the created product.



CAUTION Careful data modeling is crucial for public, partner, and private APIs. Poor design can have severe consequences, as noted in section 1.2. When modeling data for a web API, focus on the data exchanged from a subject matter perspective rather than on storage. Names, types, and organizations may differ between an API and the database; implementation handles data transformation. It's acceptable for API data to resemble database data, but we must meet consumer needs without carelessly exposing the provider's data structure (see section 2.8).

5.1.2 Introducing the JSON portable data format

HTTP supports any textual or binary data format in request and response bodies, such as HTML and images. The most common format for web APIs is JSON (JavaScript Object Notation, www.json.org), which is valued for its human readability and ease of processing in browsers that support JavaScript (typically compared to XML). Although JSON is based on JavaScript, it is programming language-independent and can be used in Java or Python, for example. It is also popular for data storage and configuration files.



NOTE It's a common oversight to limit REST APIs to JSON. Thanks to HTTP, REST APIs can use any format, such as XML, CSV, or images, if that suits our needs. The same REST API operation can even accept and return various data formats. For more about this, including XML and CSV samples, see section 9.7.1.

As shown in figure 5.6, JSON describes atomic values (strings, numbers, Booleans), ordered arrays, and unordered objects. Arrays are delimited by brackets (`[]`) and separated by commas (`,`). Objects use curly braces (`{}`) with properties separated by commas. A property key is a quoted string (`"price"`) and is separated from its value by a colon (`:`). Values can be strings (`"Cowboy Bebop"`), numbers (`49.99`), Booleans (`false`), objects, arrays, or null to indicate that a value is not set.

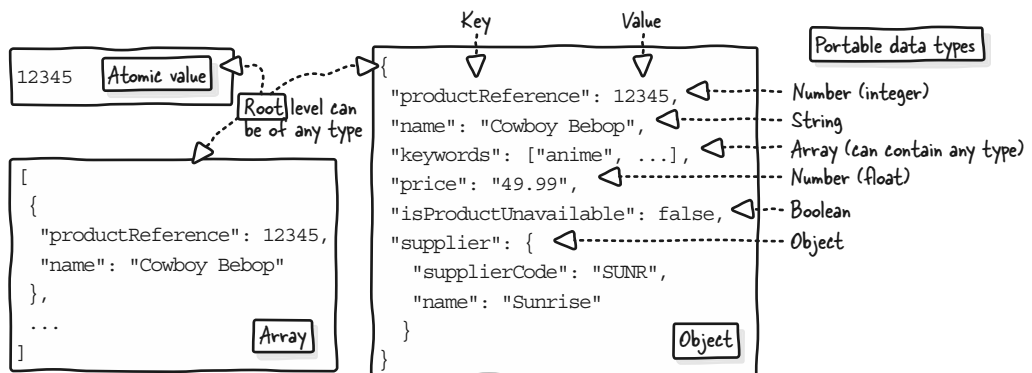


Figure 5.6 To ensure that all consumers can understand our data, we use the JSON portable data format. It handles Boolean, number, string, array, and object data types.



NOTE JSON is a possible format for representing resource data in bodies; we may use others. But we model data with JSON in mind to ensure maximum compatibility between providers and consumers. This includes path, query, and header parameters, which we'll typically map to atomic values.

5.1.3 Modeling data

As shown in figure 5.7, we go through three steps when modeling data. First we design theoretical resource models that encompass all potential business concept data. We define names, data types, and data structures. The “Product” resource is an object with properties like `productReference` (an integer) and `supplier` (an object).

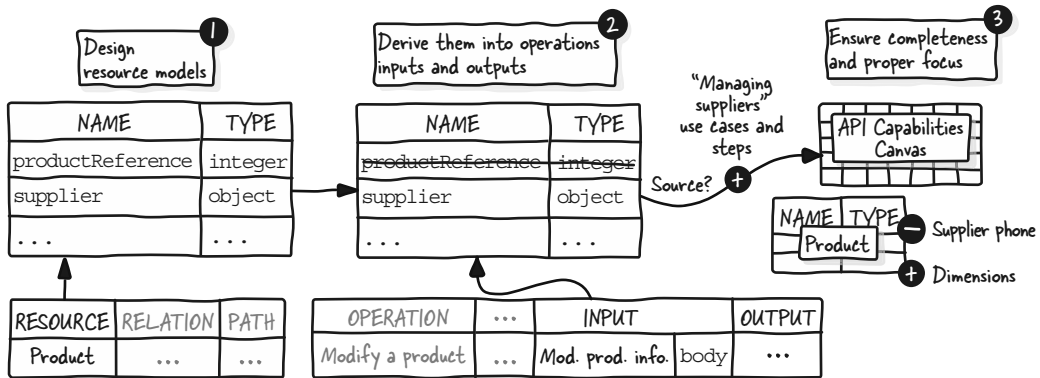


Figure 5.7 We use theoretical resource models to design inputs and outputs. We ensure that data meets consumer needs without exposing inner workings; we especially check input sources for gaps in capabilities.

Then we derive the theoretical model of a resource into inputs and outputs for the resource operations, selecting all or a subset of the model elements. When adding a product, the request body must not contain the product reference generated by the server.



NOTE Using resource models as a base for inputs and outputs simplifies our work and ensures data consistency, which is essential for creating user-friendly APIs (section 8.9). This also applies to other types of APIs. However, action resources may require a different approach (section 5.4.3).

Finally, we analyze data models to ensure that our API design meets consumer needs without exposing inner workings. For example, users need product dimensions, but the resource lacks them. The supplier’s phone number may be in our database, but it doesn’t need to be included in the “Product” resource. If adding a product requires a supplier code that consumers can’t provide, we must add supplier-related elements in our API Capabilities Canvas to fill the gap.



NOTE We temporarily store findings in our API spreadsheet to separate learning concerns; we'll find a better format for describing the programming interface data in section 7.1. Our focus is on the “versatile API that does the job” layer; however, data modeling will also address user-friendliness and interoperability (section 8.2), performance (section 13.1), security (section 12.1), and implementation constraints (section 14.1).

5.2 Designing theoretical resource data models

We're designing a REST API that relies on interacting with resources that we consider business concepts (section 3.3.1). Excluding HTTP headers metadata, all operation input and output relates to resource or concept data. Request and response bodies represent the resource, such as a product to create or its summary during searches. Path parameters, often IDs like a product reference, are part of the resource, whereas query parameters, such as search filters, originate from the resource data. Thus, our first task in data modeling is creating theoretical resource data models encompassing all potential data for the business concepts before actual inputs or outputs for any operations. This approach streamlines modeling and minimizes errors and inconsistencies, ensuring that our API meets consumer needs and is easy to use (section 8.9).

This section uses the Online Shopping example resources to discuss determining a resource's structure, choosing properties, their names and types, and whether they are essential. Finally, we explain how to achieve all this efficiently and are reminded of our current objective, which we may lose sight of when modeling data.

5.2.1 Determining a resource's structure

All elements we design must be any portable type introduced in section 5.1.2, including resource models. For simplicity at this stage of our learning, we will model a collection resource as an array and an individual resource as an object. As shown in figure 5.8, “Product” is an element of “Catalog,” so the `Product` model is of type object, and “Catalog” is a collection or list of “Product” and thus is an array of `Product`.

RESOURCE	RELATION	PATH	RESOURCE MODEL
Catalog	Contains many products	/products	array of Product
Product	Belongs to the catalog	/products/{product reference}	Product (object)

Figure 5.8 “Catalog” contains many products; it is an array. “Product” is an object.



NOTE We'll learn that collection or individual resource data must always be encapsulated in an object to enable us to add list metadata (section 9.6.7) or propose an extensible design (section 15.6.3).

5.2.2 Choosing an object resource’s properties

Designing a resource model requires identifying the properties representing the business concept, including a resource identifier. If our resource has relationships with other resources, we may include some of their data. Our expertise and input from subject matter experts (SMEs) guide us. We may also rely on existing or wire-framed UI, implementation code, or databases, which are often the only available documentation.



CAUTION As discussed in section 2.6, we must ensure that the resulting models meet consumer needs, make sense to consumers, and remain unbiased (especially when relying on the UI, code, or database). Section 5.5.3 will revisit these concerns in the data context.

Figure 5.9 shows the result of our discussions with SMEs about “What goes into a Product?” This section focuses on a few chosen elements; the following sections discuss the name, type, and required columns.

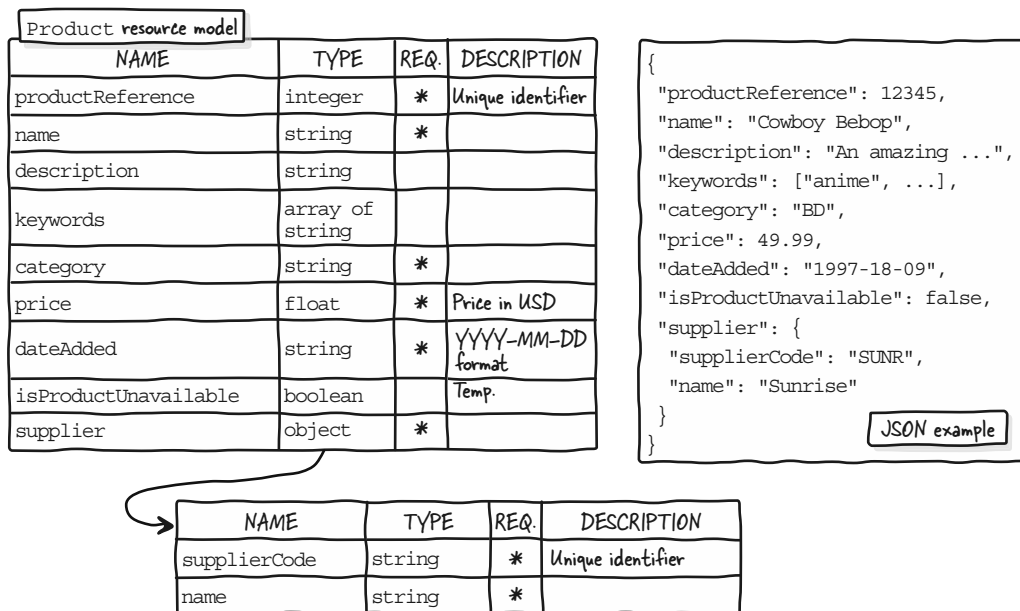


Figure 5.9 The “Product” resource theoretical model contains all the information necessary to describe a product in the context of the API.

The “Product” resource theoretical model contains the product reference resource identifier and data representing the product, such as a name and price. We included the code and name from the related supplier resource (seen in section 5.1.3); according to the SME, a product’s supplier name is essential for a product. The supplier

code can be used to read the related supplier resource for more information. Alternatively, we could include only the supplier code if the supplier name is nonessential.



NOTE Section 8.4.4 discusses the wise selection of resource identifiers. Section 8.7 covers determining how much data from related resources to include in a resource, which also affects efficiency, covered in section 13.1.

5.2.3 Choosing a property name and type

Once we have identified some data, we can name and type it. For now, we choose the first meaningful names that pop into the discussion without thinking much about them. We write them like variable names in code. For instance, the reference uniquely identifying a product is `productReference`, `product_reference`, or any other casing variation, depending on our preferences. Be consistent across the entire API (and even other APIs); don't have `product_reference` and `isProductAvailable`.

We select portable types appropriate for the data (see section 5.1.2). The `Product` model in figure 5.9 showcases all the portable data types. `productReference` is an integer, `name` is a string, `price` is a float, and `isProductAvailable` is a Boolean. Not all resource properties are atomic values; the `keywords` property is an array of string, and `supplier` is an object with its own properties.

We may provide an optional description to capture additional information that the combination of resource name, property name, type, and format can't convey. For instance, the product's `price` is expressed in US dollars. As there is no specific portable date type, `dateAdded` is a string whose description indicates a YYYY-MM-DD format.



NOTE See section 8.5 for more information about atomic data types and formats (including date formats). Section 8.8 covers the art of choosing names. Section 8.9 discusses the significance of consistency (in particular, we'll learn that resource identifiers should use specific naming patterns to be easily identifiable).

5.2.4 Indicating required properties

In a theoretical resource model, the `required` flag (the `Req.` column in figure 5.9) indicates properties essential for the concept. This is mainly a subject-matter question, but we can also consider what consumers must provide as input and what the API implementation always returns as output; this is how we'll interpret this flag when modeling input and output data. A product doesn't make sense without a `productReference`, `name`, or `price` but can exist without a `description` or `keywords`. We must also set this flag for deeper elements; for example, the `supplier` object must have a `supplierCode` property. It's not because the `supplier` property object is required that all of its properties are required; it could also have an optional `description`.



NOTE The `required` flag provides essential information for implementing and consuming the API. It indicates what the implementation always returns and what consumers must provide, which can affect user experience (see

section 9.4.4). If unsure of the value at that stage, you can define it when modeling inputs and outputs.

5.2.5 *Listing and modeling properties efficiently*

Listing and modeling resource data and data in general is not always easy. We need to identify the concept's data, organize it in objects or arrays, name it, type it, and decide whether it's essential (required). Trying to tackle all concerns simultaneously is the surest way to waste everyone's time and end up with a low-quality result. Typically, it's not the time to argue about `isAvailable` versus `isProductAvailable`. What matters is identifying that "Product" needs a property indicating its availability.

To streamline the design process, you can proceed as follows:

- 1 List the concept elements using the first names or descriptions that come to mind without worrying about details (final names or types).
- 2 Group those belonging to a subconcept to create subobjects.
- 3 For each element, choose a type, tweak the name into a variable name, and determine whether it's required.
- 4 If necessary, reiterate with deeper objects (such as the `supplier` object of a product).
- 5 Evaluate each element, and remove any that don't make sense.



NOTE Our current focus is on representing concepts and meeting consumer needs (the "versatile API that does the job" layer). This may result in suboptimal data that does not consider all usability, efficiency, or security aspects, which is normal, even for experienced designers. The following design layers will fix it; see section 5.1.3.

5.3 *Designing inputs and outputs data models*

After designing theoretical resource models, we can efficiently create inputs and outputs for each API operation by selecting the necessary elements from the theoretical models according to the context. This section provides practice in modeling inputs and successful outputs for typical create, read, search, update, and delete operations, which will help us discover recipes to streamline the process. Although this chapter focuses on successful outputs, we will also draft a temporary error model.



NOTE Section 5.4 summarizes and generalizes the insights gained from this section to streamline our work. Section 5.4.3 covers "do" operations modeling using these insights.

5.3.1 *Designing a read operation's inputs and success outputs*

Figure 5.10 shows that the "Get product details" (`GET /products/{product reference}`) operation has a "Product reference" input and a "Product information" success output. Because the operation reads the "Product" resource and returns "Product information,"

it can return a representation containing all data of the theoretical “Product” resource model (see figure 5.9).



TIP Model an operation’s output data before the input because, in most cases, each input data is identical to part of the output.

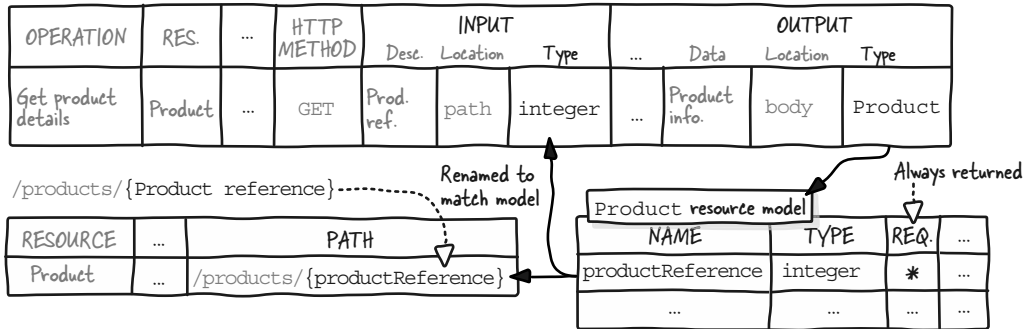


Figure 5.10 The operation output is the Product resource model; required properties are always returned. The productReference path parameter is based on the productReference property.

In the current context, each property’s required flag shows whether it is returned. For example, the name property is required, so it is returned for all products, whereas the nonrequired description property may not be returned.



NOTE Identifying always-present and sometimes-absent properties in API responses aids in implementation and testing. Knowing what data is returned and when is essential for consumers. See section 19.1 for information about how API design assists in the API lifecycle.

The “product reference” path parameter identifies the “Product” resource as the `productReference` from the Product model. Thus, we set the path parameter type to integer and rename it to `/products/{productReference}`. Although `{productReference}` is a placeholder replaced by actual values in API calls, it’s visible in the documentation; using the same name aids consumers in connecting data pieces easily (more on user-friendly data in section 8.2).

5.3.2 Designing a search operation’s inputs and success outputs

The “Search for products” operation (GET `/products`) reads the “Catalog” resource, an array of Product. It returns either “Products information” or “Empty products information.” Both can be an array of Product; one has Product elements, and the other is empty. However, figure 5.11 shows that the operation returns an array of `ProductSummary`, a subset of the Product model.

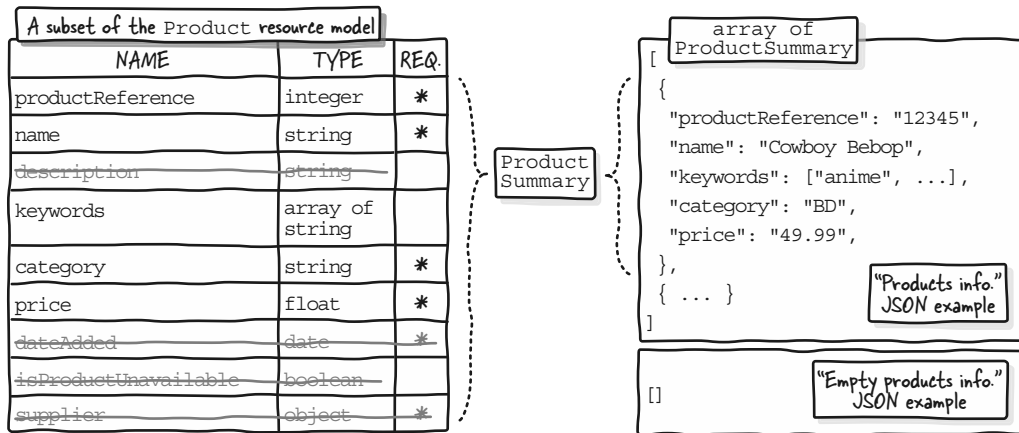


Figure 5.11 The ProductSummary model is a subset of the Product model containing essential information describing a product in the context of a list.

The ProductSummary model provides an overview of the product and fits our needs. It includes properties like productReference, name, category, keywords, and price but excludes others we consider less interesting in this context, like dateAdded. Consumers can use the returned product reference to get more data for a specific product with GET /products/{productReference}. Returning the complete “Product” resource data is also possible if it meets consumer needs.



CAUTION Putting insufficient or too much data in lists can affect API efficiency, leading to many API calls to read each element or high data volumes; see section 13.1.

The “Search for products” operation features a “Filters” input to retrieve a subset of all products. Filter options vary based on user needs and data available in API models. We can have keywords and category filter query parameters that align with the ProductSummary model properties; an example request is GET /products?keywords=anime,fantasy&category=BD. To filter by availability, we must add the isProduct-
Available property to the returned model. Filtering by the product’s origin country is impossible because it’s missing from the Product model. But we can add an origin property to Product and ProductSummary to allow filtering by origin.



NOTE Section 7.6.2 shows other options for array serialization in query parameters. Section 9.6 covers other filtering options, such as ranges; it also includes sorting and paginating lists. Having input and output consistent with each other is not an obligation. However, it’s crucial for usability; see section 9.4.2.

5.3.3 Designing a create operation's inputs and success outputs

The “Add a product to the catalog” operation (POST /products) has a single input, “Product information,” and returns “Product information” and “Product URL” when a product is successfully created. The “Product information” output is the created product’s data. We can use the `Product` theoretical model, as when reading a product. However, we may need to return less data for performance reasons (section 13.1); we can return a `ProductMinimal` model, an object containing the `productReference` property identifying the created product.

The “Product URL” goes in a standard HTTP header (section 4.6.2); its model doesn’t derive from the `Product` model. Following HTTP, we define a `Location` header with a string type, /products/12345, for example (/products/{productReference}).

As shown in figure 5.12, although they have the same name, the “Product information” input differs from the output. It is the essential data to create a product, a subset of the `Product` output model without implementation-managed properties. After discussions with SMEs and the implementation team, we conclude that the `Product-Creation` model includes all `Product` properties except `productReference`, `dateAdded`, and `supplier name`. The implementation generates the first two and determines the third based on `supplierCode`. In this creation context, each property’s required flag shows whether it must be provided. For example, a name is mandatory for creating a product, but a description is optional.

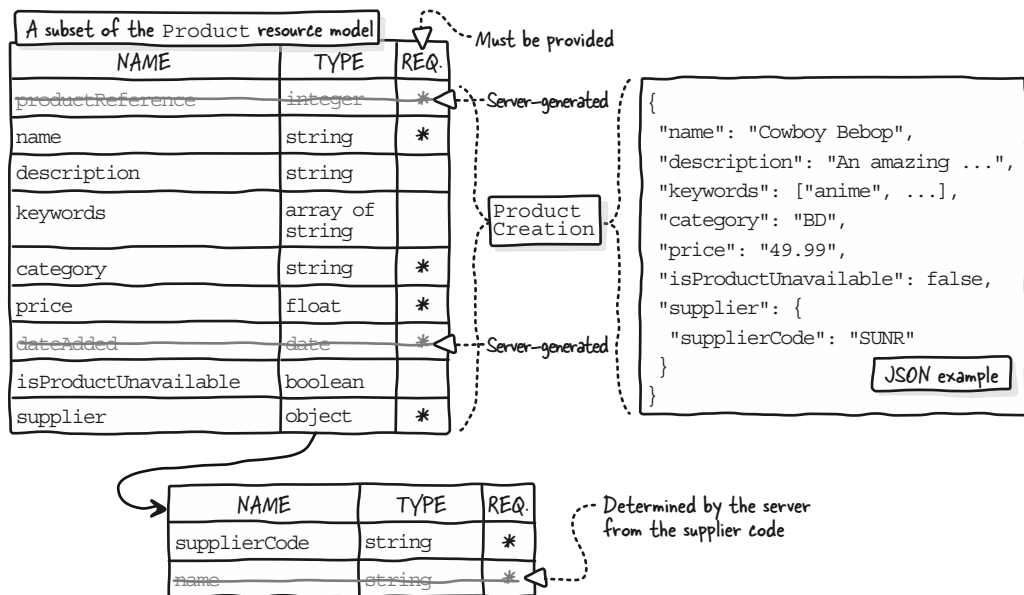


Figure 5.12 To model the data participating in product creation, we start with the complete model and remove the data managed by the server.



NOTE Knowing the required data for an operation is crucial for proper implementation. It also influences the user experience; more required data makes the operation more complex. Refer to section 9.4 for guidance on requiring minimal elements.

5.3.4 Designing an update operation's inputs and success outputs

The “Modify a product” operation (PUT or PATCH /products/{productReference}) has two inputs, “Product reference” and “Modified product information,” and returns “Product information” when the product is successfully updated. The returned “Product information” is the new state of the “Product” resource; thus, its data is the Product theoretical model, the same as when reading or adding a product. The “Product reference” path parameter is the same as when reading a product, as both operations share the same resource.

Figure 5.13 contrasts the “Modified product information” input in PUT and PATCH requests updating a product’s price and description. With PUT /products/{productReference}, the input contains all properties necessary to replace entirely; hence, we re-create the product identified by the path. So, we need a model identical to ProductCreation used when adding a product (POST /product). We can rename the creation model ProductCreationOrReplacement to have a unique model. If we use PATCH /products/{productReference}, which only needs the modified data to be sent, we can have a ProductModification model that is similar to ProductCreation, but all properties are optional.

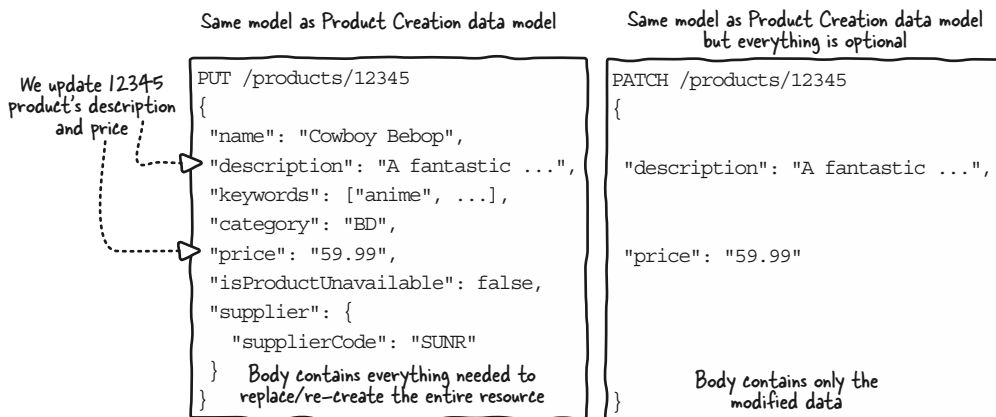


Figure 5.13 The model for PUT matches product creation. Modifying two properties requires sending all data. The model for PATCH is similar, but all properties are optional; consumers send only modified data.

Some data may be restricted from updates for subject-matter-related reasons. For example, a product’s supplier may be defined on creation and cannot be modified

afterward. To limit what can be updated, the input data model may be a subset of the one used for creation.



NOTE Section 9.8.1 will show that the implementation should accept modification and complete models as PUT input, ignoring extra properties to prevent unnecessary errors. The PATCH strategy employed is JSON Merge Patch (RFC 7396), the most common option, whereas JSON Patch (RFC 6902) is a less-used alternative; section 13.5.7 contrasts them.

5.3.5 Designing a delete operation's inputs and success outputs

The “Remove a product from the catalog” operation (`DELETE /products/{product reference}`) has a single input, “Product reference,” and no outputs. We’re in the same situation as with `GET`, `PUT`, and `PATCH /products/{product reference}`. This operation manipulates a “Product” resource, so we end with `DELETE /products/{productReference}`, where the `{productReference}` path parameter maps the `productReference` property of the theoretical “Product” resource.

5.3.6 Designing a temporary error data model

This chapter focuses on success output modeling, but we can design a temporary simple data model for all error outputs (4XX and 5XX) of all operations. The Error model in figure 5.14 is an object with a required `message` property, a string, conveying explicit, human-readable information about the problem. We’ll enhance it in section 9.8 when we discuss errors in depth.



Figure 5.14 To be exhaustive about output data modeling, we design a simplistic and generic error model that we will enhance later.

5.4 Streamlining input and output data modeling

Having designed all the resource models and operations’ inputs and success outputs, we can identify seven models to streamline the design for CRUD and “do” operations: complete, summarized, minimal, identifier, creation, replacement, and modification. Figure 5.15 illustrates their application. This section discusses designing and using these models and how to streamline property listing and modeling. It also highlights the risks related to similarly named elements.

	OPERATION	PATH	QUERY	REQUEST BODY	RESPONSE BODY
Create resource	POST /resources			Creation	Complete, Minimal
Search resources	GET /resources		Breakdown of response		List of Complete or Summarized
Read resource	GET /resources/{resourceId}	Identifier			Complete
Replace resource, Create resource	PUT /resources/{resourceId}			Replacement, Creation	
Partial res. update	PATCH /resources/{resourceId}			Modification	
Delete resource	DELETE /resources/{resourceId}				
Action resource	POST /do			Creation	Complete

Figure 5.15 The typical API operations use typical data models.

5.4.1 Designing and using the complete, summarized, minimal, and identifier models

Figure 5.16 shows we can derive the complete model to design the summarized, minimal, and identifier models. A *complete* (or *theoretical*) resource model, such as `Product`, should be designed first, as it is the source for the other models. Remember the guidance from section 5.2.5 to model data efficiently. It contains all possible business concept properties, including a resource identifier. We can use it as a successful output body for create, read, search (in a list), and update operations.

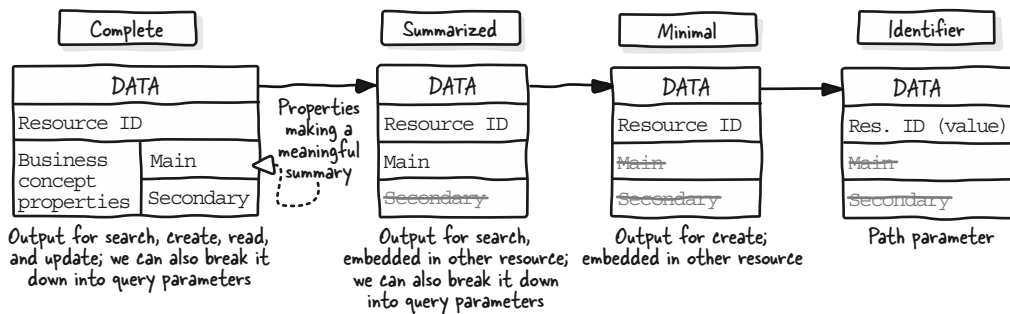


Figure 5.16 Starting from the complete data model, we can design summarized (essential information), minimal (only the resource identifier property), and identifier (only the resource identifier value) models.

A *summarized* model, such as `ProductSummary`, contains a subset of the data from the *complete* resource model, including resource identifiers and “main” properties representing a meaningful summary. We can use it as an output of search operations (in a list).

On search operations, we can use the properties of the *complete* or *summarized* model we use in the list as a base for query parameters, as in `GET /products?keywords=anime,fantasy&category=BD`. A *minimal* model, such as `ProductMinimal`, is a subset

of the *complete* or *summarized* models, containing only the resource identifier. We can use it as an output for a create operation (discussed in section 13.5.2).

The *summarized* and *minimal* models can also be embedded in other resources. For example, an *Order* may contain a list of *products* whose elements are one of these models.

An *identifier* model is the type of resource identifier of the *complete* data model. We can use it as a path parameter (`/products/{productReference}`).

5.4.2 Designing and using the creation, replacement, and modification models

Figure 5.17 shows how we can derive the complete model to design the creation, replacement, and modification models. A *creation* model, such as *ProductCreation*, contains all the properties needed to create a resource and goes in the body input of a “Create resource” operation. It is a subset of the *complete* model that excludes data managed by the implementation, such as the resource unique identifier (`productReference`) or creation date (`dateAdded`).

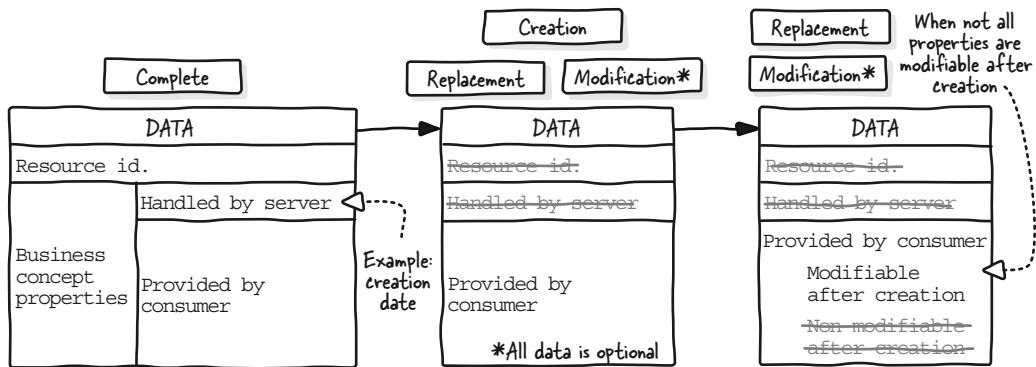


Figure 5.17 Starting from the complete model, we can design the creation, replacement, and modification of input data models that contain only consumer-managed data.

A *replacement* model, such as *ProductReplacement*, is the body input for an “Update resource” operation using `PUT`. It’s usually the same model as the *creation* model (*ProductCreationOrReplacement*).

A *modification* model, such as *ProductModification*, is the body input for an “Update resource” operation using `PATCH`. It’s usually a copy of the *creation* model, where all properties are nonrequired (section 13.5.7 discusses a less common alternative). If some properties are not modifiable after creation, replacement and modification models can be a subset of the properties consumers can provide on creation.

5.4.3 *Modeling data for “do” operations*

In section 4.7, we learned about representing “do” or non-CRUD operations using the REST model. We fundamentally defined two options: creating a business concept resource or an action resource. This section discusses how to model their data based on previous sections.

Choosing the business concept option and creating an “Executions” or “Results” resource means “do” is represented by a create operation (POST /executions or POST /results). Model the operation resources’ complete models, and derive them as input and output, as for a regular creation. Design the “Results” resource like any other business concept. The process for “Executions” is similar; the complete model includes all “do” input and output data (the numbers to sum and their total, for example) along with a resource identifier. The input is a creation model with only the necessary data for the action (the numbers to sum).

Using an action resource means having a POST /do that behaves like a function. The input body can contain all necessary data (numbers to sum), and the output provides the result (the sum). However, I suggest designing the output model like “Executions” resources without the identifier; returning all input and output data lets consumers understand the result’s source.

5.4.4 *Differentiating similarly named elements*

Similarly named resources, inputs, or outputs may be different concepts or require different data models depending on the context. Distinguishing them is essential for designing an API that meets consumer needs.

In section 5.3.3, we realized “Product information” was both an input and output for the “Add a product to the catalog” operation, but with different modeling. Thanks to the typical models in section 5.4, we can seamlessly differentiate elements between inputs and outputs and across operations that manipulate the same resource.

After identifying and modeling concepts, we must avoid using them blindly, particularly resources that can distract us, given the resource-driven nature of REST APIs. Similar terms can represent different concepts depending on context. For instance, the “Product” resource in catalog operations differs from the “Product” concept in a cart. Adding to a cart involves referencing the `productReference` and quantity, which differs from adding a product to the catalog. Therefore, we should differentiate these by naming them “Catalog Product” and “Cart Item” and modeling them appropriately. We can combine data when necessary: for example, listing cart items output may include `CatalogProductSummary` data in the `CartItemSummary`.



NOTE What seems like a unique business concept is often several concepts adapted to specific contexts. Always consult SMEs to determine whether seemingly identical concepts are truly the same; rename them with a suffix, prefix, or more precise term. As a last resort, fine-grained modeling can reveal nondifferentiation problems by showing unrelated data. When concepts

are related, use typical models from section 5.4 or their elements to compose new models.

5.5 Using data to ensure completeness and proper focus

Now that we have investigated consumer needs in depth using data modeling, we have a solid draft of our programming interface design. However, we must review the design for completeness and accuracy before moving forward. This review involves

- Spotting missing elements by analyzing input sources and output usages
- Ensuring complete business error-handling
- Focusing on the proper elements

This section examines these tasks, using the Online Shopping programming interface data as an example.

5.5.1 Spotting missing elements by analyzing input sources and output usages

We must check whether consumers can provide requested inputs and what they do with outputs to identify missing use cases, steps, and operations and ensure that our design meets all the user needs. This is similar to what we did with steps' inputs and outcomes in section 2.3.5.

The fine-grained input source check can reveal new API parts that were previously unknown during the needs analysis. For instance, to “Add a product to the catalog,” consumers must provide a `category`, which can also be used as a filter when searching for products. However, consumers can't invent this value. After discussing this with SMEs, we realize that a “Select a category” step needs to be added to the “Fill the catalog” and “Search for products to buy” use cases. Without this, it's impossible to add any products to the catalog.

We probably spotted all the use cases and steps during the needs analysis by investigating output usage. However, we can do a quick second pass. For example, we can consider what end users may do with the `supplier` information returned with products. We may add this as a filter to “Search products,” but if we decide the `supplier` property is irrelevant for end users, we can remove this information. However, catalog administrators need it. We may need to separate end users and admin operations; see section 12.3.4.

5.5.2 Ensuring complete business error-handling

Now that we have a detailed view of all inputs, we can double-check with SMEs about what possible business errors can occur, especially in creation and modification operations. At this stage of our learning, we focus on exhaustively identifying these errors, which is essential for correctly implementing the API. This information is also crucial for consumers who need to know the behavior of operations to code their applications.

In most cases, the newly identified errors should be refinements of those already detected during the needs analysis. For instance, “You can't add a product with a price

that is negative or above 100,000” refines “Wrong product information.” We can also add specific newly detected error cases to the description of the 400 error cases. We must also check whether errors affect the use cases identified during capabilities identification (adding new branches, operations, etc.), although that should be rare.



NOTE Section 9.8 thoroughly discusses handling errors, including data modeling and limiting their occurrence. We’ll add more errors as we cover new concerns, typically security (section 12.10) or planned unavailability (section 14.2.3).

5.5.3 *Focusing on the proper elements*

We must ensure that any piece of data aligns with consumer needs, is versatile, doesn’t expose the provider’s perspective, and isn’t too consumer-specific, as we did with capabilities in section 2.6. We must also ensure that consumers can send the data they want and get the data they need to achieve their goals in the context of the use cases identified during the needs analysis. For example, the “Product” resource data model returned by “Get product details” may miss data about the size of the product, which is crucial for consumers but not used elsewhere in the API.

In addition, we must be sure the data is versatile and fits in new contexts. Although the available wireframe of the first application using the API doesn’t show a `description` field, we add it to the `Product` model because it’s essential information for a product from a subject matter perspective. An application will likely need it sooner or later.

Next, we must ensure that the names, types, and data organization do not expose the provider’s perspective, which would expose the data organization or business logic (there’s less risk of software architecture problems here). For instance, if the price description says, “Add 10% on Fridays between 4 pm and 7 pm,” it would be better to find a way to avoid consumers having to deal with that. Section 8.4 will show us how to craft ready-to-use data.

Finally, we must be sure the data isn’t tainted by an overly specific consumer perspective by checking that it doesn’t mimic existing UI and isn’t specialized for one consumer. A typical UI influence would be to group the `description` and `keywords` properties under a `summary` object in the product model because that’s how information is presented on the existing website. But from a pure data perspective, agnostic of the context, this organization doesn’t make any sense.

Summary

- To improve compatibility between providers and consumers, all data should be modeled using JSON portable data types (strings, numbers, Booleans, arrays, and objects).
- Design data models for resources (business concepts), and derive them into inputs and outputs for each API operation.

- To design models, list properties without worrying about details (final names or types), reorganize and filter them, and finally, choose the name, type, and required status.
- Typical data models used as input or output for CRUD and “do” operations include complete, summarized, minimal, identifier, creation, replacement, and modification.
- The complete (or theoretical) model contains all business concept properties, including a resource identifier. Design it first; it is the source for the other models. Use it for create, read, search (list), or update operations.
- The summarized model is a subset of the complete model, including the resource identifier and properties representing a meaningful summary. Use it as output for search operations (list).
- The minimal model is a subset of the complete model containing only the resource identifier. Use it as output for create operations.
- The identifier model is the type of the resource identifier of the complete data model. Use it for path parameters.
- The creation model is a subset of the complete model that excludes data managed by the implementation. Use it as input for create operations.
- The replacement model is usually the same as the creation model. Use it as input for update operations using `PUT`.
- The modification model is usually a copy of the creation model where all properties are nonrequired. Use it as input for update operation using `PATCH`.
- Investigate data sources and usages to spot missing use cases or steps.
- Identify all business errors by using input data.
- Ensure that data is versatile, aligned with consumer needs, and free of unwanted providers or consumer influence that is too specific.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You’ll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 5.1

Analyze the response data for the “Read movie information” (listing 5.1) and “Search for movies” (listing 5.2) operations of an API for a movie streaming service, and fix them based on the lessons learned in this chapter.

Listing 5.1 Sample response for the “Read movie information” operation

```
{
  "id": "ZFqoFq",
  "title": "Ghost In The Shell",
```

```
"releaseYear": "1995",
"duration": "83",
"director": "Mamoru Oshii",
"music": "Kenji Kawai",
"language": "ja"
}
```

Listing 5.2 Sample response for the "Search for movies" operation

```
[
  {
    "title": "Ghost In The Shell",
    "language": "ja",
    "stars": 5,
    "music": "Kenji Kawai"
  }
]
```

Exercise 5.2

An API for a recipe-sharing platform allows users to search for recipes, read recipes, and save new recipes. Listing 5.3 shows an example of a request body used to save a new recipe (POST /recipes). Explain what's wrong with the requested data.

Listing 5.3 Sample request body for saving a recipe

```
{
  "id": "67890",
  "title": "Quiche Lorraine",
  "description": "A traditional French savory tart.",
  "creationDate": "2024-11-14T15:00:00Z",
  "ingredients": [
    { id: "shortcrust_pastry", quantity: "200g" },
    { id: "egg", quantity: "3" },
    { id: "cream", quantity: "200ml" },
    { id: "diced_bacon", quantity: "150g" },
    { id: "grated_cheese", quantity: "50g" },
    { id: "salt" },
    { id: "pepper" }
  ],
  "instructions": [
    "Preheat oven to 180°C",
    "Mix ingredients",
    "Pour into pastry",
    "Bake for 35 minutes."
  ]
}
```

Exercise 5.3

A fitness-tracking smartwatch application allows users to view the duration and type of a workout. The GET /workouts/{workoutId} API operation it uses returns the data shown in listing 5.4. Indicate whether each field should be kept or removed and why.

Listing 5.4 Sample "read workout" response

```
{
  "id": "abcd1234",
  "type": "running",
  "duration": { "value": 45, unit: "minute" }
  "distance": { "value": 8, unit: "km" }
  "date": "2024-11-13T08:12:34Z",
  "lastDbSync": "2024-11-14T12:00:00Z"
}
```

Exercise 5.4

The “Get car details” operation of an API for a car rental service can be used in two use cases: a customer renting a car and a mechanic checking a car for maintenance. Based on the sample data from a GET /cars/{carId} call in listing 5.5, can you verify whether the operation suits both? If not, explain how to fix this API design.

Listing 5.5 Data returned when getting car details

```
{
  "carId": "12345",
  "make": "Volkswagen",
  "model": "Golf",
  "rentalPricePerDay": 50,
  "features": ["air conditioning", "GPS", "automatic transmission"],
  "maxPeople": 5,
  "maxLuggage": 3,
  "mileage": 80000,
  "yearOfManufacture": 2018,
  "currentCondition": "No issues reported",
  "engineType": "1.4L TSI Turbocharged",
  "fuelType": "Petrol",
  "transmission": "Manual",
  "chassisNumber": "WVWZZZ1JZ9W123456",
  "lastInspectionDate": "2024-01-15",
  "nextInspectionDue": "2024-07-15",
  "tireCondition": "80% tread remaining",
  "brakeCondition": "Good",
  "batteryStatus": "Fully charged"
}
```

Describing HTTP operations with OpenAPI

This chapter covers

- Introduction to the OpenAPI Specification
- Describing resource paths
- Describing HTTP operations
- Describing HTTP operations' inputs and outputs

Our API designer's job could be considered done. We have designed a versatile REST API exposing capabilities that meet the needs identified in the Define stage of the API lifecycle. But we described the programming interface using a spreadsheet; it could also have been a word processor document or a wiki page. These formats are not made for this task; authoring and maintaining such documents can be complex and error-prone. A more efficient way is to use an API specification, a standard format for describing APIs. The OpenAPI Specification is the most common for REST APIs, simplifying design, reducing errors, and facilitating discussions while enabling more than just describing APIs and benefiting the rest of the API lifecycle.

After introducing OpenAPI, this chapter shows how the “Describe the programming interface” step of the API design process parallels “Design the programming interface.” Then we discuss how to author OpenAPI documents and provide an

overview of the steps to create an OpenAPI document describing the HTTP operations as we design them. We also go through these steps to describe the API's resources and operations and their inputs and outputs. The following chapter describes data models as we design them.

6.1 Overview of describing the programming interface

Figure 6.1 shows that we are entering the third step of the API design process outlined in section 1.6, “Describe the programming interface,” which parallels “Design the programming interface” (section 3.1). Describing HTTP operations and data models in a spreadsheet was temporary; we can use an API description format like OpenAPI when we start HTTP discussions. This format is an essential part of our API design toolbox; it helps streamline our work and discussions, helping us across all layers of API design. However, this chapter and the following focus on the basics so we can design a versatile API that does the job; we'll thoroughly discuss other layers in parts 2, 3, and 4 of this book (section 1.7).

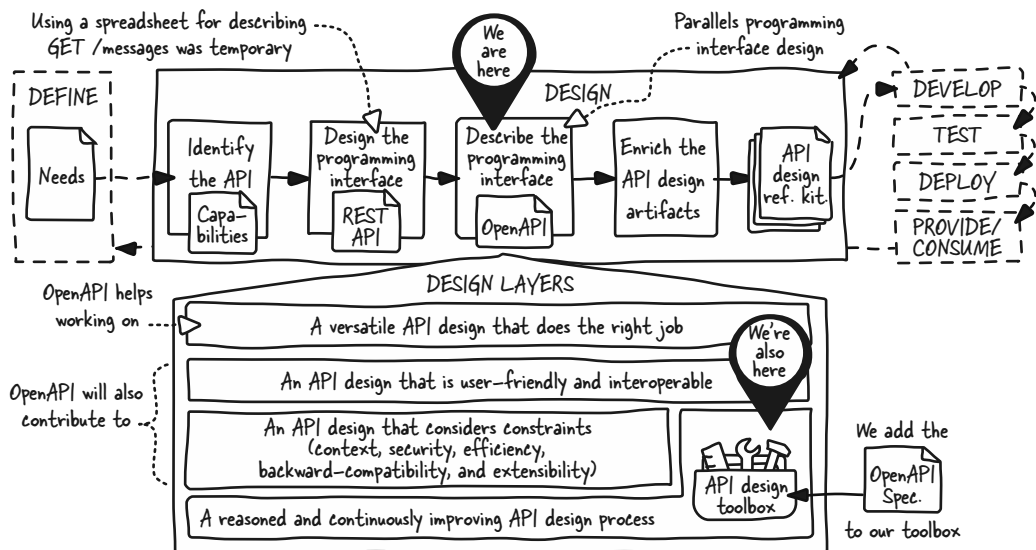


Figure 6.1 Using a spreadsheet to describe our REST API was only to simplify our learning. We can describe the programming interface with OpenAPI once we start to design it.

This section explores the OpenAPI Specification format we add to our API design toolbox and its usage. We introduce the YAML data format we'll use for writing OpenAPI documents and contrast an OpenAPI document with information from our API spreadsheet. Finally, we outline the steps to create the OpenAPI document for our API design and relate them to the tasks performed when designing the programming interface.

6.1.1 Introducing the OpenAPI Specification

The OpenAPI Specification (www.openapis.org) is an open source format for describing REST(ish) APIs. An OpenAPI document contains data about an API's resource paths, HTTP operations, inputs, outputs, and data models. Data models are defined using the JSON Schema format, discussed in section 7.1. OpenAPI can be used in many scenarios across the API lifecycle.

The OpenAPI Specification vs. Swagger and the rest of the world

The OpenAPI Specification (formerly the Swagger Specification) originated from the Swagger open source tools for generating API documentation and SDKs (Software Development Kit). In 2015, the Swagger 2.0 specification was donated to the OpenAPI Initiative under the Linux Foundation, evolving into OpenAPI with versions 3.0 in 2017 and 3.1 in 2021. The brand “Swagger” is owned by SmartBear; most non-SmartBear open source tools have been renamed from “Swagger-something” to “OpenAPI-something” or are likely outdated if not.

Alternative REST(ish) API specifications like WADL, RAML, and API Blueprint exist, but OpenAPI is now the industry standard, with most tools supporting it. Whatever type of API you work on, there is likely an API specification to use: for example, WSDL for SOAP, GraphQL Schema for GraphQL, Protocol Buffers for gRPC, and AsyncAPI for asynchronous APIs.

The OpenAPI Specification is predominantly used for API documentation; an example is shown in figure 6.2. Generating such documentation from our API spreadsheet is possible. But no commercial or open source API documentation solution can interpret our spreadsheet, whereas most REST API documentation tools understand the OpenAPI format.

However, OpenAPI has many uses throughout the API lifecycle beyond documentation. OpenAPI documents can be created via design tools, generated from code or network traffic, and used for testing, design, security checks, and application and infrastructure generation and configuration. OpenAPI is a standard that helps with all these tasks and serves as a bridge between tasks and tools. For example, an OpenAPI document created during design can be used to generate code during development, create tests, configure an API gateway during deployment, publish documentation when providing the API, and generate consumer code.



TIP You'll find plenty of OpenAPI-compatible tools on the official OpenAPI Tooling website (<https://tools.openapis.org>) or by searching the web for “*what you want to do OpenAPI*”.

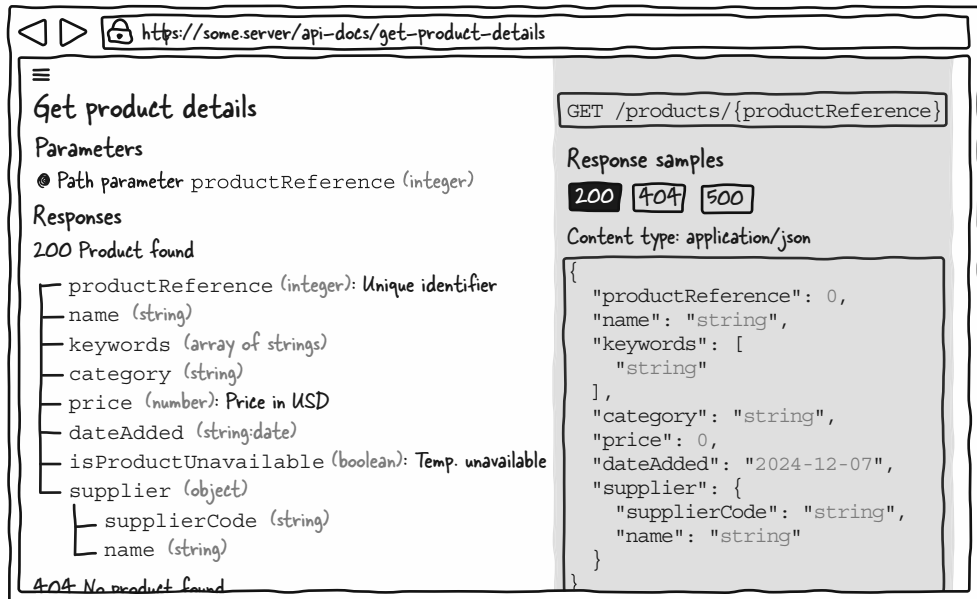


Figure 6.2 An API documentation tool can render an OpenAPI document and show information similar to our API spreadsheet, but in a more user-friendly way.

6.1.2 Using OpenAPI during design

During the design process, an OpenAPI document can be written, created via a design tool, or generated from an implementation (discussed in section 6.2). However, regardless of how it was created, an OpenAPI document simplifies describing the programming interface, helps our thinking, and facilitates discussions with stakeholders.

An OpenAPI document speeds up describing the programming interface and reduces oversights and errors. It provides a structure that can guide us when describing the programming interface. And it offers mechanisms to limit the repetition of information, such as defining data models once and reusing them in different places. Programs called *linters* can analyze it to seek design errors (discussed in section 18.1).

API documentation tools render OpenAPI documents in a way that makes it easy for API designers, subject matter experts (SMEs), and developers to understand operations and data without needing to understand OpenAPI syntax. Creating API mocks (or simulators) from OpenAPI documents is simple. Making calls to an as-yet-undeveloped API can be enlightening for an API designer. An API mock can help consumer teams draft proof-of-concept applications and catch design problems (discussed in section 18.10).

6.1.3 Introducing the YAML format

OpenAPI documents can be in either JSON or YAML format. All OpenAPI snippets in this book will be in YAML (we'll discuss why in section 6.2.5). If you need to become familiar with YAML, here's a brief introduction; for more information, visit <http://yaml.org>.

YAML (YAML Ain't Markup Language) is a human-friendly data serialization format and a cousin of JSON (introduced in section 5.1.2). As shown in figure 6.3, similarly to JSON, YAML can describe atomic values (strings, numbers, or Booleans), objects containing unordered key/value pairs, and arrays or lists containing ordered values. But the two formats represent them slightly differently:

- JSON encloses property names in double quotes (" "); YAML usually omits them (unless they are numbers).
- JSON encloses strings in double quotes (" "); YAML usually omits them (unless they only contain numbers or contain YAML-reserved or special characters).
- In JSON, commas (,) separate elements; YAML uses newlines.
- JSON's object's curly braces ({ }) and commas (,) are replaced by newlines and indentation in YAML.
- JSON's array brackets ([]) and commas (,) are replaced by newlines and dashes (-) in YAML.
- Unlike JSON, YAML allows comments beginning with a hash mark (#).

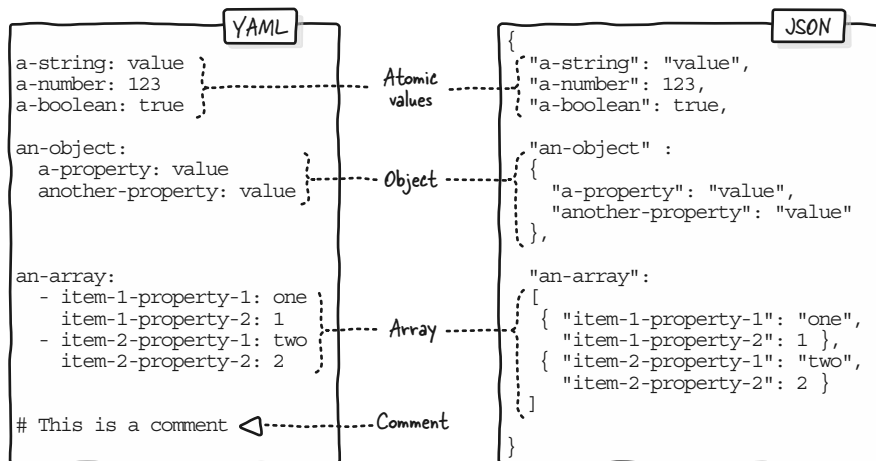


Figure 6.3 YAML is a cousin of JSON with additional features, such as commenting.

Converting one format to another is easy, but comments will be lost when converting YAML to JSON.

6.1.4 Contrasting an OpenAPI document with our API spreadsheet

We collected information on resources, operations, inputs, outputs, and data models in our API spreadsheet. The same information can be described in an OpenAPI document, as shown in figure 6.4. This section connects the dots between the two formats; we'll discuss the details in this chapter and the next one.

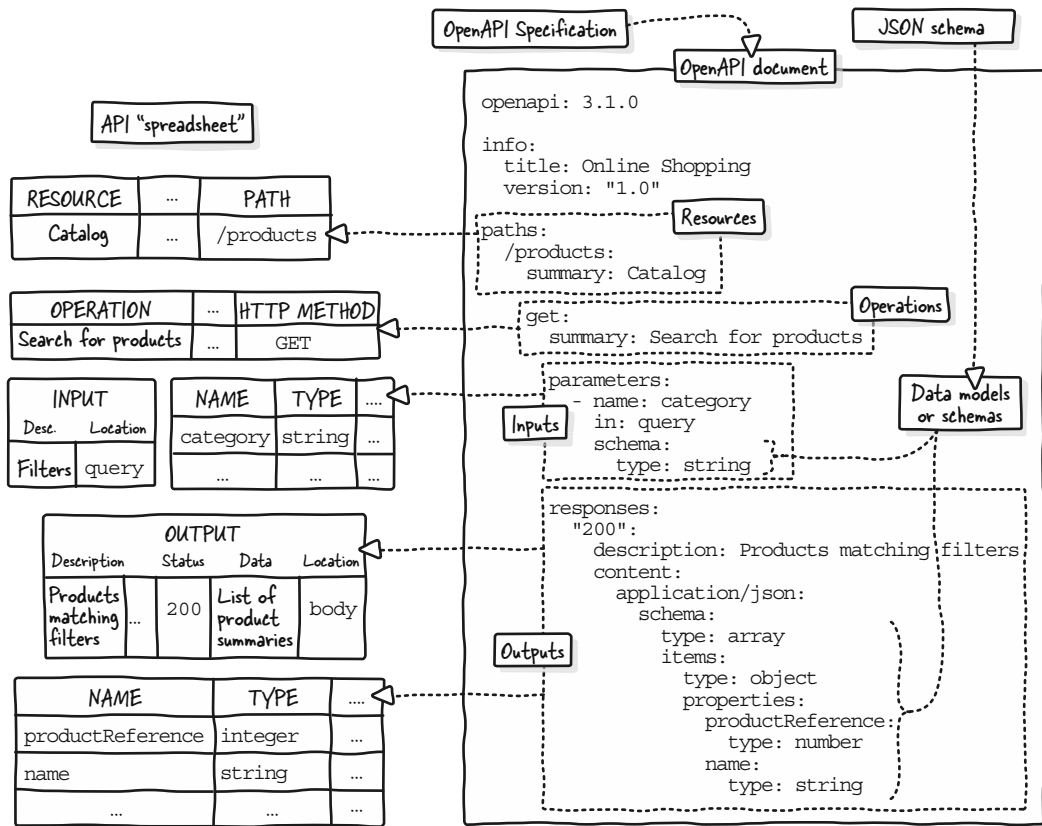


Figure 6.4 All of the REST API information we put in our custom API spreadsheet can be described in a standard way with an OpenAPI document.

The Resource table of our spreadsheet describes resources, such as “Catalog.” The OpenAPI document holds this information within the `/products` path under the `paths` key. The Operation table of the spreadsheet has information about the “Search of products” operation (“Catalog” resource), which uses the `GET` HTTP method. The OpenAPI document describes this operation with the `get` property inside `/products`. The document describes the inputs and outputs stored in our spreadsheet’s operation table under `parameters` and `responses`. Finally, the fine-grained data models we put

in the spreadsheet are described in the `schema` properties in OpenAPI. These `schema` properties use the JSON Schema format; see section 7.1.

6.1.5 Describing the programming interface while designing it

As shown in figure 6.5, we designed the programming interface in three steps: observing operations from the REST angle (section 3.2), representing operations with HTTP (section 4.1), and modeling the operations' input and output data (section 5.1). We can start using the OpenAPI format when dealing with HTTP after identifying REST elements to turn into HTTP elements.

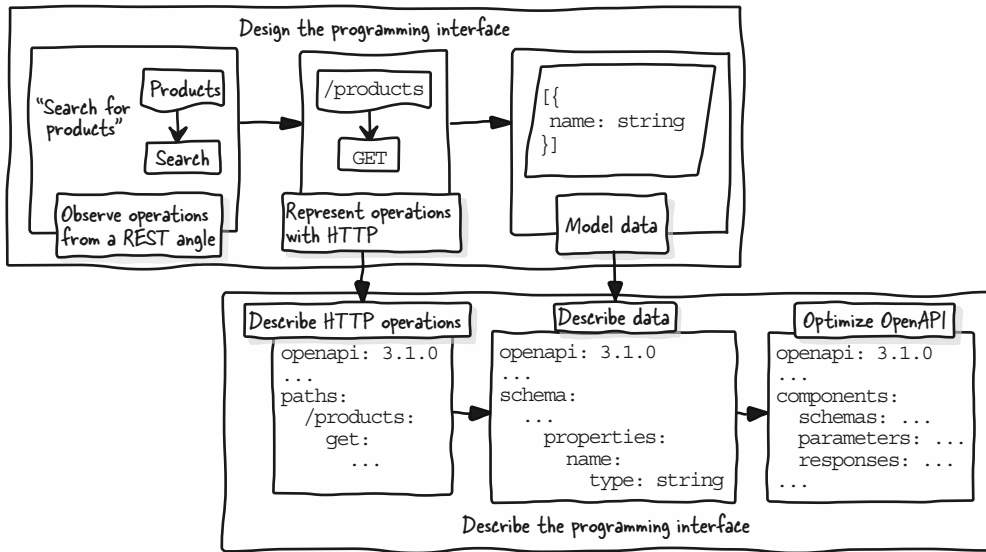


Figure 6.5 We design and describe the programming interface in parallel.



CAUTION We must not use OpenAPI during the needs analysis before the “Design the programming interface” stage. At that point, we need to identify API capabilities, and focusing on REST and HTTP would lead to a poor API (section 2.1.4).

We can create the OpenAPI document in three steps. The first two steps parallel the “Design the programming interface” stage. We describe the HTTP operations as we design them (this chapter). Then we describe data as we model it (see section 7.1). At that point, we’ll have a solid formal API description, which may contain duplication and could be better. However, it’s essential to separate concerns, first focusing on the API design and then optimizing the OpenAPI code as a third step (section 17.1), typically defining schemas, parameters, and responses shared by different operations. With experience, you’ll be able to seamlessly use some of these advanced

OpenAPI features when describing HTTP operations and data, which will speed up the authoring.



NOTE This book aims to demonstrate what you need most to design APIs. For more on OpenAPI, see the documentation at <https://spec.openapis.org/> or visit my OpenAPI Map at <https://openapi-map.apihandyman.io/>.

6.2 Authoring OpenAPI documents

Before describing the API, we must first discuss how to author OpenAPI documents in the context of API design. This section discusses two approaches to authoring OpenAPI documents: independent of implementation or generated from it. We also cover how to choose an approach and briefly mention OpenAPI editors and OpenAPI's version and format options.

6.2.1 Introducing the specification-first and code-first approaches

When designing a web API, we can create an independent OpenAPI document or generate one from the implementation. This book focuses on the former, but it's good to know both approaches to select one. This section introduces them and clarifies related terms; the next contrasts them.

Creating an OpenAPI document independently from the implementation using an OpenAPI editor (see section 6.2.3) is known as a *specification-first* approach. It is often confused with the *design-first* approach we're learning in this book, which involves designing the API (analyzing the problem, considering user experience, and integrating constraints) before coding the implementation. Specification-first can be used in a design-first approach.

Most REST API development frameworks enable the generation of OpenAPI documents from the code directly or via a library. Using this possibility during design is known as the *code-first* approach. It requires only coding the controllers and data models, not the business logic. Code-first is often opposed to design-first but is compatible with it.

The term *code-first* may also describe directly coding the API based on the output of the “Define” stage (see section 1.6.1). I usually don't recommend it, as not designing an API can have dire consequences (see section 1.2). A quick proof of concept may make sense, but be aware that once an API is consumed, it's hard to change it (see section 15.1).



TIP If you need to retro-document an existing API, adding an OpenAPI library to the implementation is a quick and simple way to achieve it. You won't get 100% exhaustive documentation, but you'll see all operations and their input and successful output data.

6.2.2 Contrasting the specification-first and code-first approaches

When designing APIs, we can use specification-first and code-first approaches. However, there are pros and cons to consider before making a decision adapted to the context, as shown in figure 6.6.

TOPIC \ APPROACH		SPECIFICATION-FIRST	CODE-FIRST
INDEPENDENT VS. SYNCED	PROS	<ul style="list-style-type: none"> ● The OpenAPI document is an independent source of truth 	<ul style="list-style-type: none"> ● The implementation de facto exposes the expected API ● The implementation and OpenAPI document are always in sync
	CONS	<ul style="list-style-type: none"> ● The implementation may be out of sync with the OpenAPI document 	<ul style="list-style-type: none"> ● Unexpected API modifications may happen
EDITING EXPERIENCE	PROS	<ul style="list-style-type: none"> ● Easy and independent OpenAPI editing 	<ul style="list-style-type: none"> ● Easy in-implementation code editing (for developers)
	CONS		<ul style="list-style-type: none"> ● Complex in-implementation code editing (for non-developers)
DESIGN AND OPENAPI QUALITY	PROS	<ul style="list-style-type: none"> ● Lesser risk of provider perspective ● More accurate and complete OpenAPI document 	
	CONS		<ul style="list-style-type: none"> ● Higher risk of provider perspective ● Less accurate and complete OpenAPI document
DEVELOPMENT STACK	PROS	<ul style="list-style-type: none"> ● Avoid too-early development stack choice 	
	CONS		<ul style="list-style-type: none"> ● Maybe too-early stack choice

Figure 6.6 When deciding between specification-first and code-first approaches, it's essential to check the pros and cons of each approach in the context in which you work.

The specification-first approach makes the OpenAPI document an independent source of truth for checking whether the implementation exposes the expected API. If the implementation offers extra operations, an API gateway based on this document won't expose them. Conversely, the code-first approach keeps the OpenAPI document aligned with the implementation, which is convenient. Yet without monitoring, the implementation may change unnoticed after initial development, lacking an independent description of the expected API.

OpenAPI is easier to learn than a development framework and requires no setup. Editors may even hide it behind a GUI. Code-first mixes documentation with code; API designers with coding skills may find it convenient, whereas those without may find it complex. Generating OpenAPI from code may involve compiling and deploying the application to modify it, which may be a problem in production unless you're used to constantly pushing in production.

Frameworks can generate OpenAPI from code using framework-native annotations, which is convenient. But some frameworks may not support all OpenAPI features, leading to less detailed or optimized OpenAPI documents (for example, path descriptions or responses defined once and shared across operations). However, this may be fine in some cases. OpenAPI-specific annotations or a complex OpenAPI generation configuration may be required to fill the gap.

The code-first approach *de facto* requires working with implementation code, which risks exposing inner workings, but we've learned to avoid this (see section 2.8). It also requires choosing a development stack early. Although we often stick to a stack that's already used, it may not be the most adapted one to implement the API; we'll discuss factors affecting this choice in section 14.8.



NOTE I usually recommend the specification-first approach, as it provides an easily editable source of truth. However, consider the context when choosing; if mixed implementation and documentation is not a problem, OpenAPI code quality is not a priority, and the implementation cannot evolve silently, the code-first approach may be suitable. Either way, you can switch between approaches at any time.

In this book, we'll use the specification-first approach to discover the main features of OpenAPI that support our learning of API design and to equip you to make specification-first versus code-first and OpenAPI-tooling-related decisions.

6.2.3 Picking an OpenAPI editor

As we use the specification-first approach, we need an OpenAPI editor. Many open source and commercial editors have basic syntax validation and rendering features. For extensive daily use, I recommend using editors that offer a GUI and hide the OpenAPI code. Search engines and the official OpenAPI Tooling website (<https://tools.openapis.org/>) can help you find one.

In the context of this book, you should use an OpenAPI editor that displays the code and renders side by side. The open source Swagger Editor Next (<https://editor-next.swagger.io/>) supports OpenAPI up to 3.1 and requires no account creation or installation. You can also run it on your machine (<https://github.com/swagger-api/swagger-editor/tree/next>). Note that <https://editor.swagger.io/> hosts the previous version, which supports OpenAPI up to 3.0; it will likely be updated one day.

6.2.4 *Choosing an OpenAPI version*

Three versions of OpenAPI are available: 2.0, 3.0, and 3.1. This section briefly discusses which one to use (or avoid). You can find a detailed comparison of the three versions in my “OpenAPI does what Swagger don’t” (grammatical error intended) presentation on my blog at <https://apihandyman.io/openapi-does-what-swagger-dont/>.

OpenAPI 2.0 (Swagger 2.0) is still widely used due to its age. But I recommend using the more recent versions, OpenAPI 3.0 and 3.1; they have many improvements covering document structure, security, documentation, and API and data modeling. They allow using JSON Schema for any data model and provide new features like callbacks and webhooks. Version 2.0 support decreases over time as tools are created or updated.

Ideally, use OpenAPI 3.1, but not all tools may support it yet. Using version 3.0, supported by most tools, is OK unless you need 3.1-specific features. The gap between 3.0 and 3.1 is minimal. The 3.1 enhancements are the support of a more recent version of JSON Schema and webhooks (discussed in section 14.6).

In this book, we won’t consider version 2.0 at all. We’ll use version 3.1, and I’ll warn about potential 3.0 backward incompatibility when relevant.



NOTE Converting OpenAPI 2.0 to 3.0 or 3.1 is simple, as no information is lost during the conversion (check <https://tools.openapis.org> to find tools). However, migrating an existing ecosystem requires ensuring that all tools using OpenAPI documents are compatible with the chosen version, which should be the case unless they are custom-made.

6.2.5 *Choosing between JSON and YAML*

OpenAPI documents can be in YAML or JSON format. If you don’t have a specific requirement, it’s a matter of preference. We’ll use YAML in this book. I prefer it over JSON for writing and reading OpenAPI documents because fewer brackets and quotes make it more straightforward for humans. I also like YAML because it can be commented on (although comments are lost when converting to JSON). However, block YAML indentation problems can be bothersome.



TIP Indentation or spacing problems may not be obvious; if an editor says, “Property X is not authorized,” it likely indicates an indentation problem at or near the “X” level.

6.3 *Describing HTTP operations with OpenAPI*

The rest of this chapter focuses on describing HTTP operations with OpenAPI. As we’ve seen in section 6.1.5, it happens in parallel with representing operations with HTTP.

As shown in figure 6.7, we describe operations following the steps used while designing them (section 4.1). We add resource paths, HTTP methods, inputs,

HTTP status codes in responses, and outputs in each response. The final document gives an overview of the API capabilities in their HTTP representation, which is helpful for discussion.

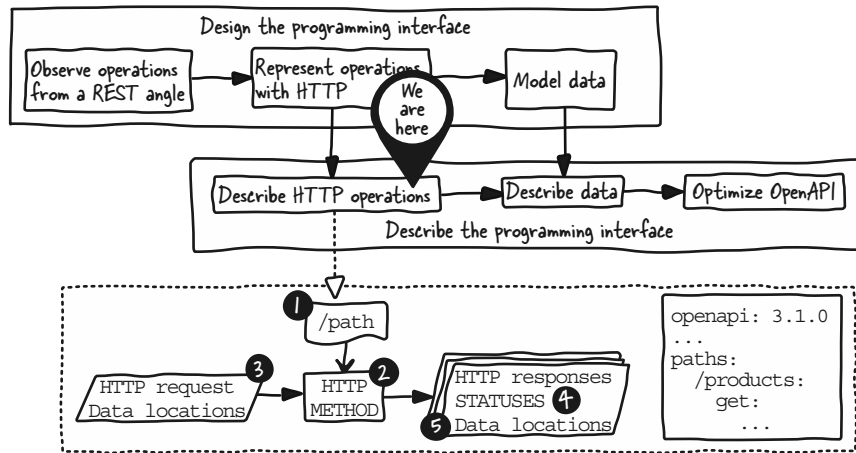


Figure 6.7 We describe HTTP operations with OpenAPI as we design them.

We continue using the “Online Shopping” resources and operations shown in figures 6.8 and 6.9. These typical REST API elements allow for a good overview of the OpenAPI Specification format possibilities for describing HTTP operations.

RESOURCE	RELATION	PATH
Catalog	Contains many products	/products
Product	Belongs to the catalog	/products/{productReference}

Figure 6.8 We can use OpenAPI to describe the HTTP representation of resources instead of the resource table of the API spreadsheet.

6.4 Describing resource paths

We can initiate our OpenAPI document when we design resource paths (see section 4.2.2). In this section, we create a minimal OpenAPI document and then learn how to describe basic resource paths and resource paths holding path parameters. We work with the “Catalog” and “Product” resources from figure 6.8.

OPERATION	RES.	ACTION	HTTP METHOD	INPUTS		OUTPUTS				
				Desc.	Location	Description	Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info.	body	Product added to the catalog	Success	201	Product info.	body
									Product URL	header
						Wrong product information	Error	400	Error info.	body
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Products info.	body
						No products matching filters	Success	200	Empty products info.	body
						Wrong filters	Error	400	Error info.	body
Get product details	Product	Get	GET	Prod. ref.	path	Product found	Success	200	Product info.	body
						No product found	Error	404	Error info.	body
Modify a product	Product	Modify	PUT or PATCH	Prod. ref.	path	Product modified	Success	200	Product info.	body
				Mod. prod. info.	body	No product found	Error	404	Error info.	body
						Wrong product information	Error	400	Error info.	body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204		
						No product found	Error	404	Error info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 204 No Content, 400 Bad Request, 404 Not Found, 500 Internal Server Error

Figure 6.9 We can use OpenAPI to describe the HTTP representation of operations instead of the operation table of the API spreadsheet.

6.4.1 Initiating an OpenAPI document

The following listing shows a minimal OpenAPI document with three entries: openapi, info, and paths.

Listing 6.1 A minimal OpenAPI document

```

openapi: 3.1.0          ← OpenAPI version used in the document

info:                  ← API's metadata
  title: Online Shopping ← API's name
  version: "1.0"        ← API's version

paths: {}              ← Empty paths needed to make the document valid

```

The `openapi` field indicates the version of OpenAPI used. Its value is `3.1.0` because we chose to use OpenAPI version 3.1 in section 6.2.4. If we had used version 3.0, it would be `3.0.3`. Note that parsers ignore the third segment of the version number, so `3.0.0` would work the same.

The `info` object contains general information about the API, such as its name (`title`) and `version` (surrounded by quotes to ensure that it's interpreted as a string). We'll discuss API names in section 11.3.2 and versioning in section 15.4. We'll learn more about the `info` object and other data to put in it in section 19.2.

The `paths` object will hold our Online Shopping API resource paths. This property is mandatory to make the document syntactically valid, so we add two curly braces, `{}`, representing an empty object, to avoid problems. We can now add the resource paths as we design them.

6.4.2 Describing a path

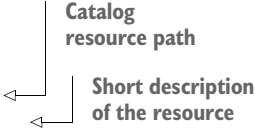
To describe a resource path in an OpenAPI document, add it as a key under `paths`, with its value as a *Path Item object* containing its name. Let's add the "Catalog" resource as `/products` (a list of products).

Listing 6.2 Resource paths

```
openapi: 3.1.0

info:
  title: Online Shopping
  version: "1.0"

paths:
  /products:
    summary: Catalog
```



To add the "Catalog" resource path, we remove `{}` (which indicated an empty `paths` object) and add `/products` as a key under `paths`. Then we add the name "Catalog" in natural language as a `summary` property.



TIP The `summary` property is optional but strongly recommended; it bridges the OpenAPI document and API Capabilities Canvas information and helps anyone, including your future self, understand the path.

6.4.3 Describing a path with path parameters

Adding a path with one or more path parameters is similar but requires describing the path parameters. For example, the "Product" resource path is `/products/{productReference}` (a product of the list of products); it contains the `productReference` path parameter. Adding this path to our document will trigger an error that requires defining `productReference`. The next listing shows the solution.

Listing 6.3 Path parameters

```

...
paths:
  ...
  /products/{productReference}:
    summary: Product
    parameters:
      - name: productReference
        in: path
        required: true
        schema: {}

```

Product resource path containing a path parameter
 Path-level parameters list
 Path parameter name (same as in path)
 Path parameter location
 Needed to make the parameter definition valid

We must declare the path parameter in the `parameters` list of the path. This property holds path-level parameters that apply to all operations under the path. We'll discuss the importance of this list when discussing OpenAPI document optimizations in section 17.3.1. The parameter object describing the `productReference` path parameter has four properties: `name`, `in`, `required`, and `schema`.

The `name` matches the one used in the path (minus the brackets `{ }`). The `in` property indicates its location in the HTTP request (`path`). We'll discover other locations in section 6.6.1.

The `required` property indicates whether a parameter is optional or not. Setting it to `true` for any path parameter is mandatory in OpenAPI. Making an API call without it wouldn't make sense.

We set the `schema` property value to an empty object (`{ }`) to avoid any parsing problems. We'll return to it when we describe data (section 7.1).

If the resource path has multiple path parameters, define them similarly by adding elements to the `parameters` list. Most GUI editors add path parameters when typing the path.



TIP It's not mandatory to use final code-like names for path parameters. You can use a path parameter name that has not yet been designed, such as `product reference`, and update it later to `productReference` when working on fine-grained data modeling.

6.5 Describing operations

We can add operations to resource paths when we choose the HTTP methods representing them (see section 4.3.2). Each HTTP method is a key in the path item object of the operation's resource, with its value being an operation object containing the operation's name.

As shown in figure 6.10, we identified five operations. We represented them with standard HTTP methods to manipulate resources: two for "Catalog" (`POST` and `GET` `/products`) and three for "Product" (`GET`, `PUT`/`PATCH`, and `DELETE` `/products/{productReference}`). Let's start with `POST /products`.

OPERATION	RESOURCE	ACTION	HTTP METHOD
Add a product to the catalog	Catalog	Add	POST
Search for products	Catalog	Search	GET
Get product details	Product	Get	GET
Modify a product	Product	Modify	PUT or PATCH
Remove a product from the catalog	Product	Remove	DELETE

Figure 6.10 We add operations to the OpenAPI document when choosing their HTTP method.

To add an operation to an OpenAPI document, add the HTTP method in lowercase as a key under the resource’s path. Then, just like at the path level, add the operation’s human-readable name as a summary property. Listing 6.4 shows the “Catalog” resource represented by the `/products` path with the `post` HTTP method added. The `post` operation’s summary is set to `Add a product to the catalog`.

Listing 6.4 HTTP methods

```
...
paths:
  /products:
    summary: Catalog
    post:
      summary: Add a product to the catalog
```

← Operation’s HTTP method
 ← Operation’s name



TIP The `summary` property is optional but strongly recommended; it bridges the OpenAPI document and API Capabilities Canvas information and helps anyone, including your future self, understand the path and HTTP operation couple.

Adding all the other operations we’ve identified is done similarly. You’ll note that we use the `put` HTTP method for the “Modify a product” operation; we could also have used the `patch` method.

Listing 6.5 Defining all operations

```
...
paths:
  /products:
    summary: Catalog
    post:
      summary: Add a product to the catalog
    get:
      summary: Search for products
  /products/{productReference}:
    summary: Product
    parameters: ...
    get:
      summary: Get product details
```

```
put:
  summary: Modify a product
delete:
  summary: Remove a product from the catalog
```

6.6 Describing operation inputs

We can add the operation inputs to the OpenAPI document when we choose their locations in the HTTP request (see section 4.4). As shown in figure 6.11, we have identified four possible locations for data in an HTTP request: headers, path parameters, query parameters, or body. From the OpenAPI standpoint, we can group the headers, path parameters, and query parameters as “non-body parameters.”

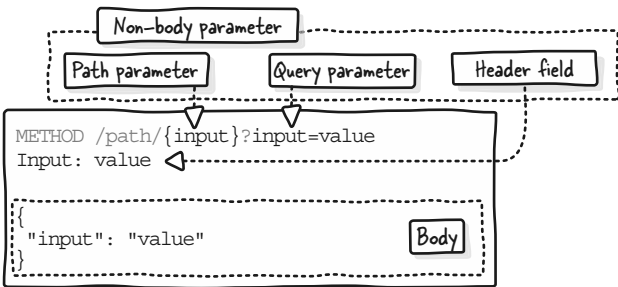


Figure 6.11 In OpenAPI, input data is organized into body and non-body data (headers, path, and query parameters).

We already described path parameters in section 6.4.3. Now we’ll learn how to describe the other non-body request parameters and then work on request bodies, focusing on the operations and information highlighted in figure 6.12.

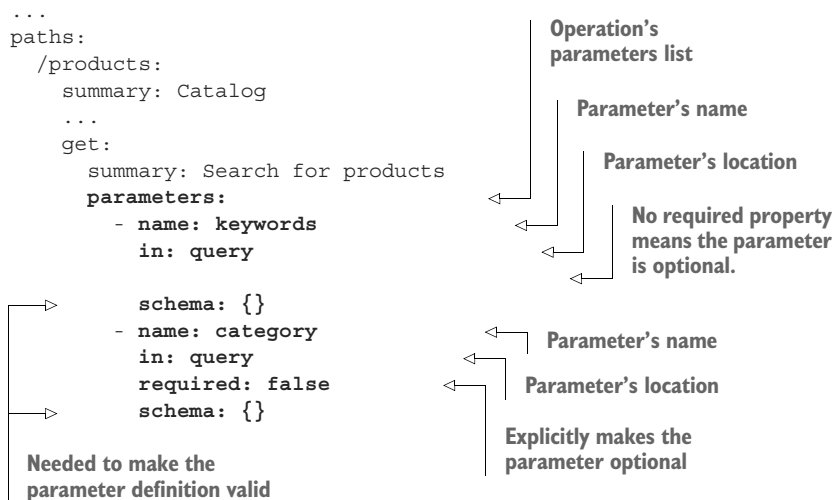
OPERATION	RES.	ACTION	HTTP METHOD	INPUTS	
				Description	Location
Add a product to the catalog	Catalog	Add	POST	Product info	body
Search for products	Catalog	Search	GET	Filters (name, category, keywords, q)	query

Figure 6.12 We add operation inputs to the OpenAPI document when choosing their locations in the HTTP request.

6.6.1 Describing query parameters and other non-body parameters

In OpenAPI, all non-body parameters go into the `parameters` list in the operation or path object. As shown in figure 6.12, during data modeling, we identified a fuzzy “filters” query parameter for the “search for products” operation, which we refined into multiple parameters, such as `category` and `keywords`. We could have worked with “filters,” but this section uses the two fine-grained parameters to demonstrate how to define multiple parameters in an operation.

Listing 6.6 Non-body parameters



This listing shows the GET `/products` operation with two query parameters added to the `parameters` list. It has the same structure as the `parameters` list seen when adding the path parameter of the `/products/{productReference}` path in section 6.4.3. Each has a name (`keywords` and `category`) and an `in` property indicating that they are in the query.

We'll address the optional versus required question in section 9.4.4. But to demonstrate OpenAPI, the document describes both parameters as optional in two ways:

- The keyword parameter doesn't have a `required` property.
- The category parameter has the `required` property explicitly set to `false`.

As we did for the path parameter, we added an empty `schema` object property to make the parameters syntactically valid. That's another place we'll describe data (section 7.1).



TIP You can temporarily describe multiple and not-yet-clearly identified parameters with a single parameter (such as `filters`) and later replace them with the final ones (like `name`, `keywords`, `categories`, and `q`) when working on data modeling.

The parameter object describes all input data in an HTTP request, except the request body, with `in` set to `query`, `path`, or `header`. We can define parameters at the path or operation level. Path-level parameters apply to all operations under the path (discussed further in section 17.3.1).

6.6.2 Describing request bodies

We saw in section 4.4.1 that request bodies are used on the HTTP methods `POST`, `PUT`, or `PATCH`. In OpenAPI, request bodies are defined in the `requestBody` property of the

operation. In our “Online Shopping” example, two operations are using an input parameter of type body: “Add a product to the catalog” (POST /products) and “Modify a product” (PUT /products/{productReference}). Let’s describe the first; you can proceed similarly for the second.

Listing 6.7 Request body

```
...
paths:
  /products:
    summary: Catalog
    post:
      summary: Add a product to the catalog
      requestBody:
        description: Product info
        content:
          application/json:
            schema: {}
```

This is the POST /products operation in which we added a `requestBody` property. Its value is a *Request Body object* containing the `description` and `content` properties.



TIP The `description` property is optional but strongly recommended; it bridges the OpenAPI document and API Capabilities Canvas information and helps anyone, including your future self, understand what goes in the request body.

Similarly to the `summary` and `name` properties previously used, the `description` property holds this input’s human-readable name from the operation table of the API Capabilities Canvas (see figure 6.12).

The `content` property is a *Content object* that describes the body’s content. It’s optional, but filling it with minimal data makes the request body visible in OpenAPI documentation tools. It also allows us to mark where we’ll need to define data, as we did in parameters.

We must provide the request body data format using a *media type*. Here, we’ll arbitrarily use `application/json`, which says the body is in JSON. We’ll discuss other options in section 9.7.1.

As we did for the non-body parameters, we added an empty `schema` object property. That’s yet another place where we’ll describe data (section 7.1).



NOTE Section 14.3 covers file uploads and downloads, and how to describe binary content with OpenAPI.

6.7 Describing operation output HTTP status codes

We can add information about the operation’s responses when we choose HTTP statuses to describe output types (see section 4.5). This section focuses on the “Add a

product” (POST /products) and “Search for products” (GET /products) operations shown in figure 6.13. You can proceed similarly for the other operations.

OPERATION	RES.	ACTION	HTTP METHOD	INPUTS		OUTPUTS				
				Desc.	Location	Description	Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Product added to the catalog	Success	201	Product info.	body
						Wrong product information	Error	400	Product URL	header
Search for products	Catalog	Search	GET	Filters	query	Products matching filters found	Success	200	Error info.	body
						No products matching filters	Success	200	Products info.	body
						Wrong filters	Error	400	Empty products info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 400 Bad Request, 500 Internal Server Error

Figure 6.13 We add responses to the OpenAPI document when choosing their HTTP statuses.

6.7.1 Describing an output case type with an HTTP status

In OpenAPI, the outputs of an operation are described in the responses property. As shown in figure 6.13, we use 201 Created to represent the success output for the “Add a product to the catalog” operation and 400 Bad Request for the “Wrong product information” error output. All operations must handle an unexpected server error with a 500 Internal Server Error. Let’s add them to our document.

Listing 6.8 HTTP status

```

...
paths:
  /products:
    summary: Catalog
    ...
    post:
      summary: Add a product to the catalog
      ...
      responses:
        "201":
          description: Product added to the catalog
        "400":
          description: Wrong product information
        "500":
          description: Unexpected server error

```

Subject-matter-oriented output description

Operation's possible responses

Output HTTP status code (created)

Output HTTP status code (bad request)

Output HTTP status code (server error)

We add the `responses` property to the `POST /product` operation object, with HTTP status codes as keys (surrounded by quotes, "201", to make them syntactically valid YAML keys). Each HTTP status value is a *Response object* with a `description` property containing a human-readable description of the response.



TIP The `description` property is optional but strongly recommended; it bridges the OpenAPI document and API Capabilities Canvas information and helps anyone, including your future self, understand the meaning of any HTTP status code.

6.7.2 Dealing with outputs sharing the same HTTP status code

Different output cases can share HTTP status codes in an operation. However, having duplicated keys in YAML or JSON is impossible. As shown in figure 6.13, the “Search for products” operation uses 200 OK for two different output cases. Recall from section 5.3.2 that in both cases, the operation returns a list of product summaries: one filled with the product found and the other empty. Let’s see how to handle this duplication.



CAUTION When outputs of an operation share the same HTTP status, they usually have similar data types. If not, there may be an operation or data type identification problem. You may use a common parent data type or split the operation to solve this problem.

As shown in listing 6.9, we merge the two outputs into a single response object under the "200" key in the operation’s `responses`. We join the two descriptions into the `description` property.

Listing 6.9 Merging descriptions

```
...
paths:
  /products:
    summary: Catalog
    ...
    get:
      summary: Search for products
      ...
      responses:
        "200":
          description: Products matching filters found or
            ➤ no products matching filters
        "400":
          description: Wrong filters
        "500":
          description: Unexpected server error
```

HTTP status code
can appear only
once

Merged output
descriptions

6.8 Describing operation output contents

We can describe the operation output data or content when identifying the data location (see section 4.6). This section shows how to describe response bodies, handle

responses without bodies, and work with response headers. We focus on two operations: `POST /products` and `DELETE /products/{productReference}`, using the information in figure 6.14. You can proceed similarly for the other operations.

OPERATION	RES.	ACTION	HTTP METHOD	INPUTS		OUTPUTS				
				Desc.	Location	Description	Type	Status*	Data	Location
Add a product to the catalog	Catalog	Add	POST	Prod. info	body	Product added to the catalog	Success	201	Product info.	body
									Product URL	header
						Wrong product information	Error	400	Error info.	body
Remove a product from the catalog	Product	Remove	DELETE	Prod. ref.	path	Product removed	Success	204		
						No product found	Error	404	Error info.	body
All operations						Unexpected server error	Error	500	Error info.	body

Status* 200 OK, 201 Created, 400 Bad Request, 500 Internal Server Error

Figure 6.14 We add output data to the OpenAPI document when choosing their locations in the HTTP response.

6.8.1 Describing response bodies

In OpenAPI, describing the response data is done similarly to describing the request body data (see section 6.6.2), but we handle the human-readable description differently.

Listing 6.10 Response bodies

```
...
paths:
  /products:
    summary: Catalog
    ...
    post:
      summary: Add a product to the catalog
      ...
      responses:
        "201":
          description: Product added to the catalog
          content:
            application/json:
              schema:
                description: Product info.
        "400":
          description: Wrong product information
          content:
            application/json:
              schema:
```

← Data to be returned on "201 Created"

← The response body's media type is JSON.

← Data description

← Data to be returned on "400 Bad Request"

← The response body's media type is JSON.

```

        description: Error info.
    "500":
        description: Unexpected server error
        content:
            application/json:
                schema:
                    description: Error info.

```

Diagram annotations:

- Arrow pointing to the `description: Unexpected server error` line: Data to be returned on "500 Internal Server Error"
- Arrow pointing to the `schema: description: Error info.` line: The response body's media type is JSON.

We add a content property on each response object of the `POST /products` operation. As we did for the request body in section 6.6.2, we take for granted that we have JSON data, so we add the `application/json` media type. We'll discuss other options in section 9.7.1.

Unlike in the request body, the schema has a description property set with the value from the API Capabilities Canvas (see figure 6.14). That keeps the human-readable data description separate from the response description, as done in the API Capabilities Canvas. We'll discuss describing data in the `schema` property in section 7.1.



NOTE Section 14.3 covers file uploads and downloads and how to describe binary content with OpenAPI.

6.8.2 Dealing with responses without bodies

Some output may contain no response bodies. That is the case for the `204 No Content` response of `DELETE /products/{productReference}`. It doesn't need a content property, as seen in the following listing. However, the `404 Not Found` response does.

Listing 6.11 Response without a body

```

...
paths:
  /products/{productReference}:
    summary: Product
    ...
    delete:
      summary: Remove a product from the catalog
      responses:
        "204":
          description: Product removed
        "404":
          description: No product found
          content:
            application/json:
              schema:
                description: Error info.

```

Diagram annotations:

- Arrow pointing to the `description: Product removed` line: No data is returned on "204 No Content".
- Arrow pointing to the `description: Error info.` line: Data is returned on "404 Not Found".

6.8.3 Describing response headers

In OpenAPI, the response headers are defined in the `headers` map of the "Response" object. The only operation using a response header is "Add a product to the catalog." When a product is successfully added, it returns the newly created product's URL in a

Location standard HTTP header (see section 4.5.3). The next listing demonstrates how to describe this header.

Listing 6.12 Response header

```
...
paths:
  /products:
    summary: Catalog
    ...
    post:
      summary: Add a product to the catalog
      ...
      responses:
        "201":
          description: Product added to the catalog
          headers:
            Location:
              description: Product URL
              required: true
              schema: {}
          content: ...
```

Response header map

Header name

Header description

The header is always returned.

Needed to make the parameter definition valid

We add a headers map to the 201 Response object in the responses of POST `/products`. The header's name, `Location`, becomes a key in this map. Its value is a *Header object*, which is a *Parameter object* minus the name and in properties. It contains the description we put in the operations table of the API Capabilities Canvas. We set the `required` flag to `true` for parameters to indicate that they will always be returned. As we do for all data, we add an empty schema object (see section 7.1).

Summary

- An API specification is a data format describing APIs; it can be used across the API lifecycle. The OpenAPI Specification, formerly the Swagger Specification, is the industry standard for REST APIs.
- During design, an OpenAPI document simplifies describing the programming interface, helps our thinking, and facilitates stakeholder discussions.
- The specification-first approach is usually recommended for authoring OpenAPI documents, but the code-first approach may be used depending on context.
- An OpenAPI document can be filled in parallel with HTTP operation design and data modeling. It can be initiated when designing the resource paths.
- Resource paths (`/products`, for instance) are keys in the `paths` object.
- A path parameter must be defined in the path-level `parameters` list of its path. Its `required` flag must be set to `true`.
- The HTTP method representing an operation goes under its resource path as a key in lowercase (`post`, for instance).

- An operation's header or query parameter goes into the operation's `parameters` list, and the request body is described in `requestBody`.
- The `in` property of a parameter indicates its location in the request (`path`, `query`, `header`).
- A mandatory parameter has its `required` property set to `true`, an optional one doesn't have this property, or it is set to `false`.
- Filling the `content` property allows a `requestBody` to be visible in API documentation tools.
- The `schema` properties mark the locations where fine-grained data will be described.
- An operation's output HTTP status code is a key under the `responses` property of the operation.
- Merge descriptions of outputs sharing the same status code only if they return the same data type or have a common parent. Otherwise, consider splitting the operation.
- A response's header is defined in `headers`, and its body goes into `content`.
- Use `summary` or `description` to keep a human-readable description of elements, connecting to the API Capabilities Canvas, so anyone (including your later self) can understand the meaning of all HTTP elements.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix. I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 6.1

Fix the book resource path definition in listing 6.13.

Listing 6.13 A resource path definition to fix

```
paths:
  /books/{bookReference}:
    summary: A book
    parameters:
      - name: bookId
        in: path
        schema: {}
```

Exercise 6.2

You are designing an API for an online teaching system. Describe the resource path representing a specific course offered by a specific instructor.

Exercise 6.3

You are designing an API for a hiking application. Describe the inputs of the operation that retrieves the list of segments in a specific trail and allows users to filter segments by difficulty.

Exercise 6.4

Fix the description of the “Get hotel details” operation shown in listing 6.14.

Listing 6.14 Operation with errors

```
paths:
  /hotels/{hotelId}:
    summary: An hotel
    get:
      summary: Get hotel details
      responses:
        "200":
          description: Hotel details successfully retrieved
          content:
            schema: {}
```

Exercise 6.5

You are designing an API for a travel agency system. Describe the inputs and outputs of an operation that adds a new destination to a specific travel package and returns the created destination.



Describing data with JSON Schema in OpenAPI

This chapter covers

- Using JSON in OpenAPI
- Describing resource data
- Describing request parameter data
- Describing request and response body data
- Describing response header data

Once we have described the HTTP operations with OpenAPI, we can move on to the next step and describe their data. Describing data models in a spreadsheet, word processor document, or wiki page is technically possible. However, although we used a spreadsheet for learning purposes, we must not use such formats for the same reasons they are not suitable for describing HTTP operations: they are not made for this task, authoring and maintaining such documents can be complex and error-prone, and their use is limited to reading them. Instead, we continue using OpenAPI, which uses another standard called JSON Schema to describe data.

This chapter introduces the JSON Schema format and provides an overview of how to describe data while designing it. We briefly discuss JSON Schema authoring in the context of OpenAPI. Then we explain how to describe resource data models with JSON Schema in an OpenAPI document and use them as inputs and outputs for operations.

7.1 An overview of describing data

Figure 7.1 shows that we’re still in the “Describe the programming interface” step introduced in section 6.1, which parallels “Design the programming interface” (section 3.1). We continue replacing our spreadsheet for describing the programming interface. After describing HTTP operations with OpenAPI (section 6.3), we describe data models with the JSON Schema format, which OpenAPI uses under the hood.

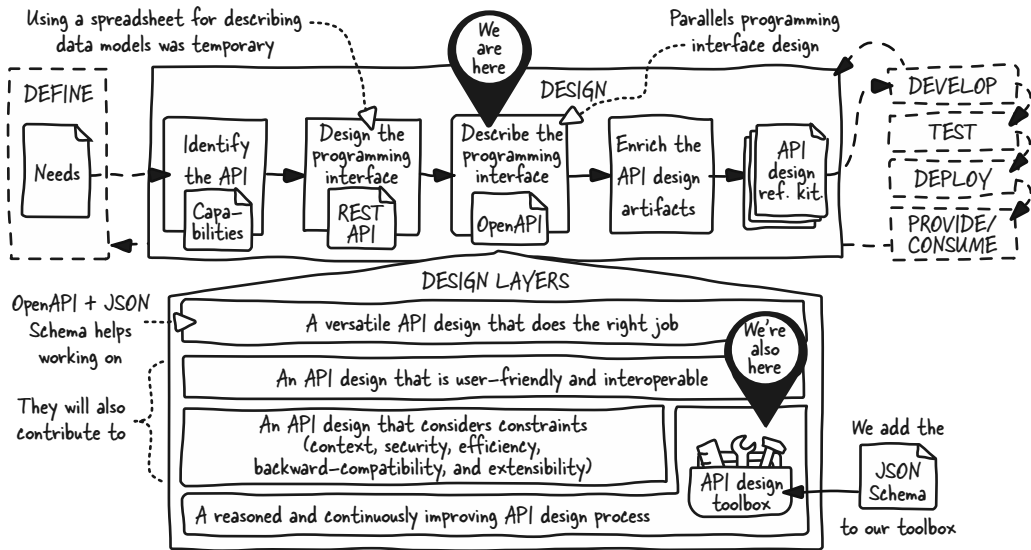


Figure 7.1 Using a spreadsheet for the data models of our API was only to simplify our learning. We can describe data models with JSON Schema in our OpenAPI document once we start designing them.

This section introduces JSON Schema and its use in OpenAPI, contrasts an OpenAPI document with the API spreadsheet to highlight JSON Schema usage, and discusses when and how to describe data models when designing an API.



NOTE Remember that using spreadsheets for data modeling was temporary; we separated concerns to facilitate learning. After reading this chapter, you can use OpenAPI and JSON Schema directly for modeling.

7.1.1 Introducing JSON Schema

When introducing the OpenAPI Specification format in section 6.1.1, we briefly stated that OpenAPI uses another format to describe data: JSON Schema. It is a format independent of OpenAPI, aiming to validate and annotate JSON (or YAML) data.

A JSON Schema document (called a *schema* or *JSON schema*) can validate that JSON data is an object with a required amount property (a number greater than 0) and an optional currency property (a string with value USD or EUR). The schema can also

contain additional information that is not used for validation. For example, the schema can describe the currency property as a “standard ISO 4217 currency code” (such standard codes are discussed in section 8.4.5).

An OpenAPI document uses JSON schemas to describe and document data. But JSON Schema contributes to all areas where OpenAPI is used: design, documentation, mocking, and testing, for example (see section 6.1.1). It can also be used independently of OpenAPI to validate application JSON/YAML configuration files or generate user interfaces with validation checks, for example.



NOTE This book focuses on the essentials of JSON Schema in the context of OpenAPI and API design. Visit <https://json-schema.org/> for resources, learning materials, examples, and reference documentation.

7.1.2 Contrasting OpenAPI and JSON Schema with our API spreadsheet

As a reminder of section 6.1.4, which overviews the “Describe the programming interface” stage of API design, figure 7.2 contrasts the information we gathered in the API

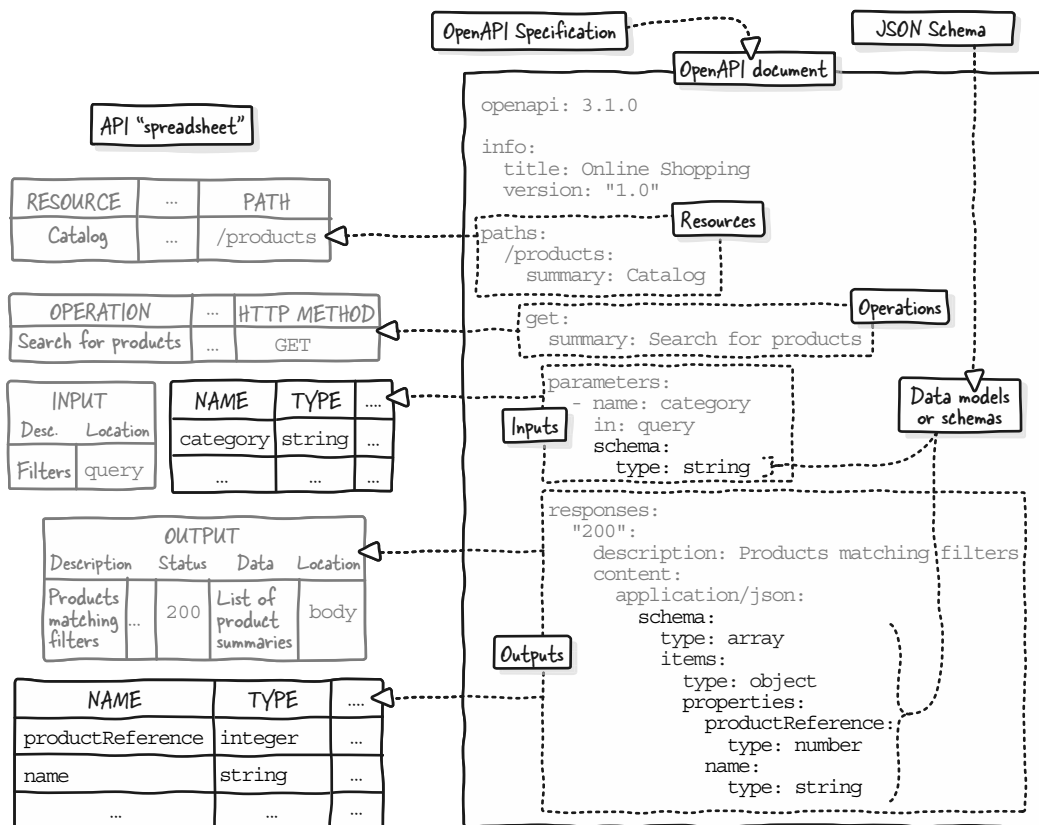


Figure 7.2 No need for the API spreadsheet when modeling data; we can use JSON Schema in our OpenAPI document instead.

spreadsheet with an OpenAPI document. It connects the dots between the two formats and highlights the element this chapter covers.

In the previous chapter, we used the resource and operation tables of the API spreadsheet to put their content under the `paths` property of the OpenAPI document (section 6.3). We added `schema` properties set to `{ }` to mark data model locations. The `schema` properties are now filled with JSON schemas containing the same information as the table we used when modeling data in “Design the programming interface.”

7.1.3 Describing data while designing it

As shown in figure 7.3, we can describe API data in our OpenAPI document when we start to model it (section 5.1). We follow the same steps: we describe complete resource models and derive them into other typical models, or pick bits of them to describe the operation’s input and output data.

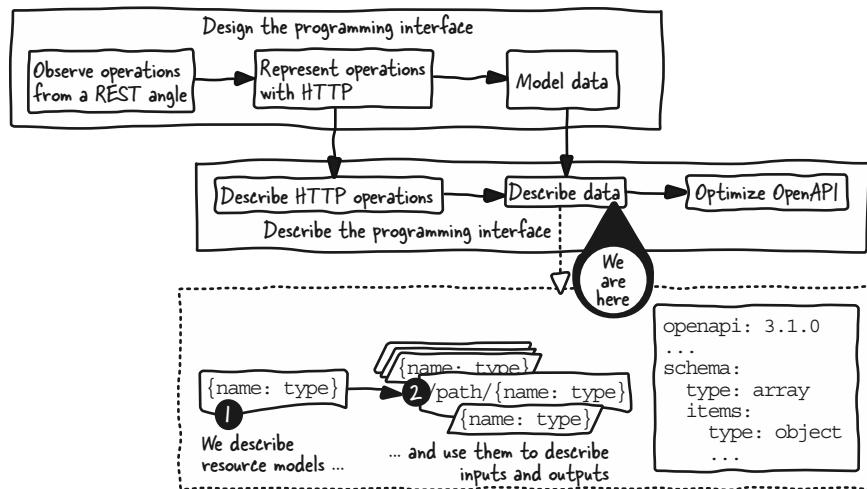


Figure 7.3 We describe data using OpenAPI and JSON Schema while modeling it. We proceed with the same steps: modeling the resources and deriving them.

In this chapter, we continue working on the “Online Shopping” OpenAPI example document initiated in the previous chapter. We use the models we previously designed for this example (section 5.1) to learn how to describe data with JSON Schema in OpenAPI. We will not describe all the API data, but you’ll learn all you need to fill the gaps.



NOTE To avoid being polluted by pure OpenAPI and JSON Schema concerns, we’ll only use essential OpenAPI and JSON Schema features to achieve our task. Afterward, we’ll optimize our OpenAPI document (section 17.1) and enrich it with more details, such as numerical ranges and string lengths (section 18.3).

7.2 *Authoring a JSON Schema data model in OpenAPI*

Once you’ve found your way to authoring OpenAPI documents (see section 6.2), authoring JSON Schema data models follows the same path. Your OpenAPI editor will support JSON Schema, as OpenAPI requires it. But we need to discuss using JSON versus YAML and JSON Schema versus OpenAPI versions.

When JSON Schema is used independently, schemas must be in JSON format. But you can author them in YAML and convert them to JSON. For example, that’s how the OpenAPI Specification’s JSON schema is managed in its GitHub repository (<https://github.com/OAI/OpenAPI-Specification/tree/dev/src/schemas/validation>). When JSON Schema is used in a YAML OpenAPI document, schemas can be in either JSON or YAML format, but if using JSON, the schemas must also be in JSON format.

OpenAPI 3.0 and 3.1 use different versions of JSON Schema (“draft 5” and “2020-12,” respectively). Unless advanced features are used, schemas usually look the same in these two versions. See the release notes available at <https://json-schema.org/specification> for more information about the differences. The main concern is that tools supporting OpenAPI may not support all (advanced) features of the JSON Schema version used.

In this book, we use OpenAPI 3.1 (see section 6.2.4). Therefore, we use version “2020-12” of JSON Schema. Our OpenAPI document is in YAML, so our JSON schemas are in YAML (see section 6.2.5). Unless stated otherwise, the features used are compatible with JSON Schema “draft 5” used in OpenAPI 3.0.

7.3 *Adding complete resource data models to the OpenAPI document*

The first thing we did when modeling data was to design the theoretical or complete resource models containing all of a resource’s data (see section 5.2). This section initiates the description of these models, using the `Product` resource model as an example. But first, we discuss where to place this model in the OpenAPI document.

7.3.1 *Choosing a location for the resource model in the OpenAPI document*

In section 6.4.3, we learned about the `schema` properties that hold operation data descriptions. However, we should define the theoretical or complete resource models in another place, `components.schemas`, which makes them agnostic of their use and reusable across operations.

Having resources described independently of operations allows us to design them independently, as we did in section 5.2. It also gives a better view of the API subject matter when the OpenAPI document is visualized via an API documentation tool, which can be helpful when discussing and validating the design.

When working on data modeling, we realized that different operations return the “Product” complete model (see section 5.4). So, defining schemas once and using them anywhere is helpful (we’ll see how to use them in section 7.7.1). It

ensures a certain level of consistency and speeds up and secures authoring by reducing copying and pasting.



NOTE We'll discuss reusable and consistent schemas in greater depth in section 17.2 when we optimize the OpenAPI document. Consistent design, especially consistent data that is identical or similar, is essential to make an API easy to use. See section 8.1 for more.

7.3.2 Initiating the resource model description

As shown in listing 7.1, we add the `components` property, which holds reusable elements such as schemas, as well as parameters or responses (discussed in section 17.1 when optimizing the document). It contains a `schemas` property, which holds reusable JSON schemas. It is a map of *Schema objects*, each identified by a key. In our example, we define a `Product` schema, which is empty for now (`{}`). We'll fill it with the JSON schema of the "Product" resource in section 7.4.



NOTE No explicit naming conventions exist for model names in `components.schemas`; `my_model` and `myModel` are acceptable, but `PascalCase` (`MyModel`) is the most used convention, likely because it aligns with class naming in object-oriented programming. Model names appear only in documentation, not in API data exchanges.

Listing 7.1 Reusable schema

```
openapi: 3.1.0
info: ...
paths: ...
components:
  schemas:
    Product: {}
```

Where to define reusable elements

Map of reusable JSON schemas

Reusable schema identifier with an empty JSON Schema value



TIP The `info`, `paths`, and `components` properties can be in any order. However, I recommend sticking to this order as this is how most OpenAPI documents are ordered. People looking at raw OpenAPI documents are first interested in general information (`info`), the operations of the API (`paths`), and then the data models and other reusable elements (`components`).

7.4 Describing complete resource data models with JSON Schema

This section demonstrates the basics of JSON Schema, including describing typical data types (object, array, and atomic) and stating whether a property is required in an object. We turn the spreadsheet data for the "Product" resource shown in figure 7.4 into a JSON schema under `components.schemas.Product`. Although this section groups elements by types to teach JSON Schema, you'll describe them as you design them in the field.

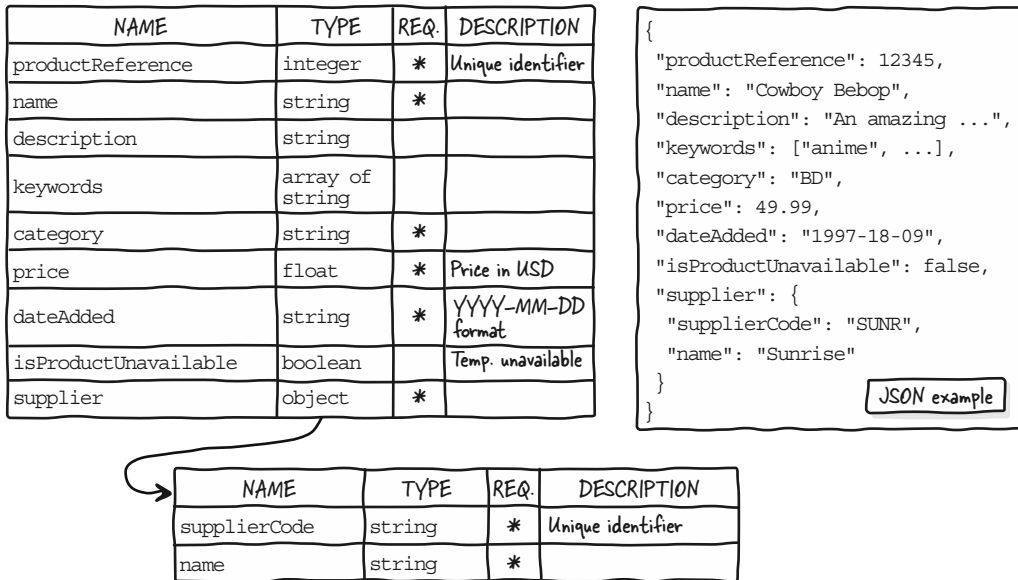


Figure 7.4 We'll reproduce (and replace) the "Product" resource complete model tables with a JSON schema.



NOTE As we said in section 5.1.3, the data models we created could be better. Our focus is on the "versatile API that does the job" layer; we'll later address user-friendliness and interoperability (section 8.2), performance (section 13.1), security (section 12.1), and implementation constraints (section 14.1).

7.4.1 Describing an object

In section 5.2, we designed the "Product" resource as an object, which is reflected in the following JSON schema.

Listing 7.2 Object type

```

...
Product:
    type: object

```

← The value of "Product" is a JSON schema.

← The JSON schema describes an object.

In the OpenAPI document, under `components.schemas.Product`, we add `type` and set its value to `object`. This JSON schema says, "This is an object," without describing its properties. Such a minimal schema could be helpful when you want to initiate and use a resource data model, leaving the fine-grained details for later.

7.4.2 Adding properties to an object

The “Product” resource isn’t empty; it contains various properties. Listing 7.3 shows how to initiate their description: add the `properties` property, a map in which keys are the properties’ names and values are their JSON schemas. We temporarily set its value to `{}` for syntax. We’ll add properties in the following sections.

Listing 7.3 Properties map

```
...
Product:
  type: object
  properties: {}
```

Map of JSON schemas describing the object's properties

7.4.3 Describing an atomic property

To describe each property of an object, we add its name as a key under `properties` whose value is its JSON schema. We start with the product’s atomic properties (not arrays or objects), shown in figure 7.5. The Required column is grayed out; we’ll discuss it in section 7.4.6.

NAME	TYPE	REQ.	DESCRIPTION
productReference	integer	*	Unique identifier
name	string	*	
description	string		
category	string	*	
price	float	*	Price in USD
dateAdded	string	*	YYYY-MM-DD format
isProductUnavailable	boolean		Temp. unavailable

```
{
  "productReference": 12345,
  "name": "Cowboy Bebop",
  "description": "An amazing ...",
  "category": "BD",
  "price": 49.99,
  "dateAdded": "1997-18-09",
  "isProductUnavailable": false,
}
```

JSON example

Figure 7.5 We’ll add the atomic properties of the “Product” resource to the Product schema without caring that they are required or optional.

We design the `productReference` property as an integer and describe it as a “Unique identifier.” The following listing shows the corresponding JSON schema. We add the `productReference` name as a key under `properties`, set its type to integer, and add the “Unique identifier” description.

Listing 7.4 A property of an object

```

...
Product:
  type: object
  properties:
    productReference:
      type: integer
      description: Unique identifier

```

The key is the property name, and the value is a JSON schema.

This property is an integer.

An optional description

All the other atomic properties can be added similarly. Not all properties have a description; for example, `category` doesn't have one.

Listing 7.5 Properties of different types

```

...
Product:
  type: object
  properties:
    productReference:
      type: integer
      description: Unique identifier
    description:
      type: string
    category:
      type: string
    price:
      type: number
      description: Price in USD
    dateAdded:
      type: string
      format: date
    isProductUnavailable:
      type: boolean
      description: Temporarily unavailable

```

The category property is a string and has no description.

Use the number type to represent a float.

Additional format information can be provided in properties of type string, integer, and number.

We map our usual programming language type names to the four atomic types defined by JSON Schema: `string`, `integer`, `number`, and `Boolean`. For example, `category: "BD"` is a string, `productReference: 12345` is an integer, `price: 49.99` is a (float) number, and `isProductUnavailable: false` is a Boolean.

The `dateAdded` property is designed as a “string” in a “YYYY-MM-DD” format but can be described formally with JSON Schema as a string with a date format (1997-18-09). Other typical values for the JSON Schema `format` are related to date and time: `time` (13:08:23+00:00), `date-time` (1997-18-09T13:08:23+00:00), and `duration` (P3D, 3 days), discussed in section 8.5.3. Consult <https://www.learnjsonschema.com/2020-12/format-annotation/format> to discover other available options for JSON Schema's `format`. OpenAPI also defines custom `format` values like `int32` and `int64` for the

integer type and float and double for number; see <https://spec.openapis.org/oas/v3.1.0#data-types>.



NOTE We'll learn to describe numerical ranges, string length, and enumeration with JSON Schema in section 18.3.

7.4.4 Describing an object property

Adding an object property is also done by adding its name as a key under properties, the value of which is its JSON schema. The “Product” resource has a supplier object property, shown in figure 7.6. It is an object with supplierCode and name properties of type string, and supplierCode is a “Unique identifier.” We'll address the required flag in section 7.4.6.

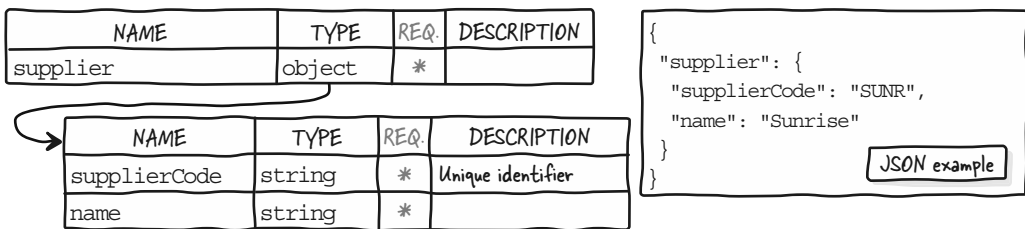


Figure 7.6 We'll add an object property and its properties without caring that they are required or optional.

Listing 7.6 shows the corresponding JSON schema. We add the supplier property, set its type to object, and add its properties map. Each sub-property name is a key in the properties map, and each contains yet another JSON schema. Both have a type set to string. The supplierCode property also has a description.

Listing 7.6 An inner object

```
...
Product:
  type: object
  properties:
    ...
    supplier:
      type: object
      properties:
        supplierCode:
          type: string
          description: Unique identifier
        name:
          type: string
```

The supplier property is an object (like the Product schema).

This object has two properties.



NOTE What we did is exactly how we started to describe this property's parent object. With JSON Schema, each element is a JSON schema regardless of

the level in the data model. Alternatively, we could have defined a dedicated model; section 7.7.4 shows how to do this.

7.4.5 Describing an array property

Adding an array or list property also requires putting its name as a key under properties, the value of which is its JSON schema; it has an array type and an items property that defines the JSON schema of its elements. As shown in figure 7.7, we designed the Product’s keywords property as an array of string; it could also have been a list of string.



Figure 7.7 We’ll add an array property to the Product schema without caring if it’s required or optional.

Listing 7.7 shows the corresponding JSON schema. We add the keywords property name as a key under the product’s properties. We set its type to array and add an items property containing the JSON schema of the array elements. We set the value of items to the type: string schema.

Listing 7.7 An array of strings

```
...
Product:
  type: object
  properties:
    ...
    keywords:
      type: array
      items:
        type: string
```

The keywords property is an array.

The items property contains the JSON Schema of the array's elements.

The keywords property is an array of string.

We can describe arrays with other types of elements, such as objects, by putting the appropriate JSON schema under items.



NOTE We’ll learn to describe array size with JSON Schema in section 18.3.

7.4.6 Stating which properties are required

Add its name to the required list to indicate that an object’s property is required. When an object is used as input, the consumer must provide the required properties; and when an object is used as output, the API must return the properties. Other properties are optional and may be absent. Figure 7.8 shows which properties we chose to mark as required when designing the “Product” resource in section 5.2.

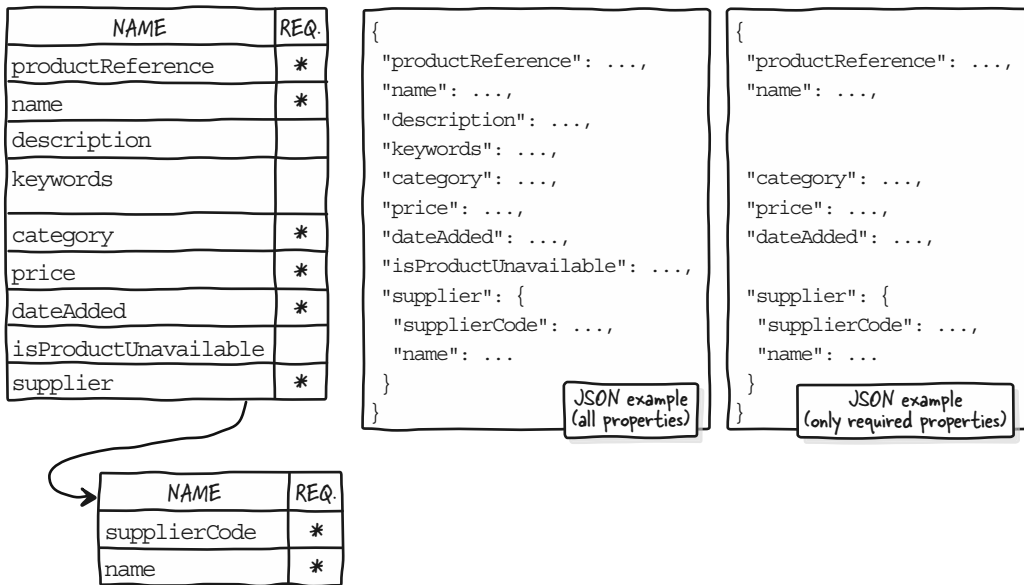


Figure 7.8 We'll indicate which properties of the "Product" resource are required.

Listing 7.8 shows the corresponding JSON schema. We add a `required` list at the root level schema and in the `supplier` property schema, which are objects. In the `supplier` object, all properties are required, so all keys of properties are present in `required`. At the root level, not all properties are required. The `description`, `keywords`, and `isProductUnavailable` property names are optional because they are absent from the `required` list.

Listing 7.8 Required properties

```
...
Product:
  type: object
  required:
    - productReference
    - name
    - category
    - price
    - dateAdded
  properties:
    productReference: ...
    name: ...
    description: ...
    keywords: ...
    category: ...
    price: ...
    dateAdded: ...
    isProductUnavailable: ...
```

← The object's required properties

Nonrequired properties are absent from the required list.

```

supplier:
  type: object
  required:
    - supplierCode
    - name
  properties:
    supplierCode: ...
    name: ...

```

← The object's required properties

We separated indicating the required statuses from describing the properties for learning purposes. You can either describe all data and then fill in the required lists on each object or add the property name to the `required` list when adding it to the schema. GUI editors often allow us to mark a required property with a check box at the property level.



TIP The `type`, `required`, and `properties` properties can be in any order. However, I recommend sticking to this order, which is the most common and facilitates reading. The `properties` block can be long and can hide `type` and `required` if they are placed after it.

7.5 Describing operation input and output data

Once we have designed (section 5.2) and described the resource model (section 7.4), we can move on to the next step: describing the operation input and output data as we design it (sections 5.3 and 5.7). We'll fill in the `schema` properties we added to the OpenAPI document when describing HTTP operations (section 6.1).

Although you'll describe all these elements as you design them in the field, here we'll learn to describe operation input and output data depending on its locations: non-body data (section 7.6) and body data (7.7). As shown in figure 7.9, body data goes in request or response bodies, and non-body data goes in all other locations (path, query parameters, and request and response header fields).

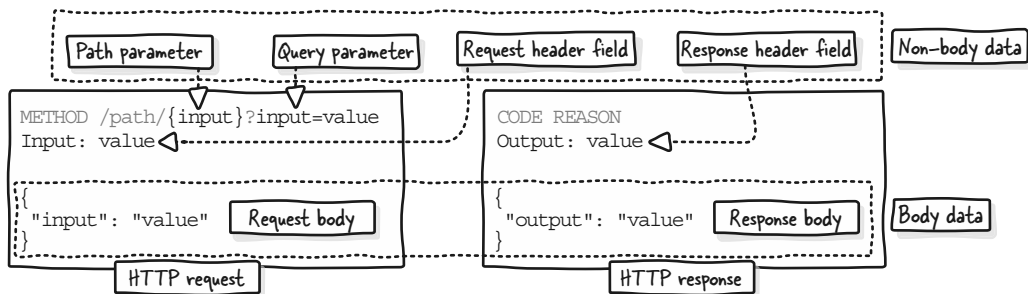


Figure 7.9 In OpenAPI, non-body and body data of HTTP requests and responses are not defined in the same location, but all use JSON Schema.

7.6 Describing operation non-body data

This section demonstrates how to describe non-body data (parameters, query parameters, and request and response header fields) using copied or ad hoc inline schemas: JSON schemas put directly under a `schema` property.



NOTE In addition to inline schemas, section 7.7 demonstrates how to describe body data using references to reusable schemas and mixing references and inline schemas. All options apply to the body or non-body data.

7.6.1 Describing non-body request parameters with inline schemas

In an OpenAPI document, the non-body input parameters are described in the `parameters` list at the path or operation level. The JSON schema describing their data model goes in the `schema` property.

Listing 7.9 Inline parameter schema

```
...
paths:
  ...
  /products/{productReference}:
    summary: Product
    parameters:
      - name: productReference
        in: path
        required: true
        schema:
          type: integer
    components:
      schemas:
        Product:
          type: object
          properties:
            productReference:
              type: integer
  ...
```

← Path-level parameters list

← The parameter's inline schema is copied and pasted from the corresponding property of the Product reusable schema.

Often, a parameter matches a property of the manipulated resource. We copied the product's `productReference` property model to design the `productReference` path parameter in section 5.3. We proceed similarly with OpenAPI in listing 7.9: the schema of the `productReference` path parameter in the `/products/{productReference}` path is a copy of the JSON schema of the `productReference` property of the `Product` reusable schema. This technique applies to any parameter regardless of location (`in`) and level (path or operation); we could proceed similarly for the `category` query parameter.



NOTE We'll learn how to limit information duplication when optimizing the OpenAPI document. Section 17.2.3 explains how to use deep references to reuse part of data models. A parameter may be used across different operations; section 17.3.2 shows how to define reusable parameters.

7.6.2 Tweaking non-atomic parameter serialization

Query parameters are not always atomic values; for instance, it's common to have an array of strings, which is the case for the `keywords` search filter. The OpenAPI Specification allows us to describe how such a non-atomic value is serialized in the URL. In the following listing, we proceed as in the previous section, copying the parameter schema from the corresponding property in the `Product` schema. Additionally, we explicitly indicate how to serialize the parameter's array value.

Listing 7.10 Array query parameter serialization

```
...
paths:
  /products:
    ...
    get:
      summary: Search for products
      parameters:
        - name: keywords
          in: query
          style: form
          explode: false
          schema:
            type: array
            items:
              type: string
        ...
      ...
components:
  schemas:
    Product:
      ...
      properties:
        ...
        keywords:
          type: array
          items:
            type: string
        ...
```

← **Operation-level parameters list**

Controls object or array serialization

The parameter's inline schema is copied and pasted from the corresponding property of the Product reusable schema.

By default, an array of atomics is represented as multiple query parameters with the same name: `keywords=anime&keywords=space`. Here, we set the parameter `style` to `form` (which is its default value) and `explode` to `false` (the default is `true`) to have a single query parameter and comma-separated values (CSV): `keywords=anime,space`.



NOTE Section 17.3.3 will show how to use these options to serialize an object. Consult <https://spec.openapis.org/oas/v3.1.0#style-examples> to discover all possible options for `style` and `explode` and their effect on arrays and objects.

7.6.3 Describing response headers with inline schemas

In an OpenAPI document, response header fields are defined in the response's `headers` map. The JSON schema describing their data model goes in the `schema` property.

Some non-body data schemas can't be copied from a predefined resource, such as the `Location` header of the `POST /products` operation (URL of the created resource; see sections 4.6.2 and 4.6.3). It needs an ad hoc schema. We describe it ad hoc with the `type: string` JSON schema instead of copying it.

Listing 7.11 Inline response header schema

```
...
paths:
  /products:
    ...
    post:
      ...
      responses:
        "201":
          headers:
            Location:
              description: Product URL
              required: true
              schema:
                type: string
```

Ad hoc inline schema
not copied from a
resource schema



NOTE OpenAPI allows us to define reusable response headers. We'll discuss this when optimizing the OpenAPI document in section 17.5.1.

7.7 Describing operation body data

This section demonstrates how to describe request and response body data using references to reusable schemas defined under `components.schemas` instead of inline schemas. We explain and demonstrate the use of schema references, illustrate resource schema derivation, and show how to mix inline schemas and references.



NOTE In addition to references to schemas, section 7.6 demonstrates how to describe non-body data using inline schemas. All options apply to the body or non-body data. Section 14.3 discusses binary data.

7.7.1 Using references to resource models in response bodies

In OpenAPI, operation responses are defined in the `responses` map; keys are HTTP status codes and values *Response objects*. The JSON schema of each is specified in `content.application/json.schema` (see sections 6.7 and 6.8).

Whatever its location, a JSON schema can either be an inline schema (see section 7.6) or a reference using a `$ref` property that targets a schema using a *JSON pointer*. It is a standard independent of JSON Schema and OpenAPI indicating the location of a value in a JSON (or YAML) document (see <https://datatracker.ietf.org/doc/html/rfc6901> for more information).

The `Product` schema under `components.schemas` (see section 7.3) represents the “Product” resource model that different operations return: for example, “Add a product to the catalog” and “Get product details” (see section 5.3). Instead of copying and pasting the schema in both operations’ responses, we can reference it, as shown in listing 7.12.



NOTE Referencing schemas avoids duplication and unwanted variations; if we need to modify the schema, we only need to do it in one place. We'll thoroughly discuss the benefits and advanced use of references when optimizing the OpenAPI document in section 17.2.

Listing 7.12 References to reusable schemas

```
...
paths:
  /products:
    ...
    post:
      ...
      responses:
        "201":
          ...
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Product"
...
  /products/{productReference}:
    get:
      ...
      responses:
        "200":
          ...
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Product"
...
components:
  schemas:
    Product: ...
```

Reference to the Product
schema defined under
components.schemas

In the schema of the successful 2XX responses of POST `/products` and GET `/products/{productReference}`, we added `$ref` properties. They share the same JSON Pointer value, `#/components/schemas/Product`, indicating the location of the actual JSON schema. It's enclosed in double quotes (") because the dash (#) marks comments in YAML.

The dash (#) is the document's root. The following segments (`components`, `schemas`, and `Product`) separated by slashes (/) represent properties in the document. This means the schema is in `components.schemas.Product`.



NOTE We can do the same for error responses: define a unique `Error` JSON schema (see section 5.3.6) under `components.schemas` and reference it in all 4XX and 5XX response schema properties using the `#/components/schemas/Error` JSON pointer. Many error responses are similar; OpenAPI allows us to

define them once and use them in multiple places. We'll discuss this when optimizing the OpenAPI document in section 17.5.

7.7.2 Deriving the complete resource model to create other reusable models

We learned to design models derived from a complete resource model in sections 5.3 and 5.4. For instance, we created a “Product Creation or Replacement” model that is used as a request body of the “Add a product to the catalog” and “Modify a product” operations by stripping the `Product` model of the properties managed by the implementation in the context of creation and replacement (see figure 7.10).

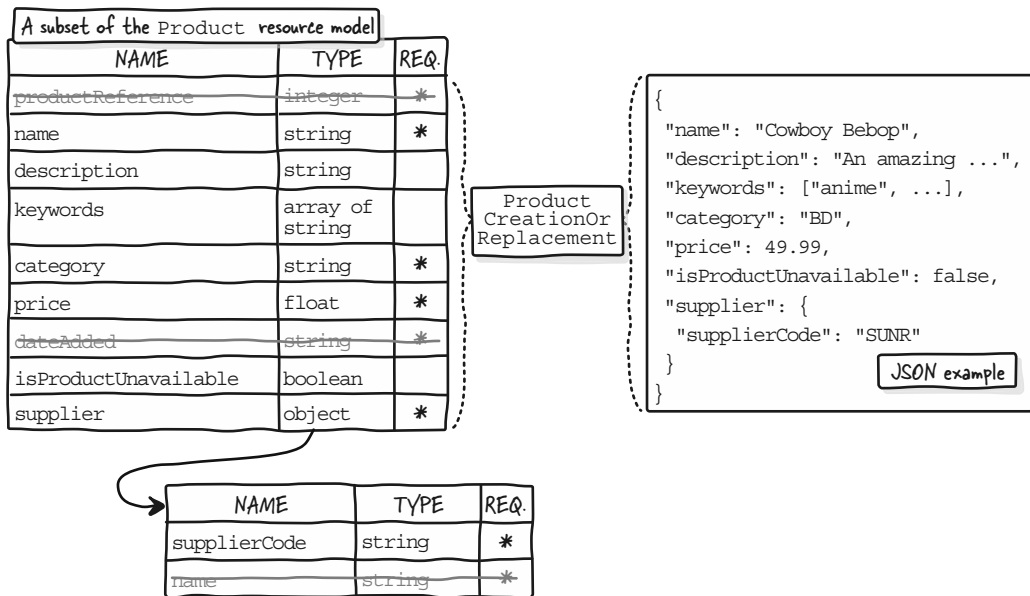


Figure 7.10 We'll derive the `Product` model into the `ProductCreationOrReplacement` model.

As shown in listings 7.13 and 7.14, to derive the `Product` model in the OpenAPI document, we add a `ProductCreationOrReplacement` schema under `components.schemas`, copy and paste the content of the `Product` schema into it, and remove the unnecessary properties (`productReference`, `dateAdded`, and `supplier.name`) from properties and required.

Listing 7.13 Product schema

```
...
components:
  schemas:
    Product:
      type: object
```

← Reusable Product schema

```

required:
  - productReference
  - name
  - category
  - price
  - supplier
properties:
  productReference: ...
  name: ...
  description: ...
  keywords: ...
  category: ...
  price: ...
  dateAdded: ...
  isProductUnavailable: ...
  supplier:
    type: object
    required:
      - supplierCode
      - name
    properties:
      supplierCode: ...
      name: ...

```

Properties managed by the implementation in creation or replacement contexts

Listing 7.14 ProductCreateOrReplace schema

```

...
components:
  schemas:
    Product: ...
    ProductCreationOrReplacement:
      type: object
      required:
        - name
        - category
        - price
        - supplier
      properties:
        name: ...
        description: ...
        keywords: ...
        category: ...
        price: ...
        isProductUnavailable: ...
        supplier:
          type: object
          required:
            - supplierCode
          properties:
            supplierCode: ...

```

ProductCreationOrReplacement is a copy of the Product schema.

Implementation-managed properties have been removed from the required lists and the properties maps.



CAUTION Copying and pasting may cause variations in the long run, but it's simpler at this stage. We may replace these schemas with a single read-and-write schema when optimizing the OpenAPI document (section 17.2).


7.7.3 Using references to resource models in request bodies

In OpenAPI, an operation request body is defined in `requestBody`. Its JSON schema is specified in `content.application/json.schema` (see section 6.6.2).

Similarly to what we did with response body schemas (section 7.7.1), we can avoid copying and pasting schemas used in different operations' request bodies with references. Listing 7.15 demonstrates the use of a reference to the `ProductCreationOrReplacement` (added in section 7.7.2) in the request bodies of the `POST /products` and `PUT /products/{productReference}` operations. In both schema properties, we add a `$ref` property whose value is `#/components/schemas/ProductCreationOrReplacement`. It targets the `ProductCreationOrReplacement` reusable schema in `components.schemas`.

Listing 7.15 References to reusable schemas

```
...
paths:
  /products:
    ...
    post:
      requestBody:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas
                <lineararrow/>/ProductCreationOrReplacement"
            ...
  /products/{productReference}:
    ...
    put:
      requestBody:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas
                <lineararrow/>/ProductCreationOrReplacement"
            ...
components:
  schemas:
    ...
    ProductCreationOrReplacement: ...
```



A reference to the **ProductCreation-OrReplacement** schema located under **components.schemas**

7.7.4 Mixing inline schema and reference

A JSON schema can be inline or referenced, allowing both approaches to be mixed. As shown in figure 7.11, we designed the `ProductSummary` model as a subset of the `Product` model. We used it in the list of products that “Search for products” returns (see section 5.3.2). We can use a mixed approach for this response's schema in the OpenAPI document.

In listing 7.16, and as done in section 7.7.2, we add the `ProductSummary` schema in `components.schemas` and copy, paste, and adapt the `Product` schema. Then, as shown

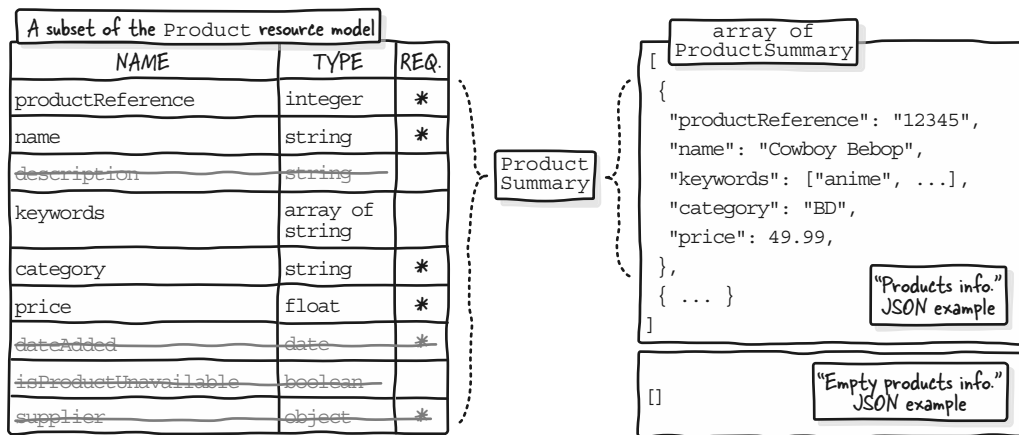


Figure 7.11 The ProductSummary model is a subset of the Product model for lists.

in listing 7.17, in the 200 response of the GET /products operation schema, we define an inline schema of type array whose items are a reference (\$ref) to this schema (#/components/schemas/ProductSummary).

Listing 7.16 ProductSummary schema

```
...
components:
  schemas:
    Product: ...
    ...
    ProductSummary:
      type: object
      required:
        - productReference
        - name
        - category
        - price
      properties:
        productReference: ...
        name: ...
        description: ...
        category: ...
        price: ...
```

← The ProductSummary reusable schema is the summarized version of the Product schema.

Listing 7.17 Inline schema and reference

```
...
paths:
  /products:
    ...
    get:
      ...
```



```

responses:
  "200":
    content:
      application/json:
        schema:
          type: array
          items:
            $ref: "#/components/schemas
            <lineararrow/>/ProductSummary"

```

Inline array schema

The array elements schema is a reference to the ProductSummary reusable schema.

Similarly, when describing the Product model, we could have defined a SupplierSummary schema under components.schemas and referenced it in the supplier property instead of defining an inline schema, as illustrated in the following listing. We could use this model in the response to “Search for suppliers.”

Listing 7.18 Reference in a reusable schema

```

...
components:
  schemas:
    Product:
      type: object
      ...
      properties:
        ...
        supplier:
          $ref: "#/components/schemas/SupplierSummary"
    SupplierSummary:
      type: object
      required: ...
      properties:
        supplierCode: ...
        name: ...

```

Uses a reference to another schema

Targeted schema



CAUTION Splitting resource models into smaller models can be helpful to avoid duplication and limit the risk of inconsistency. However, don’t try to optimize models too much in this first pass; focus on describing the API and not on OpenAPI concerns. We’ll optimize the OpenAPI file and learn new techniques in the third step of “Describing the programming interface,” including considering how to split models; refer to section 17.1.

Summary

- JSON Schema is an independent format that OpenAPI uses to describe data.
- Describe data with JSON schemas in OpenAPI while designing it after adding HTTP operations.
- Describe resource models as reusable schemas under components.schemas for better API subject matter view and reusability across operations.
- JSON schemas typically start with a type definition (object, array, string, number, integer, or Boolean).

- An object has a `properties` map with property names as keys and their JSON schemas as values.
- The `required` list of an object contains its required properties' names.
- The `items` property of an array contains the JSON schema of its elements.
- Operations input and output data JSON schemas go into the `schema` properties added when describing the HTTP operations.
- A reusable schema defined in `components.schemas.Name` can be referenced with a `$ref` property whose value is a `#/components/schemas/Name` JSON Pointer.
- Use reference to reusable schemas to avoid duplication and unwanted variations in request and response bodies.
- Describe resource model derivations as reusable schemas under `components.schemas`.
- The value of a `schema` property can be an inline schema, a reference (`$ref`) to a schema, or a mix of both options.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 7.1

You're designing an API for a pixel-art device comprising a "screen" composed of 36-by-36 luminous squares. A screen has an `id` (`SC001`, for example) and a matrix of `pixels` (array of array). Each pixel has an `rgb` array (`[10, 23, 45]`, for example), a brightness percentage between 0 and 1 (`0.3`, for example), and an `on` flag. All properties are required except the brightness. Describe the corresponding `Screen`, `Pixels`, and `Pixel` reusable schemas with OpenAPI.

Exercise 7.2

You're designing an API for a music streaming service. Use OpenAPI to describe the following sample API call.

Listing 7.19 Sample call

```
GET /artists/pink-floyd/albums?releaseYear=1969
```

```
200 OK
```

```
Content-Type: application/json
```

```
[
  {
    "id": "A890",
    "name": "More",
    "mainArtist": {
```

```
    "id": "pink-floyd",
    "name": "Pink Floyd"
  },
  "releaseYear": 1969,
  "comment": "Movie soundtrack"
},
{
  "id": "A789",
  "name": "Ummagumma",
  "mainArtist": {
    "id": "pink-floyd",
    "name": "Pink Floyd"
  },
  "releaseYear": 1969
},
]
```

Exercise 7.3

You're working on an API for a recruitment platform. Given the following sample of job offer JSON data, describe the “Create a job offer” and “Update (replace) a job offer” operations with OpenAPI.

Listing 7.20 Job offer sample data

```
{
  "reference": "xz789",
  "created": "2024-12-08",
  "title": "Technical book author",
  "description": "Writing technical books"
}
```


Part 2

User-friendly, interoperable API design

Designing a reusable API that does the right job is already an outstanding achievement. However, we can enhance our design so developers and their consuming applications can use our API quickly and seamlessly without complex thinking and coding. This will increase developers' productivity and make them feel as though they have superpowers. For instance, it could be unnerving that with the SOCNET API, adding a friend requires a user ID, whereas sending a message to a friend requires a friend ID, although they represent the same concept: a user. We must also ensure that our design contributes to building an outstanding end-user experience; for instance, we don't want end users to face an unhelpful error message like "Can't create the account." The icing on the cake is that handling all these concerns increases the reusability and flexibility of our API.

This part of the book focuses on the second layer of API design: designing a user-friendly, interoperable API (section 1.7.2). We dive into these concerns for each level we must consider. Chapter 8 introduces the concepts of user-friendliness and interoperability and then focuses on data: choosing, naming, typing, and organizing data, as well as ensuring its consistency and standardization. Chapter 9 discusses operations, diving into creating easy-to-provide inputs and ready-to-use outputs, filtering and pagination, handling various data formats, and erroring. Chapter 10 discusses flows or sequences of operations and how to make them concise, error-limiting, and flexible. Chapter 11 looks at the API as a whole: naming, sizing, and splitting it, as well as adding browsing or discovery features.

Designing user-friendly, interoperable data

This chapter covers

- Designing ready-to-use data
- Choosing atomic data types and formats
- Organizing data in objects and arrays
- Choosing data granularity
- Designing names
- Designing consistent and standard data

Imagine a washing machine that shows “EC 50400” when it’s started. You must consult the manual to decode this as the end-of-cycle time in seconds from 00:00. Even knowing this, you must do the math to get meaningful information. A clear message like “Washing ends at 2:35 pm” would be more useful. Although this washing machine fulfills its users’ needs (washing clothes and indicating when it’s done), it is not easy to understand and use: it’s not user-friendly.

Now, imagine that the washing machine uses pictograms for washing programs, like a bucket with two lines underneath for delicate cycles. These pictograms help users select the right program from clothing labels. If your machine lacks them, you can still refer to the meaning of the pictograms on the label for an appropriate setting. I never remember their meaning, so I take a photo of the pictograms with

my iPhone, swipe up the photo, and tap “Look up Laundry Care” to get their meanings. Although not always user-friendly, these universal symbols, defined by the ISO 3758 standard, ensure interoperability in the laundry world.

It’s the same in the API world; meeting user needs is a good start, but an API design must also be user-friendly and interoperable. Unhelpful data like `{"ec": 50400}` and nonstandard formats such as `{"endOfCycle": "2-35"}` lead to questions, errors, complicated coding, and lengthy development. User-friendly, interoperable API design helps create straightforward APIs that developers and their applications can use easily and intuitively and can even love.

This chapter introduces the second layer of API design covered by part 2 of this book, focusing on user-friendliness and interoperability. We then explore these aspects from a data perspective, explaining what makes data user-friendly and interoperable and when to address these features. We demonstrate how to create ready-to-use data, choose suitable atomic data types and formats, organize data effectively, determine data granularity, and craft user-friendly names. Finally, this chapter highlights the importance of data consistency and standardization in this context.

8.1 *The user-friendliness and interoperability layer of API design*

As illustrated in figure 8.1, we’re done with the first layer of API design. We have designed and described a versatile API that does the right job. It exposes the capabilities meeting the needs identified in the Define stage of the API lifecycle and can handle other scenarios; we’ll cover the last step of the design process, “Enrich the API

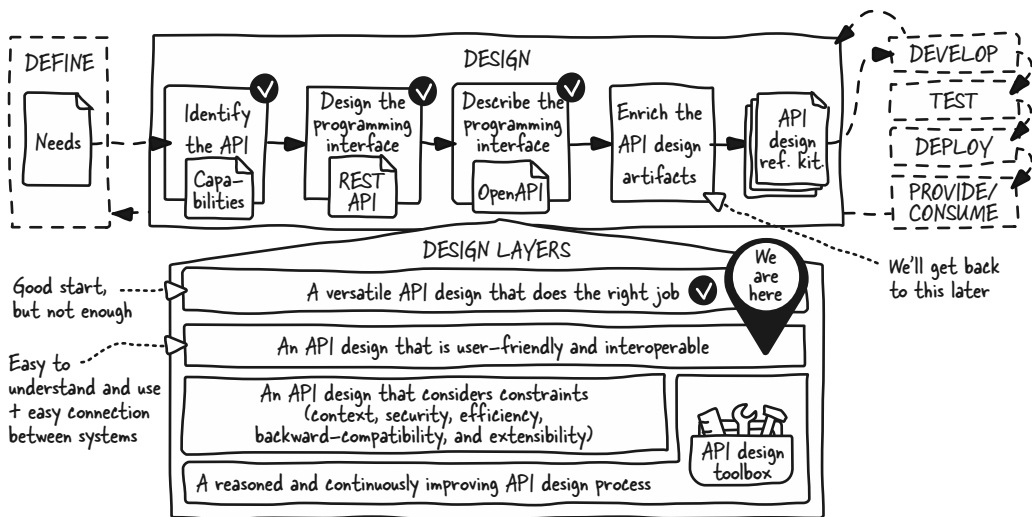


Figure 8.1 Designing a versatile API that does the job is a good start, but we must ensure that it’s user-friendly and interoperable.

design artifacts,” in section 19.1. We will now work on the second layer, ensuring that our API design is user-friendly and interoperable (section 1.7.2). A user-friendly API is easy to understand and use. An interoperable API enables systems to work together efficiently without complex processing or coding.

To introduce user-friendliness and interoperability in the context of API design, this section discusses API user experience and clarifies which users’ experiences we’re interested in. Then we explain how API design user-friendliness and interoperability affect user experience.

8.1.1 Overview of the API user experience

User experience (UX) refers to a user’s overall perception and interaction with products or systems like washing machines, microwave ovens, websites, mobile applications, command-line interfaces, and APIs. Beyond design, various factors affect the UX of APIs, from how we find and use them to their reliability.

To send an SMS from an application, we search for “SMS API” or “best SMS API” on a search engine to find SMS API providers with good reviews. Finding interactive documentation that allows us to understand the API and make API calls in seconds on the developer portal is an excellent start. Retrieving credentials for our application and coding the API calls swiftly is even better. If the pricing and terms and conditions are clear and look OK, we happily input our credit card details. Seeing our users always receive their SMSs quickly is the ultimate satisfaction.

Hiccups can ruin the experience, stopping or preventing us from using the API and making us look for alternatives. The API may not fulfill our needs; some countries we target may not be listed in the terms and conditions. The pricing model may be costly. Navigating the developer portal and retrieving credentials can be difficult. API security may not match our standards. Using the API and coding calls may be challenging, leading to complex and costly development. The API may be unreliable; SMSs may take too long to arrive, annoying end users.

The API UX is not only for public or partner APIs; it also matters for private APIs. They often come with portals that are more basic and less self-service, but this is not a problem. The big problem is that we likely can’t switch to alternatives if our private APIs are complicated, require complex and long coding, or are unreliable and affect our end users.

8.1.2 Which users’ experiences matter to us?

The main users of APIs are developers of applications consuming them, which is why API UX is often called *developer experience* (DX). However, not just typical developers use APIs; business analysts may use them in spreadsheets, and QA engineers test them via API clients. With AI’s rise, applications using APIs can also be seen as users. Decision-makers like architects, SMEs, and business analysts may explore APIs on developer portals to find the ones that meet their needs. Finally, although end users don’t directly engage with APIs, they expect a quick, seamless experience with applications that use them. API design can significantly affect all these users.



NOTE In this book, *developer*, *consumer developer*, and *consumer* refer to anyone directly interacting with an API, whatever their actual profile and objectives (coding an application using an API, using this via a client, or analyzing it for decisions).

8.1.3 How API design user-friendliness and interoperability affect UX

Even if it exposes the right capabilities, a non-user-friendly or non-interoperable API design can negatively affect developers' and end users' experiences, influencing the decision to use the API, as shown in figure 8.2. If an SMS API's documentation outlines a complex six-step operations flow for sending an SMS, we won't choose it unless it's a private API that we're forced to use. Such complexity is unnecessary for a simple action. Additionally, it may require significant UI refactoring, complicating our application and frustrating end users. Similarly, if sending an SMS requires a cryptic `POST /x2501` operation that returns a noncompliant and thus noninteroperable 200 OK with an unclear error message for missing data, we'll avoid it.

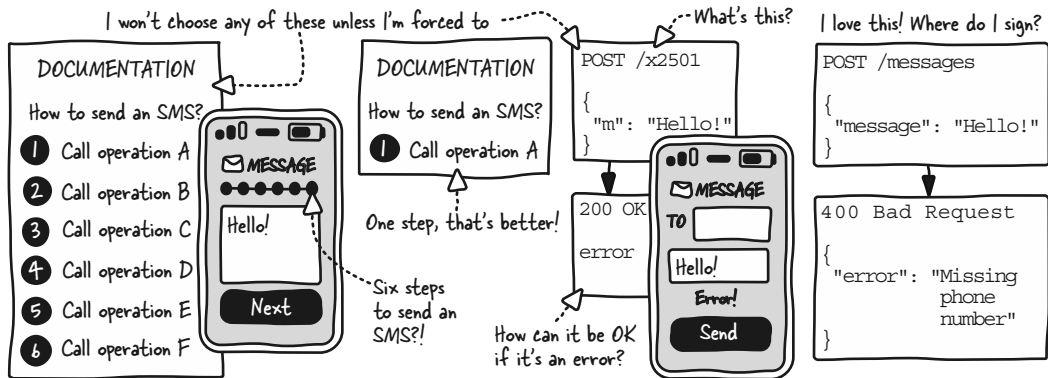


Figure 8.2 The design of an API may affect the developer and user experiences, making decision-makers think twice before choosing it (if they have a choice).

In contrast, an API with an operation like `POST /messages` that returns 400 Bad Request with a clear message like `{"error": "Missing phone number"}` will catch our eye. We'll quickly integrate such a user-friendly, interoperable API, helping us provide end users with a simple UI and clear error feedback.



NOTE A user-friendly, interoperable design enables efficient development and can enhance UX. Remember the consequences of terrible API design (section 1.2). User-friendliness and interoperability matter at all levels when designing an API: data (the rest of this chapter's focus, starting with section 8.2), operations (section 9.1), operations flows (section 10.1), and APIs (section 11.1).

8.2 What makes data user-friendly and interoperable?

User-friendly, interoperable data is essential to enable efficient development, but what are the characteristics of such data? Regardless of the context, whether it is a resource path, a path parameter, a query parameter, a header, a request body, or a successful or error response body, user-friendly data meets user needs, helps find and interpret information, limits consumers' work, and is consistent. Interoperable data is consistent, too, but also standard. This section uses a Car Rental API as an example.

8.2.1 User-friendly data meets user needs

Although meeting user needs alone is not sufficient on its own, it fosters the design of user-friendly data. Suppose we create a Car Rental API for third parties. Our API users need information about vehicles that are available in a shop. But that doesn't mean we should include vehicles' data in the shop's data; doing so will complicate and blur the meaning of our data. We likely identified a "Search for vehicles" operation that will return this data. We're used to referring to a shop's location as a "Business area reference" when using our internal jargon. However, we should use "Shop location" instead, which makes more sense in this API's context and related needs. This doesn't mean we can't use our jargon; it may suit another API with different needs.



NOTE Remember from section 2.1.2 that focusing on users' needs was our guide when identifying API capabilities. It also helps name and shape user-friendly data by forcing us to look at our usual capabilities and data from an outside-in perspective. Before making existing APIs more user-friendly, ensure that they meet user needs.

8.2.2 User-friendly data helps us find and interpret information

User-friendly data helps developers find and interpret information. As shown in figure 8.3, it's difficult for developers to understand that shop location data is available or find it when using cryptically named and scattered `la` (latitude) and `lo` (longitude) properties; renaming them `latitude` and `longitude` helps, but data is still lost among other elements. On the other hand, an explicitly named `location` object grouping `latitude` and `longitude` properties is easy to find and interpret.

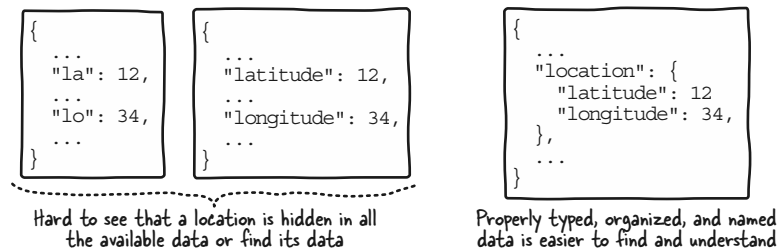


Figure 8.3 User-friendly data helps find and interpret information easily and limits consumers' work.

8.2.3 User-friendly data limits consumers' work

User-friendly data limits consumers' work and helps them achieve their objectives more efficiently. As shown in figure 8.4, we can add a ready-to-use address alongside geographic coordinates to eliminate the need for consumers to perform reverse geocoding. Once done, we might consider removing the geographic coordinates now that we have the address, because the address may suffice for our needs.

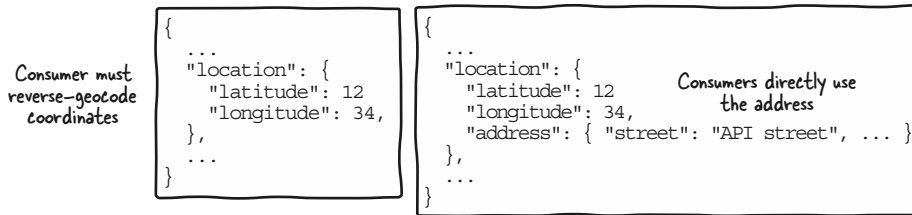


Figure 8.4 Adding the address prevents consumers from reverse-geocoding the coordinates.



NOTE Crafting user-friendly data can uncover previously unknown needs or make us reevaluate API capabilities. Modifying data will require us to ensure that there's no gap in our design, as we did when first modeling it (section 5.5).

8.2.4 User-friendly data is consistent

User-friendly data is consistent, which makes it intuitive and allows developers to rely on their experience to understand and use it quickly. As shown in figure 8.5, suppose we represent the location of a car with a "place": ["34° 0' 0", "12° 0' 0"] array where the values are the longitude and latitude in degrees, minutes, and seconds, respectively, instead of decimal degrees. Information will be harder to find, and we'll confuse users. Users will instinctively know how to identify and use location data if we use a similar name, location or lastLocation, and the same format and units for all locations.

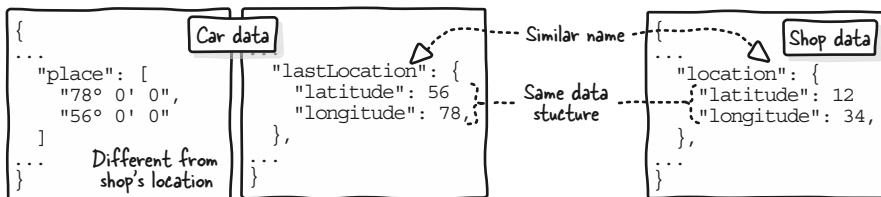


Figure 8.5 Different, intuitive, and interoperable representations of a location



NOTE An intuitive API design gives developers superpowers, allowing them to guess available data and operations. This creates an invaluable “wow” effect during use.

8.2.5 Interoperable data is consistent and standard

Interoperability is essential for APIs, allowing systems to collaborate efficiently without complex processing or coding and easing developers' work. An API operation's interoperable data can be easily used as input for other operations of the same or other APIs. Moreover, any consuming application can easily provide or use interoperable data. Achieving interoperability involves using custom (or local), domain-specific, or generic standards.

Figure 8.6 illustrates the consistent use of an object with `latitude`, `longitude`, and `address` properties to represent a location across data models. This enhances our data's local interoperability. Adopting a local standard is beneficial, but we should use existing domain-specific or generic standards whenever possible to avoid reinventing the wheel and improve interoperability. Instead of inventing our geographic point representations, we can use GeoJSON's *Geometry object* as defined by RFC 7946 (see <https://geojson.org/>). We can replace `latitude` and `longitude` with an object that includes a `"type": "Point"` string and a `"coordinates": [12, 34]` array in decimal degrees. Although not overly user-friendly (I never remember which item is the latitude or longitude in the coordinates array), this standardized data simplifies the sharing and interpretation of coordinates.

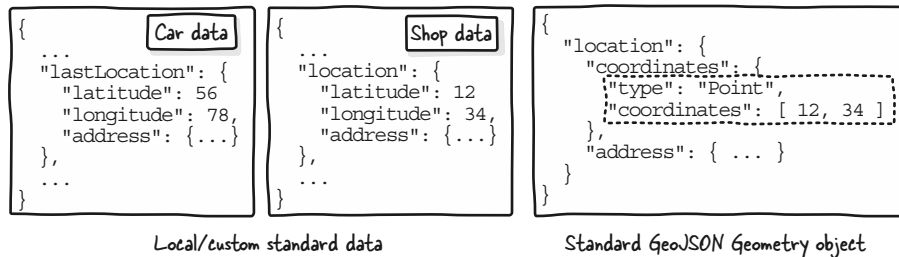


Figure 8.6 Using custom or local standards or actual standards makes data interoperable.



NOTE Interoperable standards may overlook user-friendliness or may need to sacrifice it for efficiency. Section 8.9.1 explores user-friendliness and interoperability.

8.3 When and how to design user-friendly, interoperable data

Now that we've seen what makes data user-friendly and interoperable, we can clarify which data needs to be user-friendly and interoperable, when to address these concerns, and how to design such data.

8.3.1 Which data must be user-friendly and interoperable?

We must ensure that any of the API's pieces of data are user-friendly and interoperable, including

- Resource paths
- Resource data models
- Operation inputs (headers, path parameters, query parameters, bodies)
- Operation outputs (headers, bodies)
- Operation errors (headers, bodies)



NOTE This chapter focuses on the resource theoretical or complete models designed in section 5.2 to explain principles we'll use to design resource paths and operation inputs, outputs, and errors when working on user-friendly, interoperable operations (section 9.1).

8.3.2 When to address user-friendly, interoperable data

As shown in figure 8.7, we address user-friendly, interoperable data via a secondary pass through our data modeling (section 5.1) to keep the design process efficient; to a lesser extent, we'll also revisit HTTP representations to enhance paths and path parameters, but we'll discuss this when working on operations (section 9.1). For instance, we don't waste time discussing longitude versus lng or debating about GeoJSON when initially modeling the "Car Rental Shop" resource; identifying that coordinates are needed is sufficient. There will probably be some back and forth between fulfilling user needs and being user-friendly and interoperable, as working on these aspects may raise questions and new ideas, such as when we add an address to the shop location data (section 8.2.3).

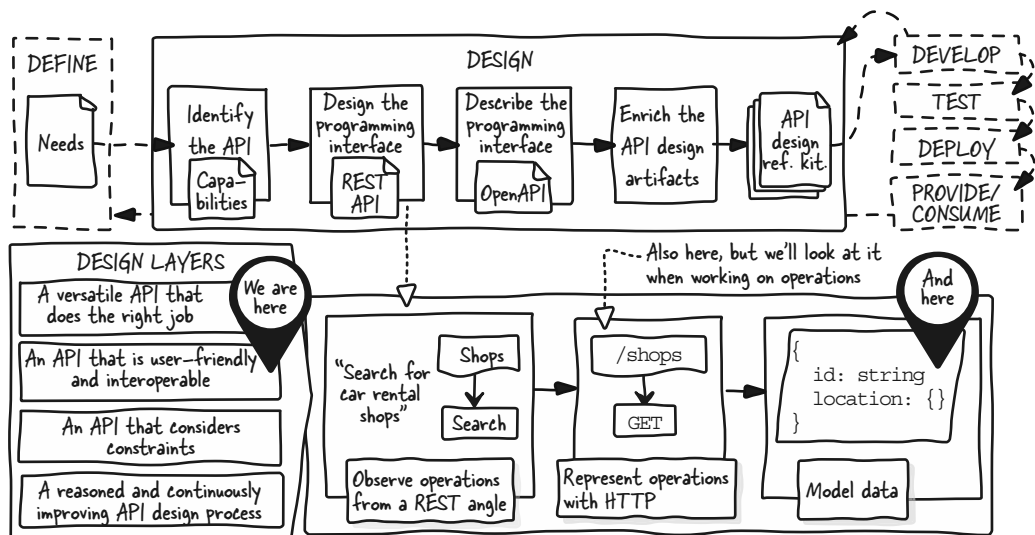


Figure 8.7 Once we have modeled versatile data that does the job, we can revisit it to ensure that it's user-friendly and interoperable.



NOTE The initial data modeling will require less and less rework to become user-friendly and interoperable, thanks to experience and the help of API design guidelines we'll craft to facilitate our work (section 16.3).

8.3.3 How to design user-friendly, interoperable data

As summarized in figure 8.8, to design user-friendly, interoperable data, we ensure that the selected data is ready to use without thinking or processing by verifying that we chose the correct data (“Business area reference” versus “Location”) and enhancing it (the reverse-geocoded address addition). Then we work on types and organization to make data easy to interpret and find (the `location` object). Afterward, we reconsider data granularity to keep only relevant data (should we include all available cars in the rental shop resource?). Ultimately, we craft easy-to-understand names that describe the data appropriately (1 versus `location` versus `place`). We must always ensure that we are making consistent design decisions and aim for standardization (`location` name and GeoJSON format). The rest of this chapter explores these concerns using a new example: a Banking API.

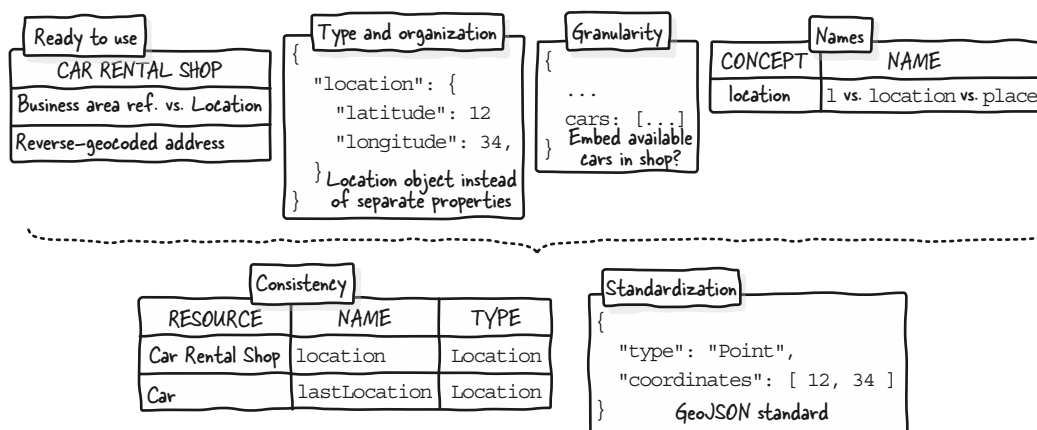


Figure 8.8 How to design user-friendly, interoperable data



NOTE This chapter focuses on user-friendly, interoperable data; however, data modeling must also address meeting user needs (section 5.1), performance (section 13.1), security (section 12.1), and implementation constraints (section 14.1).

8.4 Selecting and crafting ready-to-use data

Our first concern when modeling data is selecting the data that fulfills the identified needs (section 5.1). But afterward, we must check whether the data is ready to use as individual pieces and as a whole so that users can use or provide it directly without

thinking or processing. In the process, we may discover gaps in our needs analysis or exposure of internal workings. This process involves

- Selecting simple yet effective data
- Enhancing data with supporting or processed data
- Using well-known or standard data

This section discusses these concerns by modeling a Bank Account resource model for a Banking API, illustrated in figure 8.9. This Banking API proposes typical retail or personal banking capabilities, such as listing bank accounts, checking an account balance, and transferring money.

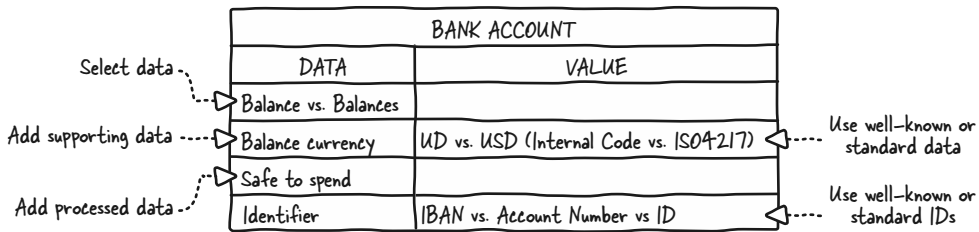


Figure 8.9 Designing ready-to-use data for the Bank Account model

8.4.1 *Choosing simple and meaningful but useful data*

The same information can be present in different forms in the system(s) behind an API. We must select simple and meaningful ones so consumers can easily understand and use them. But we must also ensure that the data is not too simplistic.

Because providing the amount of money an account holds was a requirement, we included the balances in the initial data modeling of the Bank Account model. Bank accounts have multiple balances due to various calculation methods in banking systems.

A Bank account model with all balances offers versatility for advanced use cases. However, it increases complexity, requiring consumers to decide which balance to use according to the use case. Instead, we can pick the most commonly used balance in banking websites or applications. It's simpler but reduces the possibilities.

We aim for simplicity but must consider the targeted users' needs and related API capabilities when choosing an option. A single balance makes sense if no use cases involve multiple balance types or the users are non-experts. However, if different types of balances are necessary or we target banking experts, we can keep all balances. Also, if the balances are a crucial concept, we may have missed balance-related capabilities or resources during the needs analysis. If they are accessible by other means, we could remove the balances from the Bank Account model; and why not keep the most common one? That will please both expert and non-expert users.



NOTE Remember not to overemphasize simplicity when working on user-friendliness; always confront optimizations to identified needs and capabilities.

8.4.2 Adding supporting data to ease and secure interpretation

Interpreting a piece of information may require unnecessary speculation or guesswork. Adding supporting data enhances comprehension, reduces errors, and facilitates developers' work.

The `Bank Account` model balance is an amount. It can be 123, for instance, but 123 what? If a European bank provides the API, it's probably €123 (123 Euros). However, not all European countries use this currency, and even those that do may propose accounts with currencies other than the Euro. Adding the balance currency makes interpreting the balance easy and accurate. Also, if we keep multiple balances, we can add a default flag to the most common one so non-expert consumers can choose it without thinking.

8.4.3 Adding processed data to reduce consumer effort

Picking relevant raw information is not always enough to make data user-friendly. Adding processed data can be more effective and may even replace the initially selected raw data.

One of the Banking API requirements is to help users evaluate how much money they can spend. To do so, during the first pass of data modeling, we added approved overdraft information (`active` flag and amount) along with the balance. The “overdraft facility” is a service allowing account owners to overdraw up to a specific amount without fees (which is common in France). However, evaluating how much end users can spend requires the consumer to add together the overdraft facility amount and the balance if the overdraft facility is active (`active` flag and amount), along with the balance.

Including a safe-to-spend amount simplifies the consumer's job. We can also reevaluate whether balance and overdraft information is necessary with this new addition. Depending on our API's purpose and target audience, a super-simplified `Bank Account` model may make sense.



NOTE Remember the provider's perspective (section 5.5.3). Supporting data and processed data prevent consumers from implementing our business rules and making mistakes due to a lack of knowledge or incomplete data. Should the overdraft amount be added to or subtracted from the balance? And what if determining an accurate safe-to-spend amount requires considering known but unprocessed transactions?

8.4.4 Choosing well-known or standard resource identifiers

Resource identifiers, discussed in section 4.2.3, uniquely identify resources but also connect data from different systems and end users to systems. Choosing well-known or standard identifiers for API interoperability and intuitiveness is crucial whenever possible.

As shown in figure 8.10, we have three identifiers for a bank account: ID, account number, and IBAN (International Bank Account Number). The internal ID (`a686783e-d699-421b-965b-4f039d5c6adc`) is complex and only known by the subsystem that

manages bank accounts, so we can set it aside. The account number (3333333) is a better option. It is well-known across all of our systems and by end users. It is also easily remembered by developers in test environments. The IBAN (FR761111122222000033333320) is a longer but standard version of the account number and is understood worldwide.

BANK ACCOUNT	
DATA	VALUE
Identifier	a686783e-d699-421b-965b-4f039d5c6adc
	3333333
	FR761111122222000033333320

----- Internal ID

----- Well-known account number

----- Standard IBAN

Figure 8.10 The three possible identifiers for a bank account

We can use the well-known account number as a bank account identifier for internal use. It makes it easy for our subsystems to find relevant information. However, when external systems use our API, they may struggle to connect their data to ours if we don't adopt the IBAN standard identifiers.



NOTE I recommend selecting standard identifiers unless there are compelling reasons not to do so (such as security concerns; see section 12.1). Note that picking one as a resource identifier doesn't prevent us from having both the account number and the IBAN as regular data in our model.

8.4.5 *Choosing well-known or standard data*

Interoperability and intuitiveness are crucial for more than just resource identifiers. Each piece of data should be well-known or standardized whenever possible. For instance, the balance we chose from the possible balances in section 8.4.1 is well-known. Also, although we have user-friendly internal currency codes (EU for the Euro and UD for the US dollar) that are known by all internal systems, it is better to use internationally recognized ISO 4217 currency codes (EUR, USD) for the balance currency.



NOTE Identifiers and other data can be local, widely known across systems, internationally recognized, or a standard. The larger the audience, the better. However, no worries if standards don't exist; we can still achieve interoperability with shared information. Using only locally known data is fine if all else fails.

8.5 *Choosing user-friendly, interoperable atomic types and formats*

In section 5.1.2, we learned about the basic portable atomic data types: `string`, `number`, and `boolean`. Booleans are de facto user-friendly and interoperable as long as they are correctly named. On the other hand, strings and numbers can cause hiccups. As shown in figure 8.11 and discussed in this section, we need to

- Consider formatting numbers as strings
- Avoid non-human-readable code when possible
- Use human-readable date and time formats

DATA	NON-USER-FRIENDLY TYPE/FORMAT	ISSUE	USER-FRIENDLY TYPE/FORMAT
Balance	"€1,234.5"	Unnecessarily formatted numeric data	1234.5
Account number	1234567	Numeric code missing formatting	"0001234567"
Account type	1 or "xzt"	Cryptic code	"checking"
Creation date	409228260	Non-human-readable date	"1982-12-20"

Figure 8.11 Contrasting non-user-friendly and user-friendly data types and formats

8.5.1 Considering formatting numbers as strings

We should always carefully consider formatting numbers as strings. Numbers that consumers can use in calculations must not be formatted. However, numeric references or codes may benefit from string formatting.

Representing a bank account balance with a formatted string like "€1,234.5" may seem like a good idea; it avoids adding the balance currency to our model and makes it ready to use in a UI. But it's not a good idea. Consumers must parse it to extract the value for calculations and currency. Also, the chosen format may not be suitable for all end users; in France, a UI should show "1 234,5 €". Instead of such a formatted string, it's better to use a plain number (1234.5) and add a supporting currency in a separate property (section 8.4.2).



NOTE The consumer's locale typically defines how to format an amount of money or a date for the end user to read. It's usually up to the consumer to format raw data appropriately. However, an API may sometimes need to take the locale into consideration. Check section 9.7.2 for an example.

An account number like 1234567 is a number, but it needs leading zeros ("0001234567") to reach a certain length so that it's a proper account number. Remembering this length is challenging even for banking industry veterans like me. Thus, offering a formatted string instead of a bare number is helpful.

8.5.2 Managing non-human-readable codes

We should avoid using non-human-readable codes such as 1 or xyz in our API data. They make interpreting data complex, if not impossible, if you don't have a dictionary that explains them.

As shown in figure 8.12, in our system, a bank account type is a nonintuitive value like 1 or 7 (in another case, it could be "xzt" or "tvp"). These values are not self-explanatory and may be confusing to interpret. Another option could be abbreviated

codes such as `c` and `s`, which experts might guess stood for “checking” and “savings.” Clear labeling is essential for universal understanding. Thus, we should use plain English words like “checking” and “savings” instead of cryptic code values. Such human-readable alphabetical codes are typically what we would put in an enumeration when coding.

BANK ACCOUNT			
DATA	NON-HUMAN-READABLE		HUMAN-READABLE
Type	1	xbt	checking
	7	txp	savings

Figure 8.12 Contrasting non-human-readable and human-readable bank account type codes

In some cases, it’s impossible to replace non-human-readable codes with human-readable ones because they are used across multiple systems. To address this, we can apply what we learned from section 8.4.2 and add a human-readable label next to the code to help developers understand the data or show it to end users. For example, we can use `"typeLabel": "savings"` next to `"type": 7` or have a `type` object with a `code` and a `label`. Such labels may be shown to end users.



NOTE Dictionaries of code and values or labels can be retrieved via dedicated operations; see section 13.5.4.

8.5.3 *Managing dates and times*

API data uses timestamps (number) or the ISO 8601 standard (string) for dates and times. A bank transaction date, such as December 20, 1982, 10:31 a.m., can be shown as 409228260 (Unix timestamp) or 1982-12-20T10:31:00Z (ISO 8601). Any programming language supports both formats. However, I recommend ISO 8601 over timestamps for its readability, clarity, and flexibility in representing dates and times, with or without time and time zones.

An ISO 8601 string such as 1982-12-20T10:31:00Z (date-time string format in JSON Schema) offers a human-readable date and time format: YYYY-MM-DD for year, month, day; T as a time separator; HH:MM:SS for hour, minutes, seconds; and Z for the UTC time zone. If we just need a date without time precision, we can use 1982-12-20 (date string format in JSON Schema). In contrast, interpreting or modifying Unix time, which uses the number of seconds since the Unix epoch (January 1, 1970), is more challenging for humans. Developers can’t easily determine the date 409228260 from API logs, create a date input, or modify the day or hour for a test API call via an API client. It’s unclear whether a timestamp is a date or a date and time. As a result, I’ve seen many APIs that mix timestamps for date times and ISO 8601 for dates, which makes their design inconsistent.



CAUTION A timestamp’s units may vary depending on the context. Clarifying the units used is essential, as there’s a risk of inconsistency and confusion.

For example, Unix time and Python's `time.time()` are in seconds, whereas JavaScript's `Date.now()` is in milliseconds. `1704067200` can be `2024-12-31T00:00:00Z` (seconds) or `1970-01-20T17:01:07.200Z` (milliseconds).

An ISO 8601 date-time string can also indicate the time-zone offset: `1982-12-20T11:31:00+01:00` is the same date and time as `1982-12-20T10:31:00Z`, but in CET (Central European Time), which is one hour ahead of UTC (`+01:00`). Timestamps are UTC only and don't support time zones. However, sticking to the UTC time zone is recommended because it is not affected by daylight savings time; if the clock shift happens at 2:00 a.m., 1:30 a.m. could be before or after the shift. But if there are specific requirements, we can use time-zone offsets. For example, in our banking API, we may need it for audit or regulation reasons for bank account transaction dates.



TIP ISO 8601 also defines a format for durations. For instance, `P2Y3M5DT11H30M5S` represents 2 years, 3 months, 5 days, 11 hours, 30 minutes, and 5 seconds.

8.6 Organizing data

You would probably struggle to use a video game controller if its direction buttons were randomly scattered; it's the same with data. Developers will understand and use data easily if it's well organized. As contrasted in figure 8.13 and explained in this section, data is easier to find and process if it is

- Grouped
- Hierarchized
- Sorted

8.6.1 Grouping data with objects

Objects group data into smaller sets, highlighting the existence of subconcepts and their relations and making data easier to understand, use, and browse. The overdraft facility data in the Bank Account model data example in figure 8.13 is hard to find due to the separate overdraft facility Boolean flag and limit amount. The limit could be something unrelated to overdraft facility. Even experts might struggle to make the connection.

In figure 8.14, we rename `limit` to `overdraft facility limit` to make the relation between properties prominent. But it's still inefficient, as developers must review all the data to find both properties. Ordering the properties so they're close to each other can address this. However, the concept of an independent overdraft facility needs to be clarified. And what if there are 10 or more overdraft facility xxx properties? It will be better to group them under an overdraft facility object with an active flag (holding the original overdraft facility value) and its limit amount. This way, developers can easily view the business concept and data and benefit from this organization in their code. They can also open/close the object in a JSON viewer to visualize the data better.

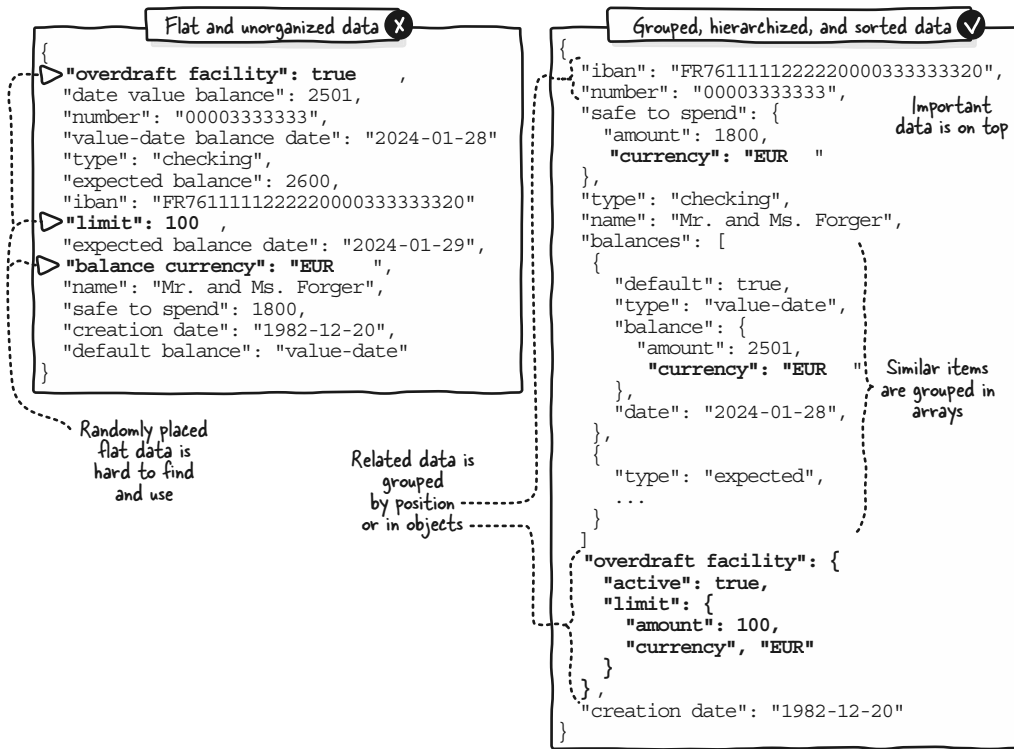


Figure 8.13 Organized data is easier to understand and use.

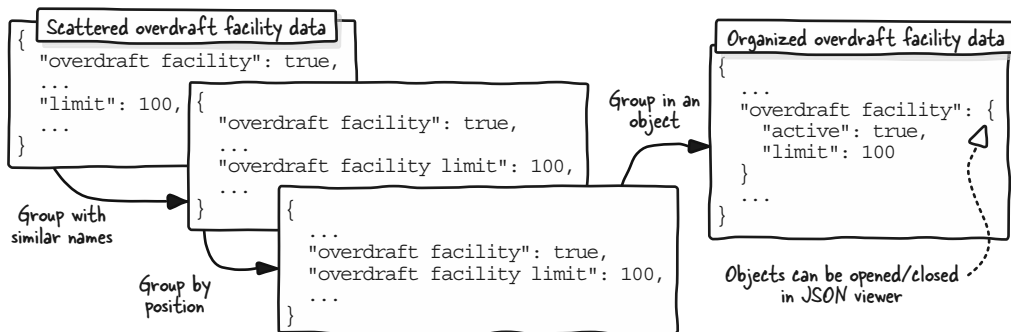


Figure 8.14 We can group data by name or position or in objects.

Business concepts can be more generic. The limit, value-date balance, expected balance, and safe to spend are all amounts in a specific currency indicated by the balance currency property. It's easy to connect a currency with the balances based on their names, but it's less evident for the other two. To fix this, we can turn each

numeric value into an object with `amount` and `currency` properties (see figure 8.13). Doing so allows developers to quickly understand and use each value independently of top-level data. We can use the same formats in different places, which makes our data more interoperable.



CAUTION Avoid over-organizing data into unnecessary sub-objects, as this can complicate interpretation and usability. Each sub-object should represent a business concept and ideally be reusable. Avoid mapping the UI organization (section 2.7.1).

8.6.2 Grouping data with arrays

Grouping similar objects into an array can simplify data browsing in code or a JSON viewer. It also helps materialize sub-business concepts, making them easier to understand.

As shown in figure 8.15, the Bank Account model has two balances described with different properties, and the `default` balance property indicates the default one. We can organize the balance data into value-date balance and expected balance object, as we learned in section 8.6.1. We also replace the `default-balance` string root property with a `default` Boolean flag in the value-date balance. Then, because they are both balances, we can group them in a `balances` array and add a `type` (`"value-date"` or `"expected"`). The fact that an account has multiple balances is more explicit and makes it easier for developers to view them or browse them in code.

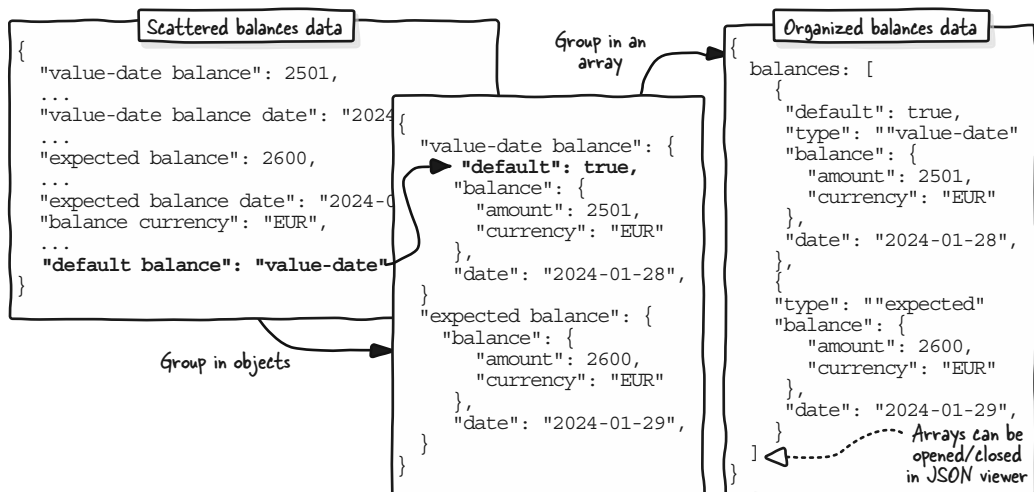


Figure 8.15 Different properties representing the same type of data may be grouped into an array.



TIP Use a `values` array to replace properties named `value1`, `value2`, ..., and `valueN`, such as replacing line1 to line4 with a `lines` array in an Address model.

8.6.3 *Sorting data in arrays and objects*

Sorting data in arrays and objects makes it easier to browse and manipulate. Sorting in an array can make data more straightforward for processing or rendering in a UI. However, we can't formally describe the array order in JSON Schema. Instead, we rely on the `description` field to specify the sorting for implementers (see section 19.5.1). For instance, we can state “sort the array by balance type: value-date and then expected” in the description of the `balances` array. This allows consumers to display the list without worrying about sorting it.

An object's property order doesn't affect code or applications but does affect viewed data and documentation. To make it easier for developers to find data, related data should be placed next to each other, and essential data should be placed first in the properties map of JSON schemas (and in implementations). For example, in the Bank Account model in Figure 8.13, the `iban` and `number` references are near each other and come first, followed by `safe to spend`.

8.7 *Choosing data granularity and scope*

After organizing the model data, we must ensure that we have only the necessary data and that we stay within the model's scope. A data model bloated with unnecessary elements or a mishmash of concepts that should live independently is not user-friendly. This section discusses

- Ensuring the relevance of each piece of data
- Embedding lists in resources
- Modeling embedded resources

8.7.1 *Considering relevance, not size*

We must question not the number of properties and depth of a data model but rather the relevance of each element. Smaller models are more user-friendly, but randomly removing properties to achieve size reduction can hinder meeting user requirements. Also, specialized smaller models lack versatility compared to larger, generic models. At this stage, model size and depth must be driven by subject matter, API capabilities, and context (section 13.5 covers size and performance). We can remove unnecessary elements to create user-friendly models by ensuring that we satisfy current and future user needs (see section 8.2.1). We do this when modeling data by focusing on the proper elements (section 5.5.3).

Suppose our Bank Account model has 30 properties and a maximum depth of 3. Although we can't compare it to other models, we know that it represents only a fraction of all bank account information and can be considered “relatively small.” But whether it has 20 or 100 properties and a depth of 3 or 10, we must investigate the purpose of each element, as illustrated in figure 8.16.

The `iban`, `number`, `type`, and `name` properties are essential to represent a bank account. The `safe to spend` property is essential to know how much money can be

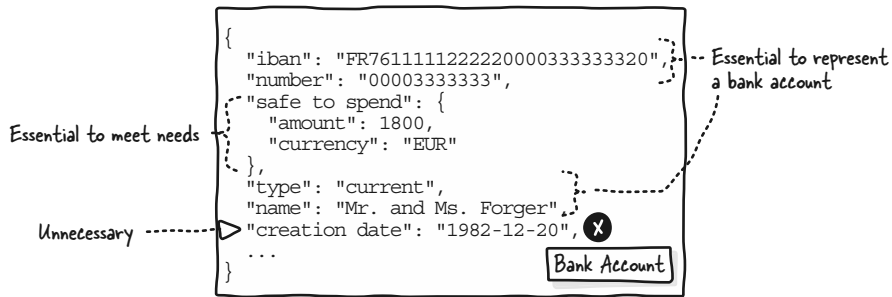


Figure 8.16 Each piece of data must be relevant.

spent and meet users' needs. However, according to our SMEs, creation date is unnecessary and not helpful in manipulating bank accounts beyond our initial need. We can simplify the model and reduce distraction by removing it.

8.7.2 Embedding lists in a resource model

When lists are embedded in a data model, we must consider their nature, relationship with the resource, and potential size, as illustrated in figure 8.17. Large arrays may necessitate separate resource models and operations for filtering and pagination so consumers can easily manipulate them (section 9.6).

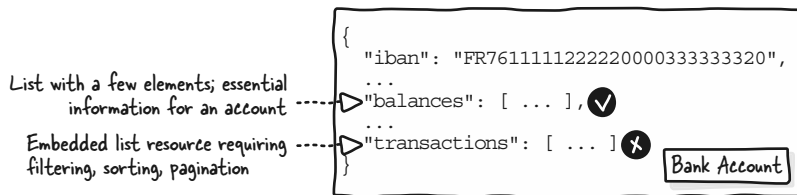


Figure 8.17 We shouldn't embed lists requiring filtering or pagination.

The balances array (see section 8.4.1) presents no problems because it's a small set comprising the current balance of each type, and it doesn't require filtering and pagination. We didn't identify balances as an actual resource, although we could do so depending on our needs. If we need historical account balance data via a dedicated operation, we could still keep this specific subset in the Bank Account model, as it's only the current balances, and a balance is essential information for an account. However, we may also reconsider the need for an array of balances and only keep the one value that makes the most sense based on our needs.

A list of all transactions directly embedded in the Bank Account model is more questionable. From a practical perspective, there can be thousands of transactions that require filtering and pagination. We should maintain the transactions separately

and create dedicated operations for ease of use. From a subject matter perspective, transactions are a standalone concept in banking. We may have missed something during the needs analysis if they have not been identified as resources. We could consider embedding a subset of the most recent transactions affecting the *safe to spend* value; however, such a model would look more like a *Safe to Spend Report* than a *Bank Account* model.

8.7.3 Modeling embedded resources

When embedding another resource (list or individual element) in the model of a resource, we should choose the appropriate model (ID, summarized, or complete; see section 5.4) depending on what's needed and to facilitate using data, as illustrated in figure 8.18.



CAUTION Providing insufficient information about related resources directly affects API efficiency and the end-user experience; see section 13.1.

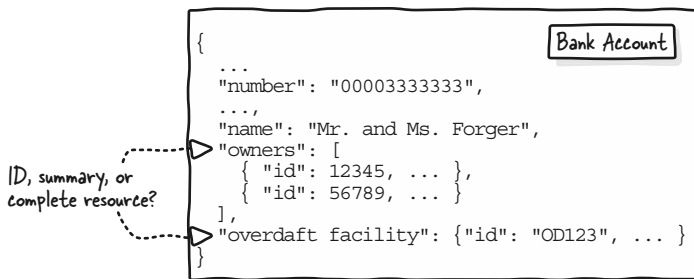


Figure 8.18 Adapt embedded resource models to needs, ensuring that consumers must not systematically read them.

Suppose that account owners and the overdraft facility are identified resources having their own life, and we need their data in the *Bank Account* model (owners and overdraft facility properties). An overdraft facility ID may be enough if we just need to know that a bank account is linked to an overdraft facility and we don't need the details when working on a bank account (thanks to the *safe to spend* value we added in section 8.4.3). Owner IDs won't be enough for owners because we need more information for bank advisors, such as owner names; consumers will systematically need to make additional API calls to read each owner after reading the account. A *Summarized Owner* model meets our needs.



NOTE To choose between using just an ID and a more complete model for embedded resources, evaluate whether consumers must read them systematically with the resource. If so, they likely form a cohesive whole. Embed at least a summarized model, or a complete model if more data is needed. If not, the resources are just related, and an ID is enough. A list of IDs may often be meaningless; include at least a summary, or consider removing the list.

8.8 Designing user-friendly names

Choosing names that everyone can easily read and understand is essential to creating a user-friendly API. Representing the overdraft facility active with the unreadable `ACTBLNDFPRTFTBKACC` or jargonesque `deficitProtectionActive` can make the API harder to understand and use. Designing user-friendly names is challenging and must be done at the right moment. This section discusses when to design user-friendly names and explains and illustrates the principles we can use to create names that are

- Simple
- Clearly organized
- Concise

8.8.1 When to design user-friendly names

To design our API efficiently, we must differentiate between identifying concepts with correct names and designing user-friendly names. This requires a deep understanding of data types, formats, and organization. Choosing user-friendly names should be our final task.

During the first design pass (section 5.1), our primary goal was to design an API that fulfilled user needs and correctly represented the subject matter. At this stage, when adding a property indicating that the overdraft facility is active to the Bank Account model, we must call it what we first named it: `overdraft facility active`. It's sufficient to identify this concept correctly from a subject matter perspective.

Discussing `overdraftFacility` versus `overdraftFacilityBln` versus `odFacActive` slows us down in achieving the first version of the design that does the job and that we can discuss with stakeholders. Additionally, we may need more information to make decisions. Names are affected by type, format, and context, which we explore thoroughly in this second pass of API design. For instance, the `overdraftFacilityBln` property may be not a Boolean but a string enumeration with an `active`, `suspended`, or `ended` value (see section 8.5). We can also rename it `active` when organizing the overdraft facility data in an object (section 8.6). Finally, it would be a shame to waste time on a piece of data we don't keep (section 8.7).

8.8.2 Designing simple, clearly organized, concise names

After agreeing on a concept and giving it a temporary name, we can represent it with a user-friendly name that is simple, clearly organized, and as short as possible. This section lists the principles we can use, and the next section illustrates them.

We must use simple language that everyone can understand, avoiding jargon. But at the same time, we must ensure that we don't sacrifice meaning for the sake of simplicity.

Names with multiple words benefit from proper casing and sorting to aid understanding. Casing examples include using camel (`someProperty`) or snake case (`some_property`) for property names and pascal case for reusable JSON Schema models in

OpenAPI (SomeModel). Also, similar to organizing paths (section 4.2), we must arrange words in a meaningful hierarchy from parent/broad/left to children/specific/right.

Aim for brevity, and consider the necessity of each word. Ideally, we should keep names to three words or fewer, but not at the expense of meaning (remember the number of properties from section 8.7.1). Context can help, and we can ditch prefixes and suffixes indicating the parent model, parent concept, or type. Also, if a word doesn't affect understanding, we can remove it. Avoid abbreviations whenever possible, but well-known abbreviations and acronyms are OK. We may discover grouping optimizations by refining names, even after designing a well-organized data structure (see section 8.6).

8.8.3 Learning by fixing non-user-friendly names

Figure 8.19 shows a poorly designed Bank Account data model sample. Property names like `ACTBLNDFPRTFTBKACC` and `LMTDFPRTFTBKACC` are hard to understand without proper documentation, even for experts. Let's use the principles listed in section 8.8.2 to fix them.

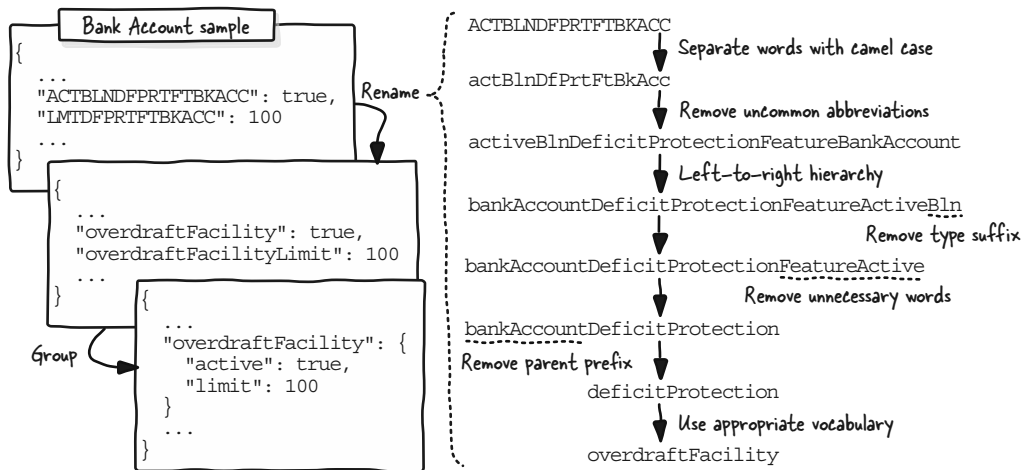


Figure 8.19 Fixing non-user-friendly names in the Bank Account model

Reading `ACTBLNDFPRTFTBKACC` is complicated. We can use camel case, `actBlnDfPrtFtBkAcc`, which helps by showing distinct words, but the words are abbreviated and not meaningful. Replacing our specific abbreviations but keeping commonly used ones, such as `bln` for boolean, gives a more readable `activeBlnDeficitProtectionFeatureBankAccount`. Unfortunately, understanding is not facilitated by words that are randomly ordered without hierarchy. Sorting them from parent to child gives `bankAccountDeficitProtectionFeatureActiveBln`.

Reading such a long name is difficult; we must shorten it. The property's true value is a Boolean, so the `Bln` suffix is unnecessary. `Active` is redundant with the Boolean type; we can also remove it. We can also challenge `Feature`; the name is still understandable without it. The same goes for the `bankAccount` prefix; there's no need to state that we're in a "Bank Account model."

That leaves `deficitProtection`, our jargon for what most people call an "overdraft facility." Using the more familiar term, we end with `"overdraftFacility": true`, which clearly indicates whether the feature is active on the bank account.

We transform `LMTDFPRTFTBKACC` into `overdraftFacilityLimit` with the same reasoning. Both properties have a prefix representing the same business concept. As seen in section 8.6.1, we can group them in an `overdraftFacility` object property with `active` (reusing the word we got rid of earlier) and `limit` properties.



NOTE Casing preferences vary. We used camel case (`overdraftFacility`), which is typical of Java and JavaScript, but we could also have used snake case (`overdraft_facility`), which is typical of Python. Consumers can easily adapt, so choose based on your preference, but be consistent (section 8.9).

8.9 Aiming for consistency and standardization

To ensure that our API is user-friendly and interoperable, we must make consistent decisions and aim for standardization. Data that looks like other data or matches standards favors intuitiveness and interoperability. This is tricky because it applies at any level: which data we select; its organization, types, formats, and names; and the standards we use. Every decision we or others have made previously may affect our and others' API design now or later. For example, if the first date we model is named `whateverDate` and uses ISO 8601, we better stick to this format for any date property we model afterward. This section illustrates these concerns by revisiting elements of our Banking API example in light of them.



NOTE Being consistent and creating or using the proper standards can be difficult. Check out section 16.1 to discover how to overcome this.

8.9.1 Seeking local, domain, or global standardization

Data must at least be consistent within the API. But it's best to be consistent with other APIs in the organization, domain or industry, and even the rest of the world. Seek consistency at the most extensive possible spectrum, but no worries if that's only local consistency. Check whether a ready-to-use standard exists, reuse preexisting data models as is, or adapt them. If no local or broader standard is applicable, reuse the same patterns when choosing types, formats, organizations, and names.

Before designing data models for our Banking API, we didn't look for preexisting material. Our team or organization may already have designed a Bank Account model for other purposes, in which case we could reuse it. Also, the banking industry's renowned ISO 20022 standard offers data models for several banking use cases; we

should investigate it. However, in both cases, we must be sure this preexisting material actually fits our needs. Being interoperable but not fulfilling user needs is not an option. If nothing exists, we can start inventing our own model, but we can also use preexisting patterns or standards in our data models.



CAUTION Be careful when using organization or standard data models. Ensure that they fulfill your requirements. Also carefully consider their complexity; standards are not always designed to be user-friendly for non-experts.

8.9.2 Using well-known or standard identifiers consistently

We saw in section 8.4.4 that we should use well-known or standard identifiers. We must be sure to use the same identifier when the same resource is used in various contexts. Figure 8.20 contrasts options for identifying a bank account in different models.

MODEL	CONCEPT	DATA		Consistent and interoperable
		Inconsistent	Consistent	
Bank Account	identifier	IBAN	account ID	IBAN
Money Transfer	source	account ID		
	destination	account number		

Figure 8.20 We must use the same identifier for the account across models.

If we choose the IBAN for the Bank Account model and ID and number, respectively, for a Money Transfer source and destination, we're inconsistent within and across models. Choosing the ID for all uses makes our design consistent, but the ID is only known in a small part of our system. Choosing the IBAN is a better option that makes our design consistent and interoperable with any banking and, most likely, non-banking APIs worldwide.

A money transfer needs an ID, too. There's no standard for that. If there's a well-known identifier shared across our system, we can use it. If not, no worries; not all identifiers have to be well-known or standard, but always check if any are available.

8.9.3 Defining a naming pattern for identifiers

It's essential to define a naming pattern that helps developers identify the identifiers of resources and whether there are related resources. As shown in figure 8.21, naming the account identifier `iban` implies that it's an IBAN but not necessarily the bank account's unique identifier. But using a generic name such as `accountId` causes a problem, too, because the IBAN is essential data. We keep both `iban` and `accountId` to strike a balance. However, adding `accountOwnerId` may complicate the resource identifier search. Therefore, we opt for the even more generic name `id` to resolve the problem.

These decisions have consequences. Any resource identifier must be named `id` within the resource and `resourceNameId` elsewhere. If this identifier is essential and meaningful data, it may be present twice within the resource as the `id` and another

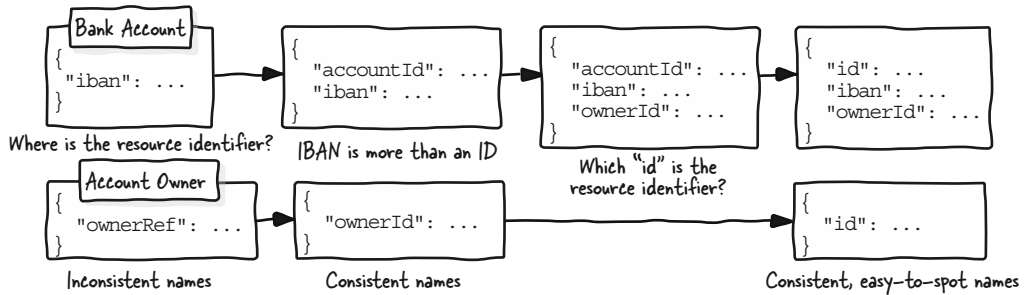


Figure 8.21 Inconsistent versus consistent and user-friendly resource identifier names

property. For instance, the “Account Owner” resource has an `id` whose value is the owner reference, but no `ownerReference`. This information is only interesting as the owner’s unique identifier.

8.9.4 Naming, typing, and structuring consistently

As shown in figure 8.22, consistency and standardization are a concern for any aspect of data, not just resource identifiers. The account number needs to be formatted with leading zeros (see section 8.5.1), and we must format it this way everywhere. The bank account creation date is an ISO 8601 string (see section 8.5.3) named `created`, whereas the execution date of a money transfer is an `executionDate` UNIX time. We must use the same type and naming pattern to be interoperable and consistent. The account balance is an object with a `value` and an ISO 4217 currency property (see section 8.6.1); the money transfer’s amount must be structured similarly.

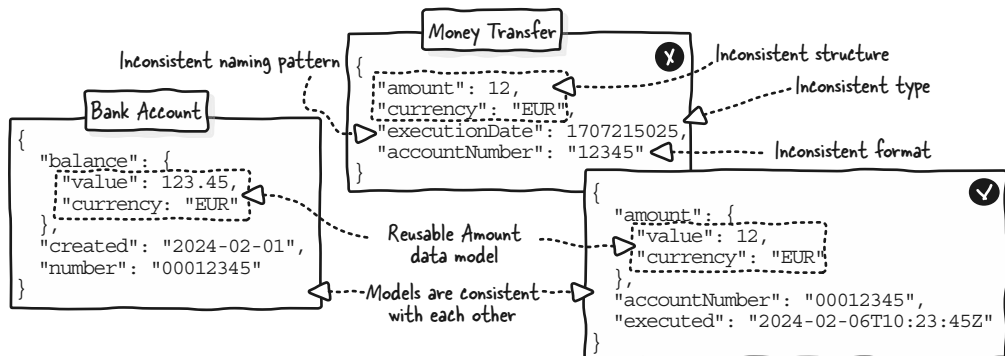


Figure 8.22 Bank Account model and inconsistent versus consistent and interoperable Money Transfer models



CAUTION When a decision is made, it creates a precedent that establishes a pattern to be applied consistently in the future. Building design guidelines helps you design consistently; see section 16.3.

Summary

- Besides fulfilling their needs, users expect an API to be user-friendly: easy to understand and enjoyable to use.
- Design plays a crucial role in attracting or repelling users, making their work easier or more complex. It can even be a source of joy.
- To facilitate API usage, data must be user-friendly and simple, clear, helpful, intuitive, and interoperable regardless of the context, whether that is a resource path, path parameter, query parameter, header, request body, or success or error response body.
- For an efficient design process, treat user-friendliness via a distinct second data-modeling pass after ensuring that the identified data fulfills user needs.
- Ensure that the data is ready to use. Select simple yet effective data, enhance it with supporting or processed data, and use well-known or standard data.
- Don't format numbers that can be used in calculations as strings; consumers would have to parse them. However, numeric references or codes may benefit from string formatting.
- Avoid using non-human-readable codes such as 1 and xyz in API data whenever possible; they are challenging to understand. If using them is unavoidable, add human-readable labels.
- Use the ISO 8601 standard to represent dates and times to facilitate readability. Use its time precision only when necessary to prevent dealing with time-zone complexity.
- Organized data is easier to understand and process. Group properties that are shared by concepts in objects (typically, `conceptThis` and `conceptThat`), and group similar elements in arrays (commonly, `item1` to `itemN`). Sort arrays for easier manipulation, and sort object properties to make it easier to find information.
- Smaller data models are easier to understand and process, but don't question the number of properties and structure depth; rather, consider their relevance.
- Consider using a resource identifier or a summarized or complete model when embedding a resource into another, to provide sufficient but useful data.
- Avoid embedding large arrays into a resource. They require search and pagination features that are only accessible via dedicated operations.
- For an efficient design process, identify concepts during the first design pass and choose user-friendly names only after typing, formatting, and organizing the data.
- To design easy-to-understand names, use simple language casing to separate words; arrange words hierarchically; remove parent prefixes, type suffixes, and unnecessary words; and aim for three words max (but not at the expense of meaning).
- Check whether standard or preexisting models can be used; doing so fosters interoperability.

- Seek consistency and interoperability across a broad range, ideally aligning with global standards or industry practices, while carefully considering complexity and ensuring that you meet all requirements. Local-only consistency can also be sufficient.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 8.1

Listing 8.1 shows partial flight resource data used by a flight comparator. Based on this chapter's teachings, how could this data be completed or improved to be user-friendly and interoperable? Hint: use an actual flight comparator as a virtual SME.

Listing 8.1 Flight data

```
{
  ...
  flight: {
    "number": "AF1234",
    "departureAirport": 12345,
    "departureTime": 1733063400,
    "arrivalAirport": 67890,
    "endTime": 1733073000,
  },
  "class": 56,
  "price": {
    "base": 150,
    "taxes": "50.00",
    "discount": 0.1
  }
}
```

Exercise 8.2

Listing 8.2 shows an artist's complete resource data for an audio streaming service. Are there any elements that should be removed, and why?

Listing 8.2 Artist data

```
{
  "id": 12345,
  "name": "Maaya Sakamoto",
  "bio": "Maaya Sakamoto started ...",
  "genres": ["anime", "j-pop"],
  "albums" : [
    { "id": 54, "name": "Grapefruit", ... }
  ],
}
```

```
"tracks": [  
  { "id": 54, "name": "Feel Myself", ... }  
]
```

Exercise 8.3

Listing 8.3 shows the data for a “Book” resource in an API for a library. List the problems in naming and organization that make this data non-user-friendly, and fix the data based on what you learned in this chapter.

Listing 8.3 Book data

```
{  
  "bookReference": "B12345",  
  "title": "The Eternal Champion",  
  "authorCode": "A123",  
  "name": "Michael Moorcock",  
  "published_year": 1970,  
  "authDob": "1939-12-18",  
  "ctry": "GBR",  
  "genre": "Fantasy"  
}
```

Designing user-friendly, interoperable operations

This chapter covers

- Designing easy-to-use requests and responses
- Filtering, sorting, and paginating lists
- Handling multiple data formats
- Erroring gracefully
- Avoiding hiding capabilities
- Standardizing operations

Imagine a Shopping API whose “Create order” (POST /orders) operation returns 400 Bad Request without further explanation for any error, such as a typo in a property name (`qantity` instead of `quantity`), an invalid product reference, or an unavailable product. Developers will have difficulty figuring out the problem in their code, or end users may face an unhelpful “Impossible to validate order” message.

And it’s not only error-handling that can be problematic. Despite a valid purpose, an operation’s HTTP representation, requested and returned data, and behavior can increase development time, code complexity, and error risk, and can even affect end users. Designing user-friendly, interoperable operations prevents these problems, making developers more efficient and leading them to love the API.

This chapter provides an overview of what makes an operation user-friendly and interoperable, and when and how to take these aspects into consideration. Then we explain how to design easy-to-understand and guessable operations, request easy-to-provide inputs, and return ready-to-use, successful outputs. We show how to design flexible operations that adapt data to consumer needs. Subsequently, the chapter dives into error-handling, illustrates how a design can hide capabilities, and discusses how to aim for consistency and standardization.

9.1 ***What makes operations user-friendly and interoperable?***

API operations must be user-friendly and interoperable to enable efficient development and ensure an excellent end-user experience. This section illustrates how operations can be user-friendly and interoperable by

- Meeting user needs
- Exposing clear capabilities
- Using user-friendly data
- Being helpful
- Being consistent
- Being standard



NOTE User-friendliness and interoperability matter for public, partner, and private APIs (section 8.1).

9.1.1 ***User-friendly operations expose clear capabilities that meet the needs***

As for data, meeting user needs fosters the design of user-friendly operations that consumers will easily understand. Suppose we create a car rental API (as in section 8.2) dealing with car rental shops, the cars they own, and renting them. Identifying capabilities that meet user needs will likely end with “Rent a car” and not “Implement a new deal,” an unclear jargon that doesn’t fit in this context. However, beyond meeting user needs, capabilities must be clearly visible through the API’s operations. For example, although it’s possible to add a car to a rental shop by updating the shop, we likely need a dedicated “Add a car to a shop” operation.

9.1.2 ***User-friendly operations use user-friendly data and are helpful***

To be user-friendly, an operation must request and return data that is also user-friendly (section 8.2). We must apply what we learned to all the operation data, including resource paths, path parameters, request headers, query parameters, request bodies, response headers, and response bodies. If our initial HTTP representation of “Search cars” is `GET /shops/{shopId}/cars`, we may wonder if we can simplify it to `GET /cars`, which requests no parameters. However, that brings us back to the needs analysis: “Should we search cars across all shops?” How we organize data in requests is also

essential. For example, “Rent a car” input data must be in the request body and not randomly split between the body, headers, and query parameters.

Operation behavior and data must be helpful and facilitate consumers’ work. “Search cars” should only return available cars by default and should be able to filter on a specific type of vehicle. If `startDate` and `endDate` are missing when calling “Rent a car,” the error feedback should explicitly state that these two required properties are missing instead of returning an empty 400 Bad Request HTTP status without any other information, which would likely affect the end user’s experience (section 8.1.3). Once the rented car is used, if the distance is tracked, indicating the remaining distance (in miles or kilometers) allowed by the contract when using “Get car rental information” would be an excellent addition.

9.1.3 User-friendly, interoperable operations are consistent and standard

Just as with data (section 8.9), consistency and standardization make operations user-friendly and interoperable. Developers can guess which operations exist and how they behave based on their knowledge of the subject matter and HTTP and their experience with this or other APIs. If `GET /shops` returns the list of car rental shops, users can guess that `POST /shops` allows for creating a new shop and `GET /shops/{shopId}` returns all shop details. A nonintuitive design might include `GET /shops` (plural resource name) and `GET /shop/{shopId}` (singular resource name); developers couldn’t build the second resource path from the first. With consistent requests and responses, developers can guess which data is needed to rent a car by looking at the response “Get car rental information.”

Using well-known or standard elements and behaviors is key at the operation level. If the API is private, developers expect the car ID used in the `/cars/{carId}` path not only to be used across the API’s operations but also to be the well-known ID shared between all internal systems. Developers also expect the API to comply with HTTP; for instance, a `GET /cars/1234` must read a car, not delete it, and return a 404 Not Found HTTP status if the car 1234 doesn’t exist.



NOTE Consistency and standardization make APIs highly intuitive, simplify coding, and guarantee an invaluable “wow” effect. Developers will feel like they have superpowers.

9.2 When and how to design user-friendly, interoperable operations

Now that we’ve seen what makes operations user-friendly and interoperable, we can discuss when to take these concerns into consideration and how to design such operations.

9.2.1 When to take user-friendly, interoperable operations into consideration

As shown in figure 9.1, we’re still working on the user-friendly, interoperable layer, a second pass through our initial versatile API design that does the job. Don’t waste time wondering whether `GET /shops/{shopId}/cars` can be optimized into `GET /cars` when first representing the “Search cars” operation with HTTP; keep it for this second pass. We can simultaneously work on data and operation user-friendliness and interoperability because they are intimately linked. As we said for data in section 8.3, there will probably be some back and forth between meeting user needs and being user-friendly and interoperable, as with the `GET /cars` optimization (section 9.1.2).

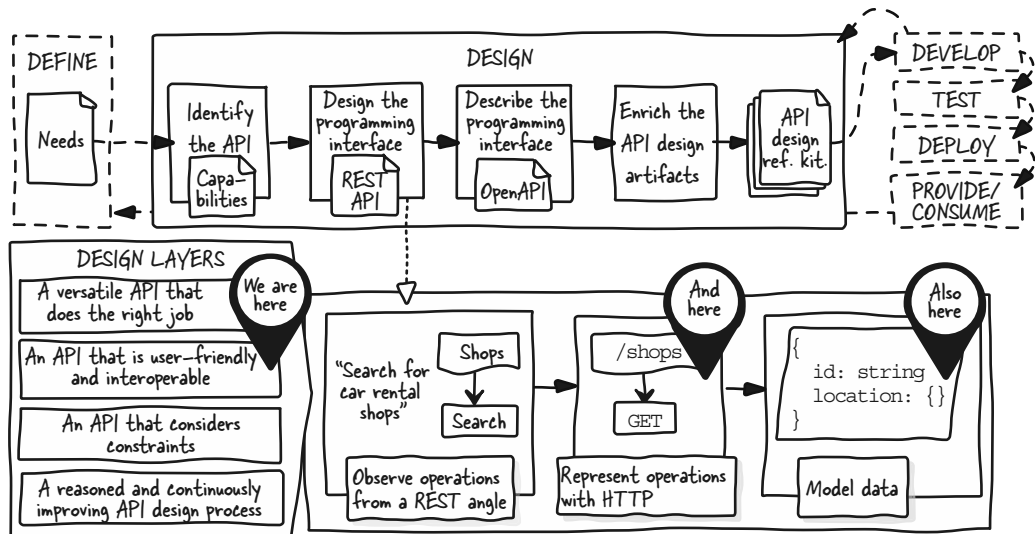


Figure 9.1 Once we have designed versatile operations that do the job, we can revisit them to ensure that they’re user-friendly and interoperable. We’ll rework the HTTP representations and input and output data modeling.



NOTE The initial operations design will require less and less rework to become user-friendly and interoperable, thanks to experience and the help of API design guidelines that we’ll craft to facilitate our work (section 16.3).

9.2.2 How to design user-friendly, interoperable operations

Although identifying API capabilities gave us a solid foundation to create user-friendly operations by meeting user needs (“Rent a car” versus “Implement a new deal”), we need to go further. Figure 9.2 summarizes how to create user-friendly operations based on what we saw in section 9.1.

We craft easy-to-understand and guessable operations by working on paths and HTTP methods (`GET /shops` and `GET /shops/{shopId}` or `GET /shop/{shopId}`). We

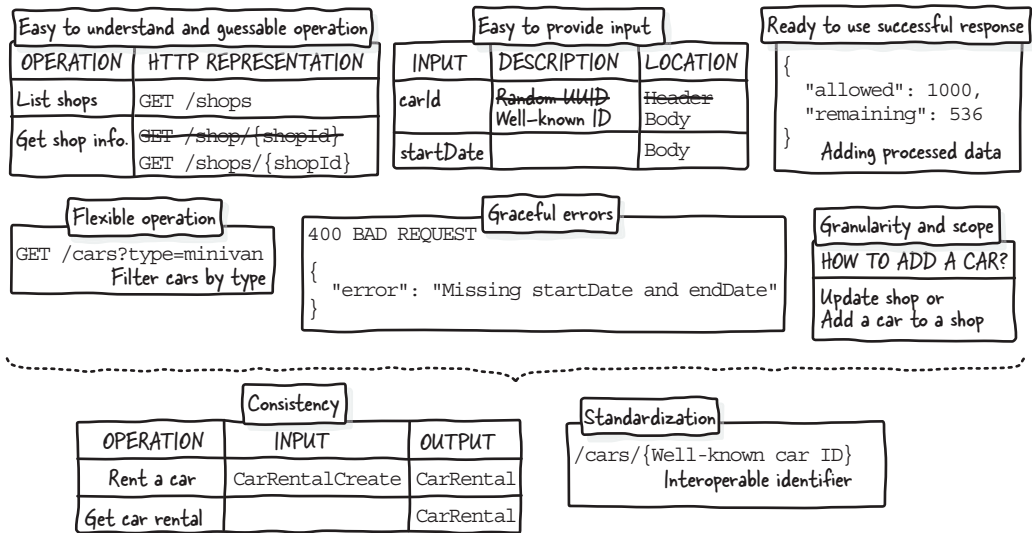


Figure 9.2 Consistent, interoperable, user-friendly operations with appropriate granularity, HTTP representations, input, and output and helpful error-handling are easily used by anyone and any system.

ensure that the inputs are easy to provide (“Rent a car” input location, well-known car’s ID). We also ensure that successful responses are ready to use (“Remaining allowed distance”). We enhance operations with features to make them flexible (filtering on a specific type when searching cars). We handle errors gracefully and help consumers fix them (indicating that the required `startDate` and `endDate` are missing when renting a car). We reconsider operation granularity and scope to ensure clear capabilities (adding a car via the shop’s update or a dedicated operation). At all times, we ensure that we are making consistent design decisions and aim for standardization (similar data in “Rent a car” and “Get car rental information” and well-known car ID). The rest of this chapter dives into all these concerns using the Banking API introduced in the previous chapter.



NOTE This chapter focuses on user-friendly, interoperable operations. However, operation design must also address meeting user needs (section 5.1), performance (section 13.1), security (section 12.1), and implementation constraints (section 14.1).

9.3 Designing easy-to-understand, guessable operations

User-friendly operations should be immediately understandable based on their path and HTTP method and should even be guessable without being seen. We can achieve this by

- Combining meaningful resource paths and HTTP compliance
- Creating predictable resource paths
- Crafting short but accurate resource paths

9.3.1 Combining meaningful resource paths and HTTP compliance

We have learned to design meaningful paths and use appropriate HTTP methods when representing operations with HTTP (section 4.1). This is a good start to make operations easy to understand and guessable. A path like `/accounts` is easily understood as the “list of accounts,” whereas a less user-friendly `/act` path can be confusing. Users can also easily understand the `/accounts/{accountId}/transactions` hierarchical path as “account’s transactions.” Thanks to the appropriate use of standard HTTP methods, if the `/customers` path represents the bank’s customers, users can guess that `GET /customers` allows them to “List or search for customers” and `POST /customers` means “Add or create a customer.” Although meaningful and readable, options like `POST /add-customer` or `POST /delete-customers` are less guessable.



CAUTION Although there are other HTTP methods (see www.iana.org/assignments/http-methods/http-methods.xhtml), only use POST, GET, PUT, PATCH, and DELETE to avoid surprising developers, their libraries, and their HTTP intermediaries with obscure methods (we’ll add OPTIONS to the list in section 11.4.1). Also, it’s essential to respect the meaning and inherent behavior of these HTTP methods; don’t modify data with a GET, for example. The implementation must comply with the methods’ “idempotent” or “safe” nature. Don’t worry if these terms are unfamiliar; we’ll discuss these concerns in section 12.5.2.

9.3.2 Creating predictable resource paths

A path must be hierarchical and strictly organized to be predictable. Each level must have a specific purpose.

Resource paths must behave like hierarchical file system paths. Using `/accounts` (plural) and `/account/{accountId}` (singular) doesn’t make any sense. A parent directory doesn’t change when we access one of its files. Users intuitively append an account identifier to `/accounts` to read a specific account (`GET /accounts/{accountId}`). Similarly, users who know that an account has transactions append `transactions` to the path to call `GET /accounts/{accountId}/transactions`. This works only when the child has its parent as root (`/path/to/parent` and `/path/to/parent/child`). As seen in section 4.2.5, using singular resource nouns is OK; just be consistent and use singular everywhere (`/account`, `/account/{accountId}`, `/account/{accountId}/transaction`).

Avoid randomly structured paths; each path segment must have an actual purpose. Although readable and hierarchical, paths such as `/savings/accounts/{accountId}` and `/checking/accounts/{accountId}` are puzzling, especially when compared to `/customers/{customerId}`. Users won’t easily guess such paths because of the `savings` and `checking` segments. Additionally, their role is unclear; are these actual resources, identifiers, or something else? The “Resource list plus optional resource identifier” pattern can fix this (`/resources`, `/resources/{identifier}`, `/resources/{identifier}/sub-resources`, `/resources/{identifier}/sub-resources/{sub-identifier}`...). To make these paths more predictable and to clarify them, we can use `/savings-accounts/{accountId}` and `/checking-accounts/{accountId}`. We can also challenge

the distinction between the types of accounts and use `/accounts/{accountId}` if it is unnecessary.

9.3.3 Crafting short but accurate resource paths

A hierarchical path is relatively easy to read because the last segment drives its meaning, but short resource paths are better for readability. Using globally unique resource IDs helps reduce path length with the nice side effect of minimizing input data (discussed in section 9.4.4 for all inputs). However, we must ensure that we do not create inaccurate paths.

Figure 9.3 shows the paths we may design to represent customers, addresses, accounts, and transactions. We shorten them by analyzing each path parameter starting from the end of the path (grayed segments in the figure).

RESOURCE	SHORTENED PATH
Customers	<code>/customers</code>
Customer	<code>/customers/{customerId}</code>
(Customer's) accounts	<code>/customers/{customerId}/accounts</code> (Only if "all accounts" and not "customer's accounts")
Account	<code>/customers/{customerId}/accounts/{accountId}</code> ◀----- Globally unique IDs-----
Account's transactions	<code>/customers/{customerId}/accounts/{accountId}/transactions</code> ↘-----
Transaction	<code>/customers/{customerId}/accounts/{accountId}/transactions/{transactionId}</code>
Customer's addresses	<code>/customers/{customerId}/addresses</code>
Customer's address	<code>/customers/{customerId}/addresses/{addressId}</code> ◀----- Non-globally unique ID-----

Figure 9.3 We question user needs and use globally unique IDs to reduce the number of path parameters in the Banking API resources paths and make them easier to use.

The `/customers/{customerId}/accounts/{accountId}/transactions/{transactionId}` path represents a transaction. We can shorten this path to `/transactions/{transactionId}` because a transaction is identified by a globally unique ID in our banking system, making all other path parameters unnecessary.

The `/customers/{customerId}/addresses/{addressId}` path representing a customer's address can't be shortened because `addressId` is not globally unique, with values such as `home` or `fiscal`. Only the combination of `customerId` and `addressId` can uniquely identify a customer's address. Similarly, `/customers/{customerId}` (a customer) can't be shortened.

We can proceed similarly with paths not ending with a path parameter. In the `/customers/{customerId}/accounts/{accountId}/transactions` path representing an account's transaction, `accountId` is necessary to identify the account. But it's a globally unique ID, so `customerId` is unnecessary. This simplification applies to the parent account resource path, with `/customers/{customerId}/accounts/{accountId}` becoming `/accounts/{accountId}`.

We haven't modified `/customers` and `/customers/{customerId}/addresses` because they're the best ways to represent the list of customers and a customer's addresses. If the `/customers/{customerId}/accounts` path represents the accounts a customer holds, we can't shorten it. But if it represents all customers' accounts, we can shorten it to `/accounts`. An API can have both options depending on the needs it serves.



NOTE Shorter resource paths are also more versatile. Customers and bank branches may have accounts accessible with `/customers/{customerId}/accounts` and `/branches/{branchId}/accounts`, but `/accounts/{accountId}` will represent one account in both contexts (as long as they actually are the same resource).

9.4 Requesting easy-to-provide inputs

We must ensure that users can easily provide the operation input data, including path parameters, query parameters, request headers, and request bodies. In addition to designing user-friendly input data (section 8.3.3), we must ensure

- Using typical and HTTP-compliant input locations
- Mapping inputs to outputs
- Requesting well-known or standard data
- Minimizing required input data

9.4.1 Using typical and HTTP-compliant input locations

Complying with HTTP standards and habits when choosing input locations creates easy-to-provide inputs. This section revisits the principles learned in section 4.4 in light of user-friendliness. We explain how to choose a data location in an HTTP request (path parameters, query parameters, headers, or body; see figure 9.4) and the bad consequences of choosing randomly or using only one location for all data.

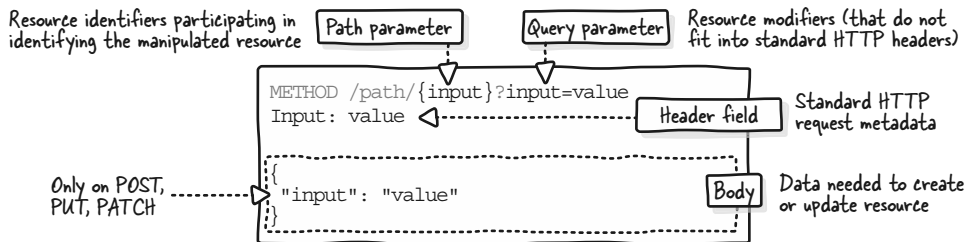


Figure 9.4 Choosing user-friendly input data locations in an HTTP request depends on HTTP and common practice. Choosing appropriate locations facilitates the provision and understanding of input data.

As shown in figure 9.5, a basic money transfer (`POST /transfers`) needs source and destination accounts and an amount of money. Randomly selecting the location of each input complicates the inputs and makes their location unguessable. Putting all

data in query or path parameters or headers (POST /transfers/12345/54321/100/EUR, for example) makes it difficult to discern the structure of the transfer resource data. Additionally, we must put the resource data in the body to be HTTP compliant.

APPROPRIATE LOCATIONS ✓	RANDOM LOCATIONS ✗	HEADERS ONLY ✗
<pre>POST /transfers { "source":12345, "destination": "54321", "amount": { "value": 100, "currency": "EUR", } }</pre>	<pre>POST /transfers?source=12345 Destination: 54321 { "amount": { "value": 100, "currency": "EUR", } }</pre> <p>Hard to understand, not guessable, and not HTTP compliant</p>	<pre>POST /transfers Source: 12345 Destination: 54321 Amount-Value: 100 Amount-Currency: EUR</pre> <p>Non-standard HTTP headers, not HTTP compliant, and data structure is not easily visible</p>
PATH PARAMETERS ONLY ✗ Not HTTP compliant (resource not in body), and hides data structure		
<pre>POST /transfers/source/12345/destination/54321/amount.value/100/amount.currency/EUR POST /transfers/12345/54321/100/EUR</pre> <p>Non-hierarchical data in path is not HTTP compliant, cryptic with stacked parameters, hard to implement and use (how to handle optional values and unordered values?)</p>		
QUERY PARAMETERS ONLY ✗ Not HTTP compliant (resource not in body), and hides data structure		
<pre>POST /transfers?source=12345&destination=54321&amount.value=100&amount.currency=EUR</pre>		

Figure 9.5 Creating a money transfer requires input data to be put in the body to be HTTP compliant and user-friendly.

The “Search account’s transactions” operation in figure 9.6 needs an account identifier and optional search filters like `from` (a date) and `minAmount`. Using only path parameters goes against HTTP recommendations by having a path with nonhierarchical data (`/accounts/12345/transactions/from/2024-01-01/minAmount/100`). Stacked parameters (`/accounts/12345/transactions/2024-01-01/100`) make the path less understandable, especially with unsorted or absent parameters (`/accounts/12345/transactions/100/2024-01-01`, `/accounts/12345/transactions/100`).

Using only query parameters can remove helpful hierarchical information from resource paths (`/accounts/12345/transactions` versus `/transactions`). Although `/accounts/12345/transactions` and `/transactions?accountId=12345` are both valid resource identifiers for HTTP, many developers consider only the path (excluding query parameters) to identify resources. This separation allows for unambiguous identification of resource and modification parameters. Additionally, inconsistencies can arise that confuse users. For example, `GET /transactions?accountId=12345` returns a list of transactions, whereas `GET /transactions?transactionId=2501` returns a single transaction.

Relying solely on headers can complicate developers’ work. It can hide significant data in logs and API clients, such as `accountId`. It also leads to having headers that are not HTTP request metadata. Using only the body leads to using only POST, because GET and DELETE can’t have a body. In the process, we lose the resources as business

APPROPRIATE LOCATIONS ✓			
GET /accounts/12345/transactions?from=2024-01-01&minAmount=100			
PATH PARAMETERS ONLY ✗			
GET /accounts/12345/transactions/from/2024-01-01/minAmount/100 GET /accounts/12345/transactions/2024-01-01/100 GET /accounts/12345/transactions/100/2024-01-01 GET /accounts/12345/transactions/100 GET /accounts/12345/transactions/2024-01-01			
Non-hierarchical data in path is not HTTP compliant; cryptic with stacked parameters, hard to implement and use (how to handle optional values and unordered values?)			
QUERY PARAMETERS ONLY ✗			
Hides the necessary resource identifiers and againsts habits			
GET /accounts/transactions?accountId=12345&from=2024-01-01&minAmount=100 GET /transactions?accountId=12345&from=2024-01-01&minAmount=100 No hierarchy, and returns many elements while GET /transaction?transactionId=2501 returns one element			
BODY ONLY ✗		HEADERS ONLY ✗	
We lose resources, and the predictable HTTP uniform interface	POST /search-transactions { "accountId": 12345, "from": "2024-01-01", "minAmount": 100, }	Headers hidden in logs and API clients	GET /transactions Account-Id: 12345 From: 2024-01-01 Min-Amount: 100

Figure 9.6 Using a path parameter for the account identifier and query parameters for the transaction resource modifiers makes the “Search account transactions” operation user-friendly.

concepts (/search-transaction is an operation) and the predictable HTTP uniform interface, making the API harder to understand, guess, and use.

9.4.2 Mapping inputs to outputs

When modeling data, we have learned to design inputs from the complete resource models and hence the outputs (section 5.3). This makes the inputs guessable based on the outputs and makes them easy to provide. For instance, the AccountCreation model maps the AccountModel. Once developers have read an account, they can guess what data to provide when creating an account. Similarly, once they have seen that the data returned by “Search for transactions” contains a currency property, they can try GET /transactions?currency=USD.

9.4.3 Requesting well-known or standard data

Using well-known or standard data for path parameters, query parameters, headers, and request bodies makes operations more straightforward. The /accounts/{iban} path is user-friendly and interoperable because IBAN is a standard for identifying bank accounts. Using the well-known customerId shared among all internal systems for /customers/{customerId} is a valid option when no standard exists.

IDs are often opaque, but not always. As we saw in section 8.5.2, we can sometimes use fixed codes that I also call *magic identifiers*. For instance, a customer’s address can be

represented with the `/customers/{customerId}/addresses/{addressId}` path, where `customerId` is likely an opaque identifier. In contrast, the `addressId` can be a fixed identifier like `home` or `fiscal`, given that a customer can have only one address of each type at a time. Thus, the path for the 1234 customer's home address is `/customers/1234/addresses/home` (`/customers/{customerId}/addresses/{addressType}`).

Using well-known or standard data makes it easy to provide, wherever it is located in the request. For example, we can use an ISO currency code as a currency query parameter to filter transactions. In the request body of a money transfer, we can use IBANs as source and destination account numbers. Similarly, we can use the well-known `customerId` in the body of “Add owner to account.”

9.4.4 Minimizing inputs with default and server-processed data

Minimizing input data facilitates the operation's use, as we learned with path parameters in section 9.3.3. This section demonstrates this principle by removing the element of a request body that the implementation can deduce and making other elements optional with appropriate default values that the implementation will set. This also applies to query parameters (discussed with pagination, sorting, and filtering in section 9.6) and headers (section 9.7).

Figure 9.7 depicts the input for a money transfer (`POST /transfers`) and how we can optimize it. This operation transfers money in any currency from one account to another immediately, at a future date, or repeatedly.

	IMMEDIATE TRANSFER	DELAYED TRANSFER	RECURRING TRANSFER
	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100, "currency": "EUR" }, "type": "immediate", "frequency": "none", "date": "2024-01-01", "endDate": "2024-01-01" }</pre>	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100, "currency": "EUR" }, "type": "delayed", "frequency": "none", "date": "2024-06-01", "endDate": "2024-06-01" }</pre>	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100, "currency": "EUR" }, "type": "recurring", "frequency": "monthly", "date": "2024-01-01", "endDate": "2100-01-01" }</pre>
Source account currency by default	→		
Determined based on provided data	→		
	Unnecessary in this context	Unnecessary in this context	Optional
	MINIMIZED INPUTS		
	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100 } }</pre>	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100 }, "date": "2024-06-01" }</pre>	<pre>POST /transfers { "source": "FR123", "destination": "FR543", "amount": { "value": 100 }, "frequency": "monthly" }</pre>

Figure 9.7 We minimize money transfer inputs by determining data based on other data and using default data.

The money transfer type (immediate, delayed, recurring) is unnecessary. The implementation can set it based on the presence or absence of other data. If there are only source, destination, and amount, it's an immediate transfer. If a date is added, it becomes delayed. And if frequency is present, it's a recurring transfer. Any transfer requires source and destination accounts and the amount of money. Although value is necessary, currency can be optional; the implementation can use the source account currency by default. The frequency is only needed for a recurring transfer. Its date range can be optional. The default start date is today, and no `endDate` means the recurring transfer is executed until the user ends it.



CAUTION Not all data is suitable to be server-defined; making the wrong data optional can lead to inaccurate data. For instance, the transaction date and time of a payment card transaction can't be set by the server because the transaction may be received by the server a few minutes or hours after its execution.

9.5 *Returning ready-to-use successful responses*

Designing user-friendly data (section 8.3.3) helps return ready-to-use successful responses. However, before that, the principles we learned when representing operations with HTTP (section 4.1) and modeling data (section 5.1) can help us with this. This section revisits and explains these principles from the perspective of user-friendliness and interoperability.

9.5.1 *Choosing adequate HTTP status and HTTP-compliant data locations*

Similarly to requests (section 9.4.1), complying with HTTP by using the proper HTTP status and adequate data location makes responses user-friendly and interoperable. When a money transfer is created successfully, we can technically return a 299 HTTP status code. Anyone with basic knowledge knows that it means success because it belongs to the 2XX success class, even if they don't know its precise significance. 299 is an unassigned HTTP status, which we can give whatever meaning we want, like "Money Transfer Created." However, I do not recommend doing this because it will surprise developers and may cause problems in their code or with HTTP intermediaries. I recommend following the HTTP documentation and returning a 201 `Created` when creating a money transfer.

According to HTTP, the response body represents the requested, created, or updated resource. This is where developers expect to see the data in response to a money transfer creation, so we should avoid surprising them using headers (unless we use the standard `Location` header to indicate the created transfer resource path).

9.5.2 Returning sufficiently informative data

An operation must return sufficiently informative data. This section explains the principles introduced in section 5.4 regarding using complete, summarized (a subset of complete), and minimal (resource ID only) data models in light of user-friendliness concerns.

Don't hold back on providing helpful data, even if it means returning a lot of information in lists (see section 13.1 for performance concerns). For instance, it is better not to use the minimal transaction model when searching an account's transactions. The transaction ID alone isn't helpful; developers must read each transaction to get valuable data. A summarized transaction model comprising `id`, `amount`, and `date` properties is better but falls short. Developers and end users are interested in additional information such as the transaction description, origin (shop), and type (bank card versus transfer). This represents almost all transaction data; using the complete model in the transactions list makes the most sense (see figure 9.8).

MINIMAL MODEL	SUMMARIZED MODEL	COMPLETE MODEL
<pre>[{ No interesting data { "id": "123" }, { "id": "456" }]</pre> <p><i>GET /transactions/{id} for each transaction</i></p>	<pre>[{ Missing important data { "id": "123", "amount": 100, "date": "2024-04-06" }, { "id": "456", "amount": 50, "date": "2024-04-05" }]</pre>	<pre>[{ Everything developers need { "id": "123", "amount": 100, "date": "2024-04-06", "description": "...", "origin": "...", "type": "...", ... }, ...]</pre>

Figure 9.8 We should return the complete transaction model when listing an account's transactions so developers have everything they need and don't have to call "Read transaction" for each transaction.

A creation or an update always implies generating or modifying data that the consuming application didn't send and is most likely interested in. That's why it is recommended to always return the complete resource rather than the minimal model in these cases. For instance, when creating a money transfer, the execution date of an immediate transfer may not be today as expected because it's a holiday. So, returning the ID is not enough; returning the complete money transfer model is better.



CAUTION Be careful not to return completely heterogeneous data including, for instance, all the account and owner information in a transaction when searching for transactions; it's not the purpose of the operation. See also section 9.9.

9.6 Filtering, sorting, and paginating lists

List and search operations require filtering, sorting, and pagination features, as illustrated in figure 9.9, to be flexible and allow consumers to access necessary data easily. For instance, bank accounts may have many transactions, but consumers often need specific ones, like the first 10 from March 6, 2024, at a restaurant or bar, with amounts between 100 and 400 (excluded), sorted from largest to smallest. If `GET /account/{accountId}/transactions` returns all transactions, it complicates obtaining these specific ones.

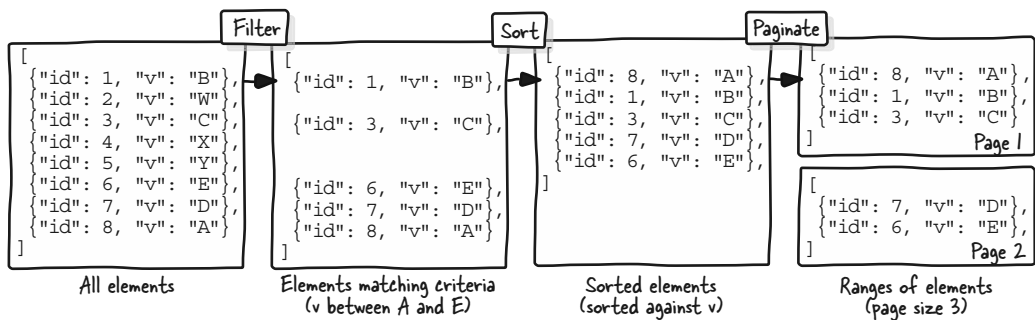


Figure 9.9 Enabling filtering, sorting, and paginating lists is essential to allow consumers to get only the elements they want, sorted as needed.



NOTE Filters and pagination reduce data volume, enhancing API efficiency; see section 13.1. This section covers filtering, sorting, and pagination with selected design options. However, many more exist; see section 16.2 for choosing design solutions adapted to your context.

9.6.1 Designing guessable filters that map returned data

Adding search parameters or filters to a list operation allows consumers to find the subset of elements matching specific criteria and get the needed data. Filters must map the returned data so developers can easily understand, guess, and provide it.

We must name and type search parameters accordingly with filtered element data. For example, suppose a transaction has `date`, `category`, and `amount` properties. To retrieve transactions from March 6, 2024, in a restaurant, with an amount equal to 100, a consumer can call `GET /accounts/12345/transactions?date=2024-03-06&category=restaurant&amount=100`. The query parameter names and types are similar to the transaction data. If the `amount` property is an object with a `value` and `currency`, we can use a fully qualified name such as `amount.value` or `amountValue`. I prefer the dotted notation as it clearly maps the data structure and is similar to code.

Avoid filtering on data that is not returned. It is inconvenient to look for transactions with bar or restaurant categories without knowing which category has a returned

transaction. Also, developers won't be able to guess whether the category filter exists based on returned data if there is no category property.



NOTE As we learned in section 5.3.2, designing filters based on output data ensures consistency between input and output. Filters will also inherit the user-friendly, interoperable features of output data.

9.6.2 Designing flexible filters

To enable consumers to retrieve the necessary data, we shouldn't just filter by specific values; we can use enumerations, ranges, or fuzzy values. The `category` filter can become an enumeration to filter transactions in restaurants and bars. As seen in section 7.6.2, it can be represented as `category=restaurant,bar` or `category=restaurant&category=bar`.

We can filter on amount or date ranges. For example, `amount.gte=100&amount.lt=400` or `amount=gte:100&amount=lt:400` filters on amounts greater than or equal (`gte`) to 100 and less than (`lt`) 400 ($100 \leq \text{amount} < 400$). Instead of using `gt`, `gte`, `lt`, and `lte`, ranges can be represented with prefixes or suffixes such as `from` and `to` or `min` and `max` (`minAmount` and `maxAmount` or `fromDate` and `toDate`). However, they don't clearly state whether the comparison is strict: do `from` or `min` mean `>` or `≥`?

Filters can accept less strict values. For example, accepting a shorter date value, such as `fromDate=2024-03` or `date=2024`, simplifies month and year range inputs. If the request has a `description=good` filter, the implementation can look for transactions with a description that includes the "good" string, not just this exact value. We can also accept regex filters: `description=^good|nice` looks for transactions whose description starts with "good" or "nice."

9.6.3 Enabling free search and complex logic with a *q* filter

We can use a `q` parameter to enable complex logic or free textual search covering different properties. For example, `GET /account/12345/transactions?q=nice restaurant last week` returns transactions based on their date, category, and description. We can also use an engine- or database-like query, which gives us more possibilities for logical conditions. I recommend not inventing a syntax but using an existing one like that used by Lucene, which is an open source search engine: for example, `GET /account/12345/transactions?q=date:2024-03-06 AND (category:"restaurant" OR category:"bar") AND amount:{100 TO 400}`. The sample `q` values are not URL encoded for readability, but consumers will need to URL encode them in their requests (`nice%20restaurant%20last%20week`, for example).



NOTE Not all consumers may need that complexity. Simple query parameters and `q` options can be proposed to satisfy all consumers. Additionally, the `q` parameter can be combined with other filters. Implementing such `q` filters may not be evident; check with the implementation developers.

9.6.4 *Minimizing filters*

Providing too many filters can make the design and implementation needlessly complex. However, we must not artificially limit the number of filters; focusing on meeting user needs helps select the appropriate filters. It's OK not to have filters if they are not needed. Having filters for every transaction property, excluding its ID, makes sense when listing an account's transactions. We can enable searching for account owners by name but not by their address.

Filters must be optional and, unlike body data, often don't have default values (section 9.4.4); `GET /accounts/1234/transactions` will return all transactions for the 1234 account regardless of their date, category, or amount. A required filter may indicate a missing resource identifier in the path. For example, `GET /transactions?accountId=12345` with a required `accountId` represents an account's transactions; it must be represented with `GET /accounts/12345/transactions`.



CAUTION Query parameter filters must exclude sensitive data; see section 12.6.2 for guidance. If filter combinations lead to URLs over 2,000 characters, refer to section 14.2.4 for insights on this limit and workarounds.

9.6.5 *Enabling sort with helpful defaults*

Consumers may need to sort list elements, whether they use filters or not. A list can be sorted on one or multiple values in ascending or descending order. For example, using `GET /accounts/12345/transactions?sort=amount:desc` will return the 12345 account transactions sorted by descending amount. Similarly, `sort=date:desc,amount:asc` will sort transactions by descending date (from most recent to oldest) and ascending amount (from smallest to largest).

To ensure ease of use for consumers, keep the `sort` parameter optional and provide a helpful default ordering. Most consumers will prefer recent transactions first, so `GET /accounts/12345/transaction` should return transactions in reverse-chronological order as if `sort=date,desc` were provided. Making direction optional and setting it to `asc` by default will simplify things. Thus, `sort=date:desc,amount` is equivalent to `sort=date:desc,amount:asc`.

9.6.6 *Paginating lists*

Pagination allows consumers to get elements based on their position in a list (possibly sorted or filtered). They can get the first *N* elements or jump to a deeper position without going through unneeded elements.

With index-based pagination, `GET /accounts/12345/transactions?page=3&size=10&amount=gt:200` returns the third page (`page=3`) of 10 elements (`size=10`)—that is, the 21st to the 30th—of account 12345 transactions having an amount greater than 200. Change the pagination parameters to `page=1&size=10` to get the first 10 transactions. With cursor-based pagination, `GET /accounts/12345/transactions?after=2501&size=25&amount=gt:200` returns the 25 transactions after the one identified by the 2501 cursor (which happens to be its ID).



NOTE Pagination is mainly related to performance concerns. Section 13.6 discusses optimizing page size and explains cursors and when to use index- or cursor-based pagination.

Using standards is generally a good idea, but not always. We could use the `Range` HTTP header for pagination, but it's best not to. HTTP intermediaries may strip this header if it's used for a purpose other than retrieving chunks of binary data. Having pagination parameters separate from the filter and sort parameters complicates inputs. Ultimately, we should stay within the common practice: nobody uses `Range` for pagination.

Keep pagination parameters optional, and choose helpful default values to facilitate the operation's use. A month of transactions provides data that's useful for many use cases, and an average bank account has around 40 transactions per month, so we decided that `GET /accounts/12345/transactions` returns the first 40 transactions.



NOTE Filter, sort, and pagination parameters are query parameters because they are resource modifiers (section 4.4.5); this is also common practice.

9.6.7 Returning filter, sort, and pagination metadata

As we learned in section 9.5.2, we should return informative filter, sort, and pagination data; in particular, it can help consumers know which default values are used. This is not actual subject-matter data but metadata related to the list, so as we learned in section 8.6, we must reorganize response data to make a clear separation.

As shown in figure 9.10, the response of the “List account transactions” operation is no longer an array of transactions but an object with data and metadata properties.

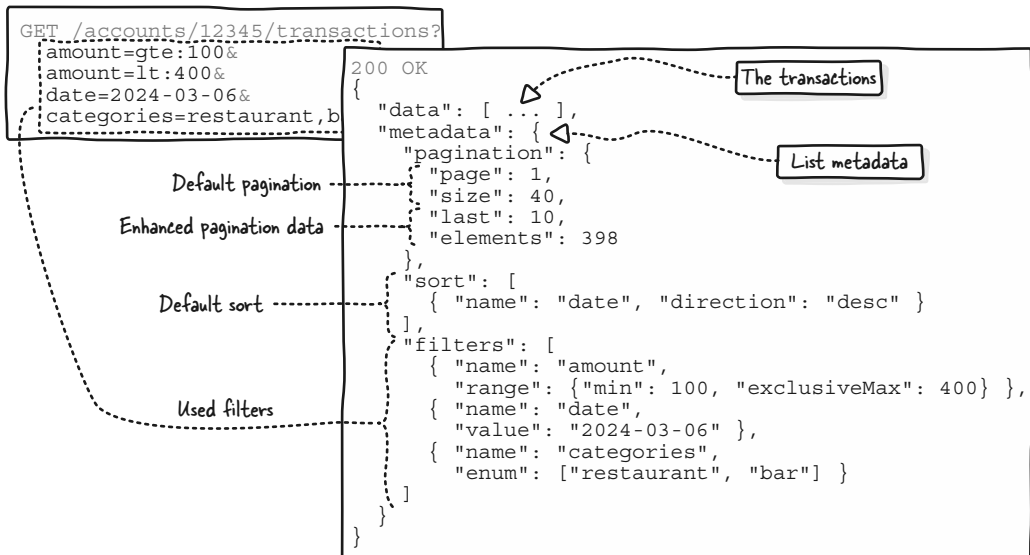


Figure 9.10 The pagination, sort, and filter metadata lets consumers know exactly what they will get (especially when default parameter values are used) and whether other elements are available.

Section 9.10.1 discusses the choice of the name data. The metadata object proposes a clear organization with `pagination`, `sort`, and `filters` properties.

If a parameter is not provided, metadata indicates the default value used. For instance, if no sort or pagination parameters are provided, the operation returns the first 40 transactions in reverse chronological order. This is indicated in metadata's `pagination` and `sort` properties. We also enhance the pagination metadata with the last page index and the total number of transactions (`elements`) to facilitate working with the list.



NOTE We decided to always return a successful status when data is found (section 3.4.5); this makes search operations user-friendly and is discussed in section 9.8.1.

9.7 Adapting request and response data

Another way to make operations flexible is to adapt data to consumers' needs. We can return or accept different data formats, adapt to the consumer locale, or tweak returned data.

9.7.1 Handling different data formats

Thanks to *content negotiation*, HTTP supports various data formats, not just JSON, in requests and responses, which can simplify consumers' work. Many end users analyze transactions in a spreadsheet, where importing comma-separated values (CSV) data is easier than JSON. Figure 9.11 shows that `GET /account/{accountId}/transactions` returns data in CSV format if the consumer sends an `Accept` HTTP header with the `text/csv` media type (see section 6.6.2). Check the IANA media types list for other standard media types (www.iana.org/assignments/media-types/media-types.xml). The

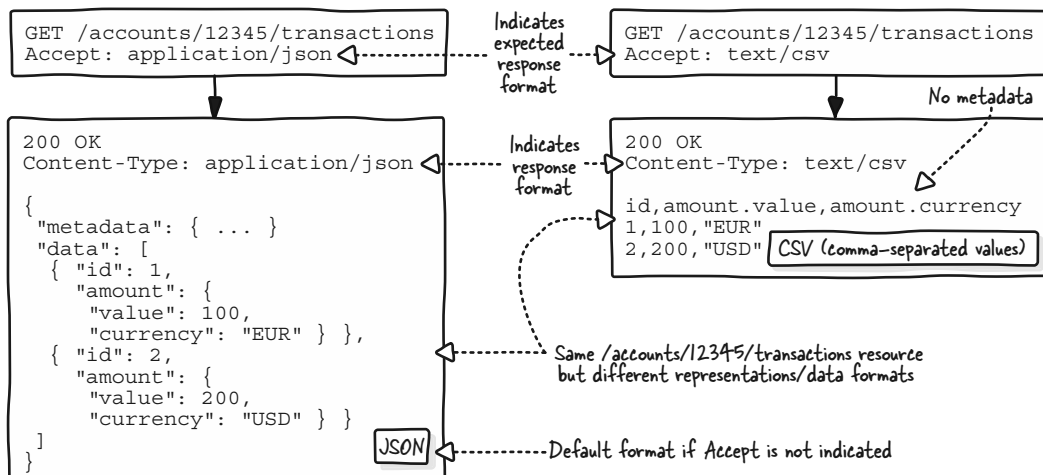


Figure 9.11 Content negotiation allows consumers to request an account's transactions in different formats by sending a standard `Accept` header.

response has a `Content-Type: text/csv` header to indicate that the response body is in CSV format. Sometimes, data may be different depending on the format; for example, the CSV response lacks the list metadata.

As we learned in section 9.4.4, we make the `Accept` header optional and return JSON data as the default. Whether consumers send an `Accept: application/json` request header or not, they'll get JSON data.

HTTP also supports different formats in requests, as shown in figure 9.12. Some banking systems that need to transfer money can send XML natively but require extra work to produce JSON. When using `POST /transfers` to transfer money, consumers can send XML in the request body by adding a `Content-Type: application/xml` header. Formats for the request and response don't have to match. Consumers can add `Accept: application/xml` to their JSON request (`Content-Type: application/json`) and get an XML response (or vice versa).

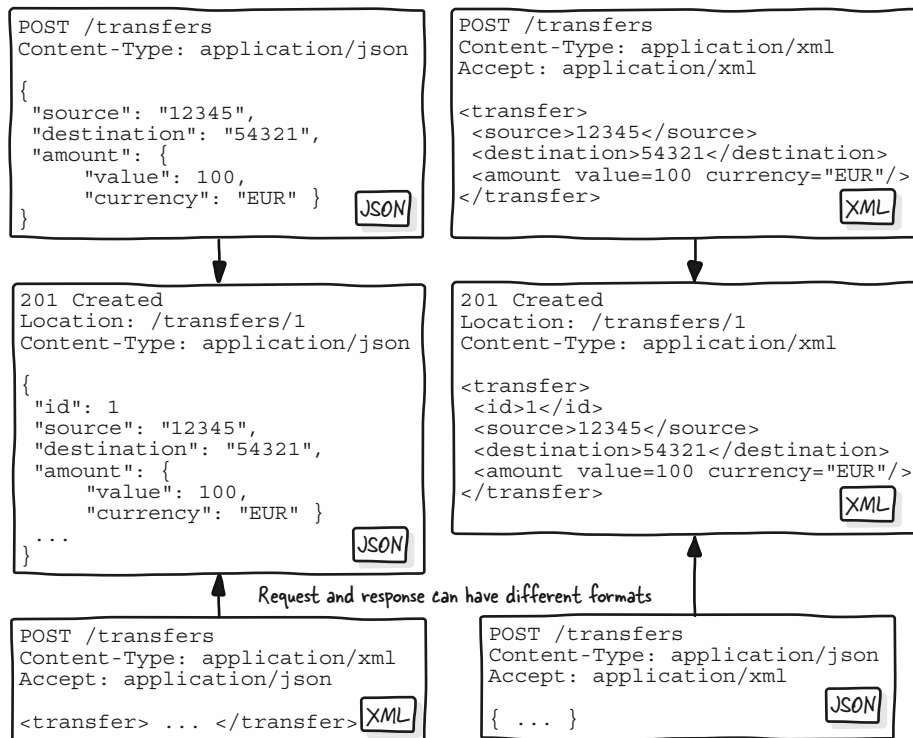


Figure 9.12 It is possible to transfer money with a JSON or XML input and get a response in JSON or XML regardless of the input format.



NOTE Operations must only support the formats that make sense for the identified needs; section 9.8.2 discusses handling unsupported media types.

Not all operations need to support the same formats, but a certain level of consistency is essential; see section 9.10.

9.7.2 *Translating data and adapting to locale*

Similarly to section 9.7.1, we can use HTTP content negotiation to adapt to the consumer's or end user's locale to translate data. `GET /accounts/{accountId}/transactions` returns a list of transactions, each with a category code and label (`travel_agency` and `Travel agency`, for example). The label can be displayed to end users, but English may not be appropriate for everyone.

Consumers can send the `Accept-Language` HTTP request header to get the label in French (`Accept-Language: fr, Agence de voyage`) or English (`Accept-Language: en, Travel agency`). As we saw in section 9.4.4, we can make this header optional and choose English as the default value because it suits most cases.

The translation must only affect values that are meant for humans and not used in consuming application code; we must not translate property names or human-readable codes. For example, developers of consuming applications who want to show a pictogram for the transaction category would have difficulty managing the `category` (English) or `categorie` (French) properties or `travel_agency` and `agence_de_voyage` category code depending on the end-user language.

The `Accept-Language` header supports language variations: for example, `fr-FR` for French from France and `fr-CA` for the Canadian variant. Additionally, consumers can prioritize languages with the `q` parameter representing a weight. The `fr;q=0.9, it;q=0.8, *;q=0.5` value means they prefer French over Italian and will accept any language if neither is available.

The `Accept-Language` header also dictates how data can be formatted as strings, which an API usually lets consumers handle (section 8.5.1). However, an API may sometimes need to handle this formatting. For example, if our Banking API returns the list of transactions as a PDF (`application/pdf` content type), the API implementation can format amounts based on `Accept-Language`.



NOTE We're not obligated to support all possible languages; section 9.8.2 discusses handling unsupported ones.

9.7.3 *Tweaking returned data*

In addition to HTTP content negotiation headers, we can use query parameters to tweak the returned data. A typical use is to change the units used in the returned data. For example, when reading a bank account with `GET /accounts/1234?currency=EUR`, the balance amount will be in `EUR` instead of the default `USD`. This also works on search operations, but I chose this example because using query parameters on a read operation is uncommon and is often overlooked, if not wrongly considered incorrect.



CAUTION Don't use this technique to drastically change the data structure or enable the return of different resources. Check out section 14.2.2 if you need to tweak the resource model that is returned.

9.8 Handling consumer errors gracefully

Considering how operations handle errors is essential to creating a user-friendly, interoperable API. Operations must return intuitive, informative, problem-solving, and exhaustive feedback on consumer errors. But before that, our design must prevent errors as much as possible. Handling consumer errors in a user-friendly, interoperable way requires

- Limiting errors
- Using adequate HTTP status codes
- Providing informative and problem-solving feedback
- Returning machine-readable feedback
- Returning all possible errors
- Using standards



NOTE This section focuses on consumer errors related to inputs. Section 12.10 discusses consumer and server errors from the security perspective, and section 14.2.3 discusses planned interruptions (server errors).

9.8.1 Limiting consumer errors

To avoid unnecessary errors, we can use a common software principle, Postel’s law (also known as the robustness principle), and create flexible operations by being liberal in what they accept and conservative in what they return. We can also carefully consider what an error is.

Consumers can modify pending money transfers with `PUT /transfers/{transferId}`. If the input data checks are strict, a consumer sending data returned by `GET /transfers/{transferId}` with some modifications will receive a 400 Bad Request for sending the complete data model instead of the required “creation or replacement” version (see section 5.4). They must exclude nonmodifiable data like `id` before updating, which is inconvenient. It’s preferable to ignore extra data and return a 400 Bad Request only if there’s a problem with modifiable data. We can also propose `PUT` and `PATCH /transfers/{transferId}` to allow consumers to send the full transfer data or just the necessary modifications.

We can also be flexible with properties. We may face time-zone complexities if a money transfer is triggered at a specific date and time. Accepting only Coordinated Universal Time (UTC, 2024-03-10T14:10:00Z) simplifies our work but complicates things for consumers, leading to the rejection of other formats (2024-03-10T05:10:00+09:00). Instead, we can accept all options and always return UTC. Also, supporting a date (2014-03-10) would be nice for consumers who don’t care about the time.

If `GET /accounts/12345/transactions?category=restaurant` finds nothing, it can return either 200 OK (with an empty list) or 404 Not Found (the `/accounts/12345/transactions?category=restaurant` URL doesn’t exist); both make sense for HTTP. But a successful empty list may simplify the code and cause fewer bugs in lazy consumer applications—and that is common practice.

9.8.2 Using adequate HTTP status codes

User-friendly error feedback requires HTTP-compliant and usual HTTP statuses. Section 4.5.8 discussed typical options, and this section discusses some new ones.

Consumers can obtain account transactions in CSV format by including the `Accept: text/csv` header in their request (section 9.7.1). However, if they request the unsupported PDF format (`Accept: application/pdf`), the response will be 406 Not Acceptable. This also applies to those requesting data in unsupported Japanese using `Accept-Language: ja` (section 9.7.2).

The “Transfer money” operation supports an XML request body (`Content-Type: application/xml`). However, if consumers send unsupported CSV data (`Content-Type: text/csv`), the response must have the 415 Unsupported Media Type status.

400 Bad Request can indicate errors like invalid JSON, a missing source account, or an insufficient balance in a money transfer. Alternatively, 422 Unprocessable Content can be used if the body can be parsed (valid JSON) but validation fails due to data or business rules, such as a missing source account or insufficient balance. Despite 422 being “more HTTP compliant,” I prefer 400 as it addresses body and query parameter problems with one code (see section 9.8.5). The choice is yours.

Returning 404 Not Found on `POST /transfers` because the source account doesn’t exist is not HTTP compliant. In this case, 404 means the `/transfers` resource was not found; we must use 400 (or 422) instead.



CAUTION As discussed in section 9.5.1, remember not to surprise developers, their applications, and HTTP intermediaries with custom HTTP status codes that use unassigned values.

9.8.3 Providing informative, problem-solving feedback

User-friendly error feedback helps consumers understand the problem and how to solve it. If a `POST /transfers` with `Content-Type: text/csv` returns 415 Unsupported Media Type, consumers may assume that the format is not supported. To aid them, we can return `Accept: application/json, application/xml` with the 415 status to indicate supported formats. Although this is standard HTTP behavior, not all developers may realize that they need to check response headers for more details. Additionally, we can return a JSON body with a `message` stating, “text/CSV is not supported; use application/json or application/xml.”

On the same operation, returning 400 Bad Request for a missing source account in the request body falls short. Returning a clear `message` property like “Required source account is missing” will help consumers fix the request immediately. Similarly, if a transfer only supports EUR and USD but a consumer sends JPY, the message should specify, “Unsupported JPY currency; amount currency must be EUR or USD” rather than just “Unsupported JPY currency,” which is unhelpful.

In the case of 404 Not Found on `GET /customers/54321/addresses/home`, returning a “customer 54321 doesn’t exist” message will help consumers know which of the `customerId` and `addressId` path parameters was not found in `/customers/`

{customerId}/addresses/{addressId}. To facilitate understanding, we can use Accept-Language, as in section 9.7.2, to return the error message in a language that developers or end users can understand.

9.8.4 Returning machine-readable feedback

A message string may not always help determine the origin of the error, or consumers may need machine-readable information (to show errors to end users in the UI, for instance). As shown in figure 9.13, we can add the source and type of errors and optional values (if any). The source property indicates the location of the problem (header, path, query, or body), the element's name, and an optional JSON pointer indicating a location in the request body (as used in OpenAPI; see section 7.7.1). The type is a human-readable, generic error code; we don't want to overwhelm consumers with hundreds of specific, cryptic numeric codes. For instance, `INVALID_FORMAT` can be used if the body is in an unexpected media type or a date property doesn't have a proper ISO 8601 date format.

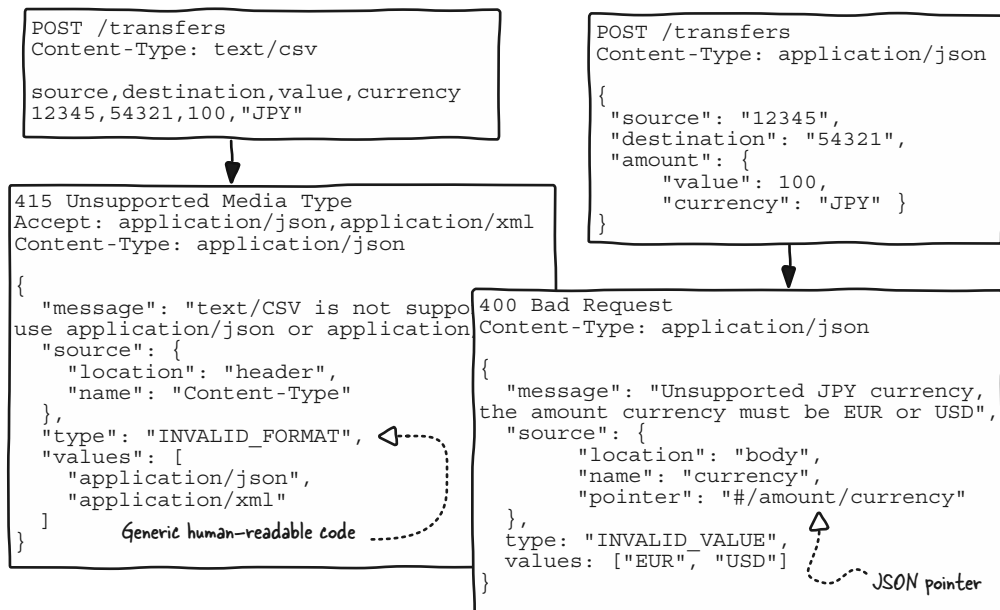


Figure 9.13 Money transfer machine-readable errors returned when an unsupported data format or currency is sent make it easy to show errors in the UI.

9.8.5 Returning an exhaustive list of errors

User-friendly feedback is exhaustive and lists all errors. If a consumer sends a `POST /transfers` request with a missing destination account and an amount greater than the account balance, the operation can return an error message indicating the first

problem (“Required destination account is missing”). The consumer fixes it, tries again, and gets the second error (“Insufficient balance”). Imagine such behavior in a mobile application or on a website; it’s an irritating experience. As shown in figure 9.14, the operation must return an exhaustive list of errors indicating the two problems to provide developers and end users with the best experience. To limit back-and-forth, the implementation should ideally perform all possible controls on the possibly imperfect provided data in one shot, including basic schema control (missing destination account, for example) and business rules (insufficient balance).

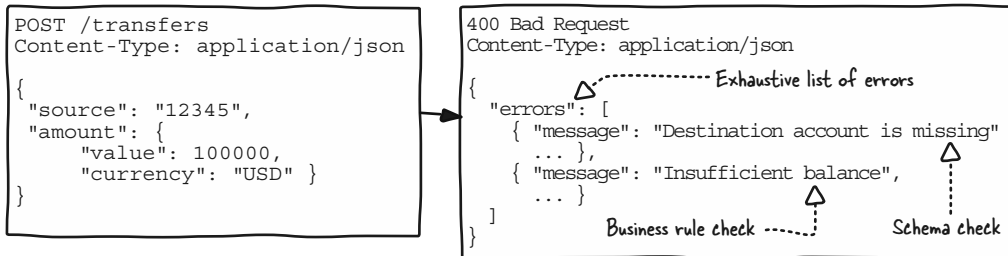


Figure 9.14 The “Money transfer” operation performs all possible checks and returns an exhaustive list of errors, which avoids annoying back-and-forth.

9.8.6 Using standards

We can use internal, industry, or general standards to create easy-to-use, interoperable error feedback. This section briefly examines the “Problem Details for HTTP APIs” standard; check its documentation for more details (www.rfc-editor.org/rfc/rfc9457.html).

The Internet Engineering Task Force (IETF) defined the “Problem Details for HTTP APIs” standard to “avoid the need to define new error response formats for HTTP APIs.” The format can be extended with custom properties. Figure 9.15 shows an example based on previous sections.

The `Content-Type` is `application/problem+json` instead of the generic `application/json`. It lets consumers know precisely which error format they use (`application/problem+xml` exists, too).

The status is the returned HTTP status. The `type` identifies the problem type; if its value is a URL, it may be called to get more information about the problem (in the form of an HTML page). The `title` is a human-readable description of the problem type; it’s the same in all occurrences. The `detail` is a human-readable description of the problem. Both the `title` and `detail` may be translated. All these properties are defined by the format.

The `errors` list is a custom property, almost the same as defined in previous sections. In each error, we rename the `message` property to `detail` to be consistent with

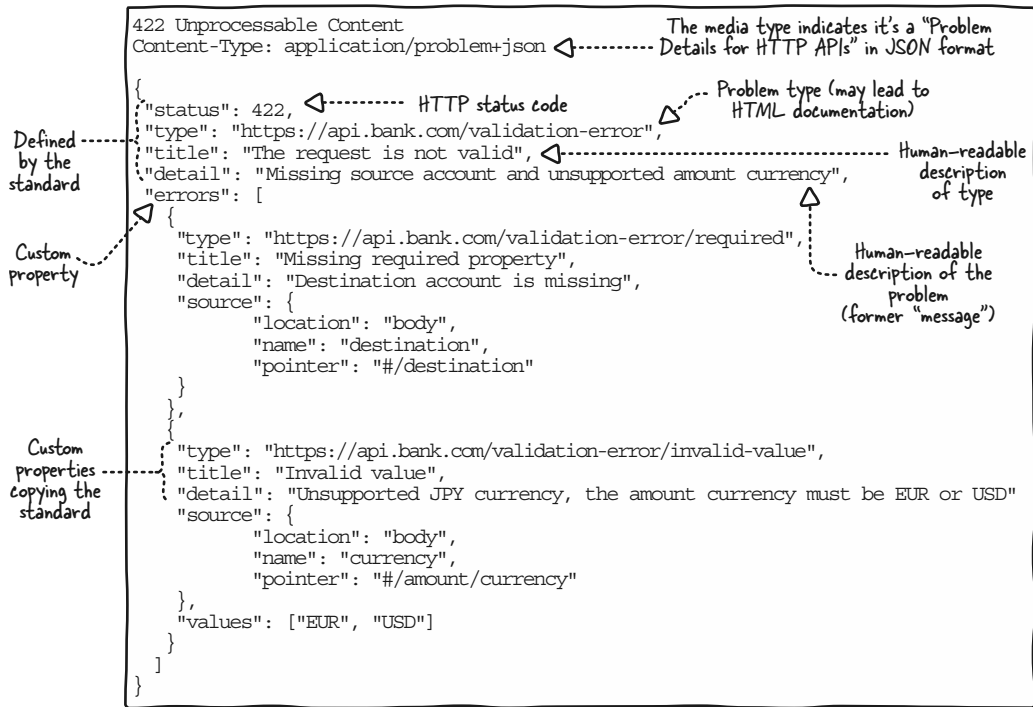


Figure 9.15 An error response for the “Money transfer” operation using and customizing the “Problem Details for HTTP APIs” standard

the format. Similarly, we replace the `type` value with a URL. We also add a generic human-readable `title` describing the type.

9.9 Avoiding hiding multiple capabilities in a single operation

Our work on API capability identification (section 2.1) should limit risks, but we should be cautious about creating do-it-all operations that hide capabilities from developers. This section discusses

- Ensuring that request and response data has appropriate granularity
- Ensuring that operations have a clear purpose

9.9.1 Reconsidering request and response data granularity

We can investigate request and response data granularity, as we learned in section 8.7, to ensure that an operation has appropriate granularity. We can challenge a `GET /accounts/{accountId}` (“Read” account) whose response includes transactions that need to be filtered, sorted, and paginated; a dedicated “Search” account’s transactions would be better to make the capability more visible. But a more specific “Get

dashboard” operation that combines summarized accounts and their latest transactions may make sense.

Similarly, we can check the input data. It’s not because the “Read an account” operation returns the holder’s information, such as their address, that we should include this information in the “Update account” operation input (`PUT /accounts/{accountId}`). Although related, an account and the customer holding it are distinct concepts requiring distinct operations. A dedicated “Update customer address” is better (`PUT /customers/{customerId}/addresses/{addressId}`) because it has a clear purpose. We could also challenge having such detailed information about an account holder at the account level, but it may make sense depending on the needs and subject matter.

9.9.2 *Reconsidering an operation’s purpose*

Sometimes the need for separate operations is less evident because the data forms a cohesive whole. Resources are essential, but don’t lose sight of capabilities. We can check the purpose of an operation to ensure that it is accurate.

An account has a `status` indicating whether it’s active or closed. We may be tempted to use `PUT /accounts/{accountId}` and set its `status` to `closed` to close it. However, closing an account is an essential action that deserves an independent capability and operation. Based on what we learned in section 4.7, we could represent it with `POST /accounts/{accountId}/close` (action resource) or `POST /account-closures` (result resource). A `DELETE /accounts/{accountId}` is also possible. But if closing an account is one of many important events, a `POST /accounts/{accountId}/events` is preferable. Whatever option we choose, the `status` visible when reading the account with `GET /accounts/{accountId}` will reflect what happened.

On the other hand, there is no need for a dedicated operation to modify the end-user-defined account title. Using `PUT /accounts/{accountId}` is an appropriate solution even if it’s the only property that can be modified.

9.10 *Aiming for consistency and standardization*

As was the case for the data in section 8.9, we must make consistent decisions and aim for standardization when designing operations to ensure that our API is user-friendly and interoperable and to simplify wiring systems via APIs. This applies to operation data, features, and behaviors. Operations must at least be consistent within the API. But it’s best to be consistent and share standards with other APIs in the organization, the domain or industry, or even the rest of the world.



NOTE Being consistent and creating or using the proper standards can be difficult. Check out section 16.1 to discover how to overcome this.

9.10.1 Using standardized data consistently

Follow section 8.9 to design consistent path identifiers, resource paths, query parameters, request and response bodies, and headers. This section illustrates how data consistency and standardization affect operation use.

An operation's input and output data must be standardized as much as possible to be provided and used easily. Using an IBAN as `accountId` in `/accounts/{accountId}` and ISO 4217 currency codes for balances and money transfer amounts (EUR, USD) is best. If there is no standard, the well-known `customerId` shared among systems is better than an internal ID known by a single system.

Consistent data patterns simplify using data and the API as long as the data is interoperable. Our standardized models in section 5.4 ensure that a customer, account, transaction, or any other resource model has a recognizable `id` property that consumers know they can use to read, update, or delete resources and can probably use any time the resource is mentioned in a data model. However, that works only if the data is consistent across resources and operations. For instance, `GET /customers` returns a "customer ID," but account creation requires a "customer reference," returned by `GET /customers/{customerId}`. Although both identify a customer, they are distinct. Developers may spend hours troubleshooting account creation problems due to these inconsistent identifiers.



CAUTION APIs should hide inner complexity, such as differing identifiers for the same element. If identifiers vary for similar resources in different contexts, verify whether they represent distinct concepts that need unique names and identifiers (section 5.4.5).

Similar to `id` properties, we returned transactions in a generic `data` property when searching for transactions in section 9.6.7. Naming the list `transactions` could clarify its content. However, searching for accounts would return an object with an `accounts` list. The location of the list is different. Both generic (`data`) and specific name (`transactions`) options are common; I prefer a generic name (like `data` or `items`) for consistent access because developers can refer to the resource path (`/transactions`) to understand the content. Now that we have decided, we must stick to it for all list operations; we can't have `data` from `transactions` and `accounts` for `accounts`.

9.10.2 Adopting standardized behavior consistently

When and how operations succeed or fail must be standardized to avoid surprising developers and causing their code to crash. Once the behaviors of a single operation are defined, a standard is de facto established. All operations designed after that must consistently return a similar status and data under the same conditions. Let's look at a few examples.

If creating a money transfer returns `201 Created` with a `Location` header and a complete resource model, creating a customer, an account, or anything else should behave the same way. Having account creation only return the `Location` header and

204 No Content or customer creation return only the resource ID with 200 OK will complicate developers' work.

If searching for accounts returns 200 OK with an empty list when no accounts matching the criteria are found, searching for an account's transactions must not return 404 when no transaction matches the criteria.

If transferring money fails with a 422 Unprocessable Entity and an application/problem+json error response with a detailed list of errors when the destination account is missing or the source account balance is insufficient, we'd better not return 400 Bad Request with a message string only indicating that an invalid account type was provided when failing to create an account because of a missing owner ID or invalid account type.

If PUT /transfers/{transferId} accepts a complete transfer model (ignoring the extra unmodifiable properties), PUT /owners/{ownerId} must not return 400 or 422 when provided with an Owner complete model.



NOTE Respecting HTTP semantics (method, status, standard headers), using the models from section 5.4, and designing predictable paths (section 9.3.2) will seamlessly incorporate significant standardization in operations.

9.10.3 *Offering standardized features consistently*

We must be consistent in adding features, such as filtering, sorting, pagination, and content negotiation, to operations and in designing them (parameter names, types, and values). Once they see a generic feature on one operation, developers and their code expect it to be applied to all operations of the same type with the same design.

If “Search account transactions” (GET /accounts/{accountId}/transactions) proposes pagination but “Search account” (GET /accounts) and “Search customers” (GET /customers) don't, developers will be badly surprised. If GET /accounts/12345/transaction?amount=gt:400&sort=amount:desc returns transactions with an amount greater than 400 ordered by decreasing amount, the parameters should be similar on GET /accounts. Searching accounts with a balance greater than 2,000 ordered by increasing balance should be done with balance=gt:2000&sort=balance:asc, not minBalance=2000&order=balance,up.

If we handle XML input and output on POST /transfers, we should support XML in all operations of the Banking API. Similarly, if “Search account transactions” can return CSV data, we should consider adding this possibility to any other search operations; it doesn't make sense for other operations, such as modifying a customer's address.

Summary

- User-friendly, interoperable operations meet user needs, expose clear capabilities, use user-friendly, interoperable data, and are helpful, consistent, and standard.
- Work simultaneously on data and operation user-friendliness and interoperability in a second pass after designing the programming interface that does the job.

- To design easy-to-understand, guessable operations, use appropriate HTTP methods and design meaningful and hierarchical paths where each segment has a specific purpose (`/resources`, `/resources/{identifier}`, `/resources/{identifier}/sub-resources`, `/resources/{identifier}/sub-resources/{sub-identifier}` ...).
- Use globally unique resource IDs to craft short but accurate resource paths requiring few path parameters.
- Every piece of input data, including path parameters, query parameters, request headers, and request bodies, must be easy to provide.
- To make data easy to provide, use typical and HTTP-compliant input locations, map input to outputs, request well-known or standard data, and minimize required data.
- To minimize input, remove input elements that the implementation can deduce, and make elements optional with a helpful default value when possible.
- Return ready-to-use responses by using an adequate HTTP status, using HTTP-compliant data locations, and returning sufficiently informative data.
- To return sufficiently informative data, always return the complete resource rather than the minimal model on creation or modification. Don't hesitate to include more information in lists if it meets user needs and the subject matter.
- Add filter, sort, and pagination features to list or search operations, but balance these options with needs and complexity.
- All query parameters, including filter, sort, and pagination parameters, must be optional; a required one may indicate a missing path parameter.
- Return filter, sort, and pagination metadata on list and search operations.
- Put list data in a generically named property (`data`, for example) to make it easier to access.
- Support multiple formats (JSON and XML, for example) in input and output using HTTP content negotiation (`Accept` and `Content-Type` headers).
- Translate data using HTTP content negotiation (`Accept-Language` header). Only translate data for humans; don't translate property names or code.
- Return intuitive, informative, problem-solving, exhaustive error feedback, and prevent errors as much as possible.
- Be flexible with inputs to limit errors; be liberal about what is accepted and conservative about what is returned. Accept a complete model on update operations. Accept data format variants, but return a standardized format.
- Return an appropriate error HTTP status, but return an empty list with 200 OK when a search operation finds nothing.
- Error feedback must describe the problem and help consumers fix it. Provide human- and machine-readable error feedback.
- Return all errors. Ideally, the implementation should perform all possible controls on the possibly imperfect provided data in one shot, including basic

schema control (a missing destination account, for example) and business rules (such as an insufficient balance).

- Use standard error data, such as “Problem Details for HTTP APIs.”
- Ensure that each operation has appropriate granularity and doesn’t hide another capability. Check the granularity of the input and output. Also check how the operation is used.
- Input and output data, features, and behaviors must be consistent within and across operations.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You’ll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 9.1

A fitness tracker API allows consumers to search a user’s exercises. Listing 9.1 shows a request for walking exercises for user 5678 between December 20 and 23, 2024. The user ID and dates are required, and the exercise type is optional. Listing 9.2 displays the response to this request. Discuss why and how this operation could be made more user-friendly and interoperable.

Listing 9.1 Search exercises request

```
POST /fitness/tracking/summary/exercises?user=5678
```

```
{
  "exerciseType": "walking",
  "fromStartDate": "2024-12-20",
  "endDate": "2024-12-23"
}
```

Listing 9.2 Search exercises response

```
200 OK
```

```
[
  {
    "id": 123,
    "type": "walking",
    "startTime": 1734699900
    "end": 1734702300,
    "duration": "45 minutes",
    "distance": "3.5 kilometers"
  }
]
```


Exercise 9.2

An API proposes the following operations to get information about available player classes and magic spells in various tabletop role-playing games, such as *Dungeons and Dragons* or *Warhammer*. How could you improve these operations?

- GET /games/{name}/classes.json
- GET /games/{name}/classes.xml
- GET /games/{name}/classes.csv
- GET /games/{name}/spells?format={format} (format is required and can be json, xml, or csv)

Exercise 9.3

Listing 9.3 illustrates how to reschedule an event with an event management API. What's the problem with this operation?

Listing 9.3 Rescheduling an event

```
PATCH /events/12345

{
  "date": "2024-12-26",
  "reason": "Venue conflict"
}
```

Exercise 9.4

Listing 9.4 shows an invalid sample request to add a user with an IAM (Identity and Access Management) API. Listing 9.5 shows the response to this request; why must this response be fixed, and how?

Listing 9.4 Invalid request

```
POST /users
Content-Type: application/json

{
  "username": "curtisnewton",
  "email": "curtis.newton@captainfuture.com",
  "firstName": "Curtis",
  "lastNam": "Newton",           ← Should be lastName
  "permissions": [
    {
      "group": "futuremen",
      "role": "administrator",   ← Should be admin
    }
  ]
}
```

Listing 9.5 Error response

432 Missing And Invalid Data

Exercise 9.5

While optimizing the design for an operation recording blood pressure measurements in a healthcare system API, a designer allows consumers to skip sending the measurement time (the server generates it) and the unit for blood pressure values (always mmHg). Listings 9.6 and 9.7 provide a sample request and response. What problems arise from the resulting design?

Listing 9.6 Record blood pressure request

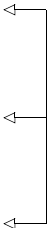
POST /patients/P12345/blood-pressures

```
{
  "deviceId": "AZ456",
  "systolic": 120
  "diastolic": 80
}
```

Listing 9.7 Record blood pressure response

201 Created

```
{
  "id": "BP7890",
  "measurementTime": "2024-20-12T10:34:23Z",
  "deviceId": "AZ456",
  "systolic": {
    value: 120,
    unit: "mmHg"
  }
  "diastolic": {
    value: 80,
    unit: "mmHg"
  }
}
```



Server-defined
values

10

Designing user-friendly, interoperable operation flows

This chapter covers

- Designing concise, error-limiting, flexible flows
- Designing flexible data-saving flows

Imagine that buying 10 copies of *The Design of Web APIs* using the shopping API involves searching for products, listing promotions, merging these two operations' data to show accurate prices, adding the product to the cart 10 times, and failing to check out due to insufficient stock. Such an operation flow leads to a cumbersome experience for developers who are creating applications that consume the API and also for their end users. Even built with user-friendly, interoperable operations, the flows needed to achieve use cases can be complex, require heavy data processing to transform previous output data into the following input, or lead to complex and inflexible UI flows. Optimizing API operation flows can limit such annoyances and ensure a smooth experience for developers and end users.

This chapter examines what makes operation flows user-friendly and interoperable and explains when and how to optimize flows to take these concerns into consideration. We then illustrate how to optimize a flow to make it concise, error-limiting, and flexible so that it is easily usable in various contexts and minimizes

effects on consuming applications and end users. Finally, we show how to optimize a flow to enable flexible partial and one-shot complete data saving.

10.1 What makes an operation flow user-friendly and interoperable?

An *operation flow* is a sequence of calls to API operations that a consumer performs to achieve a use case. These use cases include those we described when identifying API capabilities (section 2.1) and new ones invented thanks to our design’s versatility. We must design our operation flows to be user-friendly and interoperable to enable efficient development in various contexts and ensure an excellent end-user experience. This section illustrates how an operation flow can be user-friendly and interoperable by

- Using user-friendly, interoperable data and operations
- Being designed as a whole
- Being concise
- Being flexible
- Meeting user needs within the flow
- Forming a helpful whole
- Being consistent and standard

10.1.1 Using user-friendly, interoperable elements

A user-friendly, interoperable operation flow is built on user-friendly, interoperable operations that use user-friendly, interoperable data (section 9.1). For instance, to rent a car with the Car Rental API introduced in section 8.2, consumers need to “Search for cars” to find cars matching specific requirements, “Read a car” to get more information about car characteristics, and “Rent a car.” The car IDs returned by the search should be the ones needed to read and rent a car. If the “Rent a car” operation fails because the requested car isn’t available for the given period, it must return feedback that helps to understand and solve the problem.

10.1.2 Being designed as a whole

Although using user-friendly, interoperable elements is a good start, a flow cannot be guaranteed to be user-friendly and interoperable if it isn’t seen as a whole. We must ensure that it forms a concise, flexible, helpful whole that meets user needs.

The six-step “Renting a car” flow in figure 10.1 requires creating a rental request, searching for a shop, adding the shop to the request, searching for cars in a shop, adding the car to the request, and validating the request. Although each step of this flow is individually user-friendly and interoperable, the flow as a whole could benefit from a few enhancements described in the following sections.

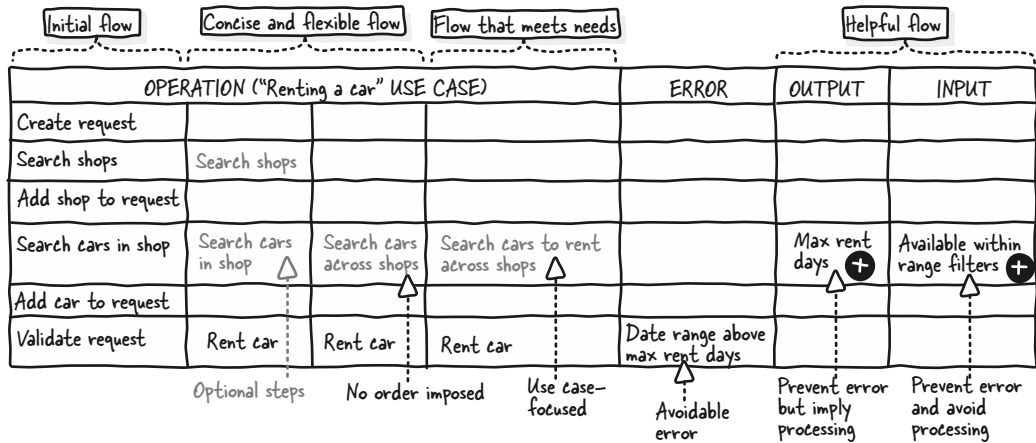


Figure 10.1 We'll enhance the "Renting a car" operation flow by reducing the number of steps, avoiding constraining the order of operations, creating use-case-focused operations, and preventing error and data processing.

10.1.3 Being concise and flexible

An operation flow must include minimal steps and ideally avoid constraining consumers. As illustrated in figure 10.1, the "Renting a car" flow has unnecessary steps and constrains consumers to a specific sequence of API calls, limiting UI possibilities and reuse in other contexts. It can be simplified into a three-step flow involving searching for shops, searching for cars in a shop, and renting a car. The search steps are optional; consumers can call the "Rent car" operation directly if they already have the shop and car information. We can also merge the search steps into a single "Search cars across shops" operation to prevent constraining searching for a shop and then searching for cars. The new flow is concise and flexible. Being able to enter the flow at different steps increases its interoperability.

10.1.4 Meeting user needs within the flow

An operation flow must use versatile but use-case-focused operations that make sense and meet user needs in the flow's context. If our API deals with both renting and buying cars, it would be best not to use a "Search for cars" operation that is usable in both contexts when renting a car but to instead use "Search for cars to rent," which returns only the cars to rent with the data that matters in that context. The data returned by this operation or an added "Read a car to rent" operation should help decide which car to rent. Otherwise, getting only the cars that matter in the rental context and finding the necessary information to decide which to rent may be complicated.

10.1.5 *Being helpful across operations*

A helpful operation flow minimizes processing and the risk of errors. For instance, as illustrated in figure 10.1, if a car can only be rented for a maximum of 20 days, including this information in the search result can help avoid the “Date range above max rent days” error on the following “Rent car” operation. But enabling filtering cars by availability within a date range is even more helpful; it avoids complex data processing based on rental dates and maximum rental days to select cars that match the requirements.



CAUTION The steps and errors of an operation flow can directly affect the end-user experience (section 8.1.3).

10.1.6 *Aiming for consistency and standardization*

As is the case for data (section 8.9) and operations (section 9.10), consistency and standardization make operation flows user-friendly and interoperable. Similar use cases and sub-sequences must use similar flows so developers can guess them and easily code applications consuming the API.



NOTE By making APIs highly intuitive and simplifying coding, consistent and standardized operation flows heavily contribute to the “wow” effect of APIs, providing an outstanding developer experience.

Many flows rely on chaining typical API operations. Thanks to our respect for the HTTP uniform interface and our predictable paths, it’s easy for developers to guess that to modify a rental from scratch, they can GET /rentals (search), GET /rentals/{rentalId} (read), and PUT or PATCH /rentals/{rentalId} (update).

We can define standard flows beyond HTTP. For instance, renting a car or adding one to a shop involves uploading documents, like a driver’s license or car registration. A consistent file-upload pattern makes these flows intuitive and interoperable, allowing developers to guess how they work and reuse code. Section 14.3 discusses file uploads and downloads.

The most typical area where flows from APIs of different organizations will use an actual standard is security. For example, the flows to obtain the credentials needed to call an API are standardized by security frameworks, which section 12.1 illustrates with an example.



NOTE As we do for data and operations, we implicitly define patterns when designing operation flows; developers expect to see them used in similar contexts. Being consistent and creating or using the proper standards can be difficult. Check out section 16.1 to discover how to overcome this.

10.2 *When and how to optimize flows*

Now that we’ve seen what makes operation flows user-friendly and interoperable, we can discuss when to consider optimizing our flows from these perspectives.

10.2.1 When to consider flow optimization

As shown in figure 10.2, we're still working on the user-friendly, interoperable layer, a second pass through our initial versatile API design that does the job. Trying to optimize use cases while identifying API capabilities risks, at best, lengthening the discussion and, at worst, can lead to a design failing to meet user needs. In the user-friendly, interoperable layer, optimizing flows for user-friendliness and interoperability comes after reworking data (section 8.3) and operations (section 9.2) because we need individually optimized operations and their inputs, outputs, and errors to optimize flows (or use cases) that use them.

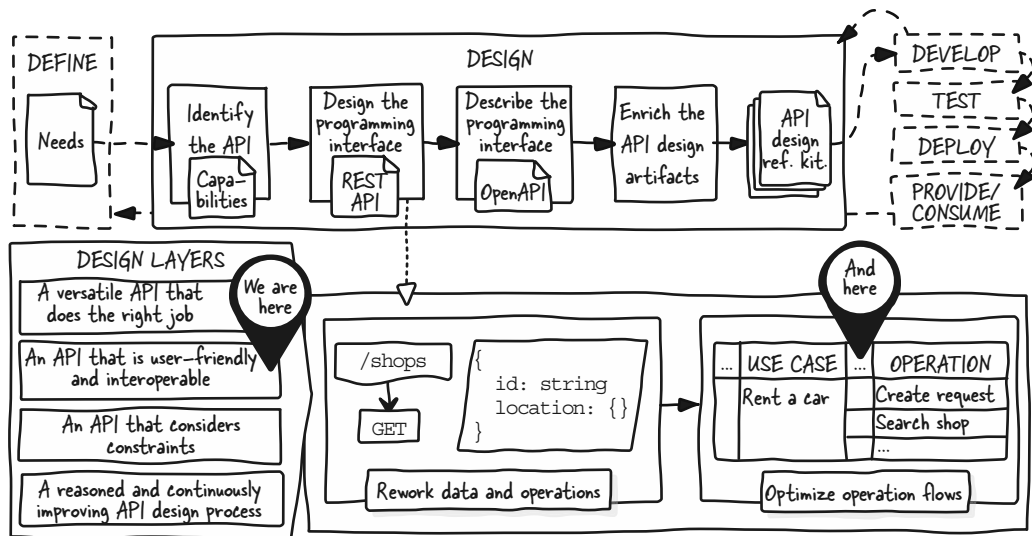


Figure 10.2 We optimize operation flows after reworking data and operations to make them user-friendly and interoperable.



NOTE This optimization will become easier with practice and the help of API design guidelines we'll craft along the way (section 16.3).

10.2.2 How to optimize flows

The example in section 10.1 provided clues about finding potential points to optimize and how to optimize flows to make them user-friendly and interoperable, enabling efficient development in various contexts and ensuring an excellent end-user experience. To find potential optimizations, we can

- Question the need for certain steps ("Search for cars" can be optional)
- Check whether errors can be prevented ("Date range above max rent days" error on "Rent car" operation)

- Check whether the processing of outputs to generate inputs can be prevented (calculating availability dates based on the maximum rental days returned by the search operation)
- Reconsider how data is saved during the flow (replacing “Create a request,” “Add shop to request,” and “Add car to request” with the final “Rent a car”)

To optimize a flow, we can

- Enhance operations with use-case-related features (adding the “Available within date range” filter)
- Create use-case-focused operations (“Search cars” versus specific “Search cars to rent”)
- Add use-case-focused informative or processed data (adding maximum rental days in search output data)
- Aggregate operations (merging “Search shops” and “Search cars in shop” into “Search cars across shops” or replacing the “Create” and “Validate” operations with “Rent a car”)

As is the case for data and operations, we must do this work with consistency and standardization in mind (file-upload pattern or retrieving API credentials).



NOTE The rest of this chapter illustrates how to identify opportunities for improvement and perform optimizations. Section 10.1.6 provides guidance on operation flow consistency and interoperability.

10.3 *Designing concise, error-limiting, flexible flows*

Checking whether an operation flow can be optimized is essential, whatever its final purpose: reading or saving data, renting a car, 3D-printing an object, or turning lights on and off. We must be mindful of limiting data processing between calls and limiting errors while ensuring that consumer UI flows aren’t overly constrained. However, we must also ensure that any optimization doesn’t come at the expense of clarity and scope of data and operations. This section first introduces the Banking API money-transfer use case that we’ll optimize, and then discusses the following:

- Detecting points to optimize
- Removing redundant calls
- Enhancing existing operations with use-case-specific features and data
- Creating use-case-specific operations from scratch or aggregating existing ones
- Creating flexible operation flows

10.3.1 *Introducing the money-transfer use case*

Figure 10.3 shows how to transfer an amount of money from a source account to a destination account using the Banking API. Consumers list source accounts with the “List accounts” operation to get all accounts the end user owns and exclude those with a status set to `BLOCKED`. They list third-party destination accounts that the end user

has preregistered using the “List beneficiaries” operation. Similar to source accounts, they get end-user-owned destination accounts. Finally, consumers call the “Transfer” operation with the end-user-selected source and destination and an amount provided by the end user.

USE CASE STEP	OPERATION FLOW STEP	INPUT	OUTPUT	ERROR
List owned source accounts	List accounts ◀----- Duplicates		Accounts (consumer must exclude BLOCKED) ◀----- Data processing	
List registered destination accounts	List beneficiaries		Beneficiaries	
List owned destination accounts	List accounts ◀-----		Accounts (consumer must exclude BLOCKED)	
Transfer money	Transfer	Source, Destination, Amount	Avoidable errors	Invalid source and/or destination, Amount not in acceptable range

Figure 10.3 The money-transfer use case or operation flow has some problems we can fix.

Transfers may fail for several reasons. The end user may not own the source account, or the state of the source and destination accounts may not allow money transfers (status set to BLOCKED). The destination account may not belong to the end user or match a registered beneficiary. Some source and destination combinations, such as transferring money from a savings account to a non-end-user-owned account, are unauthorized. The transfer amount cannot exceed the source balance, and the destination may impose minimum and maximum limits. Savings accounts require transfers greater than a specific value and have a maximum balance limit. The amount has to be below a particular limit based on the end-user profile and the total amount of money transferred in the last 30 days.

10.3.2 Uncovering operation flow problems

Any flow must be reviewed to detect potential improvements. We must address

- Repeating the same operation call needlessly
- Processing output to generate an input (or helping the end user decide on one)
- Preventing avoidable errors

With four calls, the money-transfer operation flow could be considered short and does not require optimizations. But developers may write error-prone code, and end users will be infuriated by easily avoidable errors. As shown in figure 10.3, distracted developers may call the “List accounts” operation twice and retrieve duplicate data. Developers must exclude accounts with a status set to BLOCKED to get eligible end-user-owned source or destination accounts. They must merge accounts and beneficiaries to get a valid list of destination accounts. To limit errors when calling the “Transfer” operation, they must add extra processing, such as determining acceptable source and destination

combinations and the amount range (hoping that's possible based on the data they can get). Without this processing, end users will be left with trial and error; this behavior will frustrate them, even with the best error feedback.

10.3.3 Calling read and search operations once

The same data can be retrieved multiple times if the same operation is called with the same parameters each time. We can keep only one call to retrieve the needed data. Such an optimization does not affect the operation design but does affect how we describe use cases (see section 2.5). As shown in figure 10.4, we can reduce the money-transfer flow from four calls to three steps because “List owned source accounts” and “List owned destination accounts” retrieve the same data from the “List accounts” operation. The “end-user-owned accounts eligible to transfer” data is used for different purposes but is still the same.

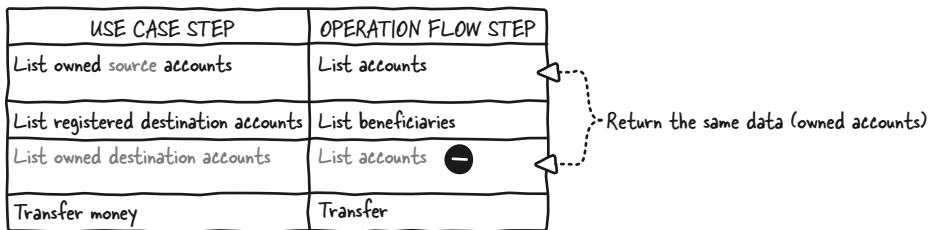


Figure 10.4 We can remove calls that return data that has already been retrieved.

10.3.4 Enhancing operations with use-case-specific features

To avoid having consumers process the output of one operation to create or help end users create the input of another operation, we can add use-case-specific features to the operation itself. For example, to avoid transfer failures, API consumers may exclude BLOCKED accounts from the response of “List accounts” to generate the source or destination accounts list.

As shown in figure 10.5, adding a “status not equal” filter to the “List accounts” operation would avoid consumer processing (for example, `GET /accounts?status=not:BLOCKED`). However, this filtering smells like the provider’s perspective and should be hidden in the implementation (see section 2.8). Also, it needs to be clarified whether the account status filter is related to money transfers. Instead, we may want to have `GET /accounts?transfer=enabled` return accounts that can be used for transfers.

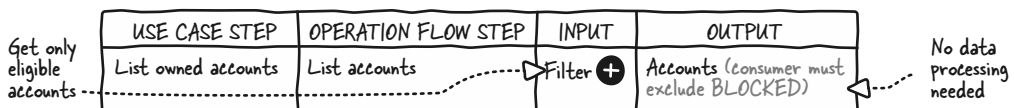


Figure 10.5 We add a transfer-related filter on “List accounts” to get only transfer-eligible accounts.



CAUTION Be careful with use-case-specific features, as essential capabilities can be hidden inside generic operations (see section 9.9). Adding a status filter makes sense because it is data from an account, but adding a filter that returns money-transfer-eligible accounts may go beyond the “Account” resource’s scope. Consider adding a use-case-specific operation instead (see section 10.3.5).

10.3.5 Adding use-case-specific operations

If use-case-specific data can limit errors and consumer processing but can’t be retrieved from an existing operation because it would go beyond the operation scope, we can create a use-case-specific operation. In section 10.3.4, we opted against adding the `transfer=enable` filter on `GET /accounts` to get eligible source accounts for a transfer. Instead, we can create a “List transfer sources” operation (`GET /transfer-sources`) that returns the subset of accounts eligible for use as a source for the “Transfer” operation (see figure 10.6). This operation, specific to the “Money transfer” use case, avoids the need for data processing. Also, developers can easily connect it to the `source` property of the “Transfer” operation.

USE CASE STEP	OP. FLOW STEP	INPUT	OUTPUT	ERROR
List eligible owned accounts	List accounts	Filter	Accounts	
List sources Replaces	List transfer sources		Eligible source accounts	
List registered destination accounts	List beneficiaries		Beneficiaries	
List eligible owned accounts	List accounts	Filter	Accounts	
List destinations for source Aggregates	List transfer destinations for source	Source	Eligible destination accounts or beneficiaries	Avoids
Transfer money	Transfer	Source, Dest, Amount		Invalid source and/or destination

Figure 10.6 The “List transfer sources” operation prevents adding money transfer parameters to “List accounts.” The “List destinations for source” use-case-specific operation avoids processing data of the generic “List accounts” and “List beneficiaries” operations.



TIP Such specific operations can be used for search operations whose search filters require predefined static or dynamic data. For instance, we can create a dedicated operation to retrieve the categories used when filtering transactions (see also section 5.5.1).

10.3.6 Combining operations into a use-case-specific operation

Achieving a use case may require aggregating data from different operations. We can create a use-case-specific operation for this purpose, but only if the aggregation aligns with the subject matter and is not specific to a consumer.

Now that we can determine the money-transfer source (see section 10.3.5), we must determine its destination. A “List destination accounts” operation that returns

money-transfer-eligible accounts owned by the end user is insufficient because we need to combine this list with the beneficiaries. The aggregated concept of a money-transfer destination (account or beneficiary) makes sense regarding the “Money transfer” subject matter. However, a “List transfer destinations” operation that returns both eligible accounts and beneficiaries is inadequate because not all combinations of source and destination are acceptable for the “Transfer” operation. To avoid developer guesswork, we can create a “List transfer destinations for source” operation that returns eligible destinations for a source (see figure 10.6).

We can represent it with `GET /transfer-sources/{accountId}/destinations`. It uses the interoperable `accountId` as an identifier for the source; this allows developers to get destinations without the need to list sources easily. This design doesn’t imply that `GET /transfer-sources/{accountId}` exists; we don’t need to get more information about a transfer source, so we don’t add it to the API.



CAUTION Beware of pushing the aggregation technique too far. Remember what we’ve learned about a perspective that is too consumer specific (see section 2.7) and the importance of data (section 8.7) and operation granularity (section 9.9). For example, reading all accounts, transfers, and end-user data with one operation doesn’t make sense from the perspective of “Money transfer” or “Banking” subject matter and is hard to comprehend and use.

10.3.7 Adding use-case-specific output data

As shown in figure 10.7, we can add use-case-specific data to operations involved in a flow to avoid data processing or errors—but only if doing so makes sense from a subject matter perspective. The “Transfer” operation’s input `amount` must be within a specific range depending on the source balance, destination type, and end-user profile. The source balance should not be exceeded, and the destination account has a maximum value based on the account type. If the transfer is to a beneficiary, the amount has to be below a specific limit based on the end-user profile and the total amount of money transferred in the last 30 days.

USE CASE STEP	OP. FLOW STEP	INPUT	OUTPUT	ERROR
List destinations for source	List transfer destinations for source	Source	Eligible destinations and amount ranges	Avoids
Transfer money	Transfer	Source, Dest., Amount	Ready-to-use data +	Amount not in acceptable range

Figure 10.7 We add ready-to-use amount ranges when listing destinations to avoid consumers calculating them and to limit the risk of an “Amount not in acceptable range” error when calling “Transfer money.”

We can’t provide raw data like balances, savings ranges, and transfer limits directly. It may make sense to add the balances and savings ranges to the responses of “List transfer sources” and “List transfer destinations for source,” but where should we put the

transfer limits? Adding them to “Read user profile” (if it exists) is beyond its scope. Adding a use-case-specific “Read end user transfer profile” is not an option either; consumers still need to do the math. We’d better let the implementation handle calculating the ready-to-use ranges for the source and destination combination and adding them to the response of “List transfer destinations for source.”

10.3.8 Avoiding constraining consumer flow

Despite being optimized with new features, operations, and data, an operation flow can still limit the possibilities on the consumer side and impose UI flows. The first table in figure 10.8 shows the improved “Money transfer” use case and its UI effects. Consumers only need to list transfer sources, list transfer destinations for a source, and perform the transfer without data processing. The risk of error is almost nonexistent because the data guides end users (eligible accounts, valid source and destination combination, amount limits). However, unexpected server errors or errors due to updated account balances during the process are still possible, although unlikely. This flow is better but not very flexible. End users must select the source before choosing the destination, which is necessary to input the amount within the appropriate range.

No data processing, limits errors		But no flexibility		New operations add limited flexibility		Full flexibility with aggregated operation	
OPERATION	UI	OPERATION	UI	OPERATION	UI	OPERATION	UI
List transfer sources		List transfer destinations		List transfer sources and destinations		List transfer sources and destinations	
	❶ Select source		❶ Select destination		❷ Select source ❸ Input amount		❷ Select source, destination, and amount in one or three steps in any order
List transfer destinations for source		List transfer sources for destination	Devs risk using List transfer sources				
	❷ Select destination		❷ Select source				
	❸ Input amount		❸ Input amount				
Transfer		Transfer		Transfer		Transfer	

Figure 10.8 The optimized money-transfer flow lacks flexibility, which can be fixed by adding new or aggregated operations.

The second table shows that trying to offer more flexibility with new operations like “List transfer destinations” and “List transfer sources for destination” could confuse developers. They may use “List transfer sources” and “List transfer destinations,” leading to selecting an invalid source and destination combination. Additionally, end users must still choose a source or destination before inputting an amount. To avoid all this, developers can call “List transfer sources” and then loop on each source to “List transfer destination for source.”

But we can do that work for them and have a single “List transfer sources and destinations,” as shown in the last table in figure 10.8. This allows developers to

build any UI flows in their application to gather the source, destination, and amount in any order.

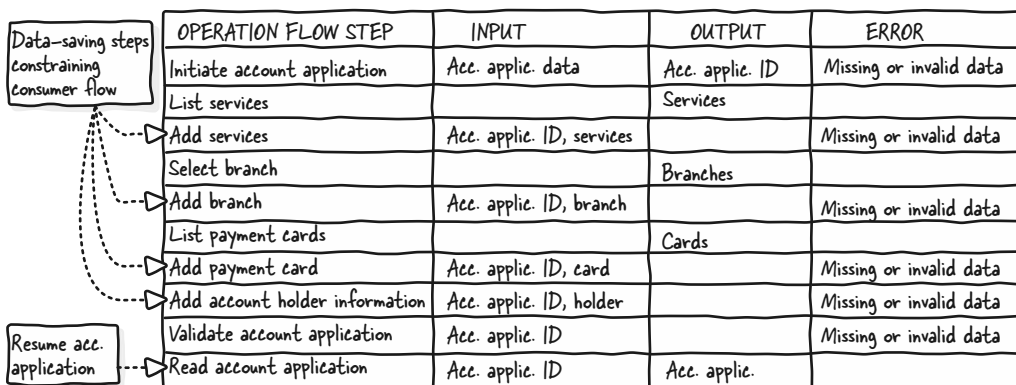
10.4 Designing flexible data-saving flows

Saving data at different steps in an operation flow is common, but doing so affects consumers and end users. An operation flow should allow consumers to save data as they need to, without restrictions. Such flexibility allows consumers to freely design their own flows, especially UI flows. This section introduces the “Open an account” flow we’ll optimize and discusses

- Understanding how saving data constrains consumers’ flow
- Enabling partial data-saving
- Carefully aggregating data saving operations
- Enabling partial data validation
- Separating validation from completion
- Enabling full and partial data saving flows
- Redirecting the consumer to the finalized resource

10.4.1 Introducing the “Open an account” use case

Figure 10.9 illustrates how to open an account using the Banking API. The operation flow includes initiating an account application to obtain an identifier and adding data related to services, a branch, a payment card, and an account holder. Executing “Add” operations in another order triggers an error. Consumers can resume from where they stopped using the “Read account application” operation. The “List” operations provide up-to-date data for the “Add” operations, preventing consumers from storing outdated data.



The diagram illustrates the operation flow to open an account. It features a table with five columns: OPERATION FLOW STEP, INPUT, OUTPUT, and ERROR. To the left of the table, a box labeled 'Data-saving steps constraining consumer flow' has dashed arrows pointing to the 'Add services', 'Add branch', 'Add payment card', and 'Add account holder information' rows. Another box labeled 'Resume acc. application' has a dashed arrow pointing to the 'Read account application' row.

OPERATION FLOW STEP	INPUT	OUTPUT	ERROR
Initiate account application	Acc. applic. data	Acc. applic. ID	Missing or invalid data
List services		Services	
Add services	Acc. applic. ID, services		Missing or invalid data
Select branch		Branches	
Add branch	Acc. applic. ID, branch		Missing or invalid data
List payment cards		Cards	
Add payment card	Acc. applic. ID, card		Missing or invalid data
Add account holder information	Acc. applic. ID, holder		Missing or invalid data
Validate account application	Acc. applic. ID		Missing or invalid data
Read account application	Acc. applic. ID	Acc. applic.	

Figure 10.9 The operation flow to open an account comprises several data-saving steps that constrain consumer flow. We can fix that.

10.4.2 Understanding how data-saving constrains consumer flow

Requiring consumers to execute data-saving steps in a specific order limits consumer flow and affects the end-user experience. In our example, the end user must select and input services, a branch, and a bank card in that specific order, followed by account holder information. Optimizing the UI flow to boost account opening rates requires modifying the API, which is tedious (see section 2.7.1). Furthermore, we don't collect vital data early on; we gather the information we need to get back to the applicant in the account holder step just before validation. If the applicant doesn't reach this step, we cannot contact them to finish the account application.

10.4.3 Enabling partial data-saving

Collecting minimal vital data first matters, but the rest can be collected as consumers need it. Instead of requesting data in a specific order (services, then card, etc.), we can let consumers submit any data in any order while enforcing obtaining an email address so they can get back to the end user. This partial data-saving offers flexibility for consumers to create a flow that meets their needs. However, invalid data will be rejected.

As shown in figure 10.10, consumers can execute “Add” operations at their convenience, such as calling “Add account holder” early to collect an email to get back to the applicant for an unfinished account application. However, to avoid constraints, no other account holder information should be required, allowing completion later through the “Update account holder information” operation. But collecting this email is crucial for the entire flow; we'd better do it when we initiate the account application. Additional information can be gathered with “Update account application.” This approach applies to other “Add” operations (services, branch, card) and corresponding “Update” operations. After initiating the account application, consumers can call operations in any order to collect data as they wish.

OPERATION FLOW STEP	INPUT	OUTPUT	ERROR
Initiate account application	Applicant email*, data	Acc. applic. ID	Missing or invalid data
Update account application +	Acc. applic. ID*, data		Invalid data
List services		Services	
Add services	Acc. applic. ID*, services		Missing or invalid data
Update services +	Acc. applic. ID*, services		Invalid data
...			
Validate account application	Acc. applic. ID*		Missing or invalid data

Annotations in the original figure:

- Non-blocking partial creation:** Points to the first three steps (Initiate, Update, List).
- Same for other data-saving steps:** Points to the remaining steps (Add, Update, Validate).
- Required data (*) is minimal:** Points to the input column.
- Fail only if invalid data:** Points to the error column for the 'Update account application' step.

Figure 10.10 By enabling partial creation and completing data with “Update” operations for account application, services, branch, card, and holder information, we allow consumers to build a UI flow that matches their needs.

However, “Add,” “Initiate,” and “Update” operations must not accept invalid data. For example, it must be impossible to add an invalid card type with the “Add payment card” operation or set an invalid address with “Update account holder information.”

Proceed as usual for HTTP representation and data modeling. Use POST to initiate or add, and PUT or PATCH to update. For example, POST /account-applications initiates an account application, and PUT or PATCH /account-applications/{account-ApplicationId} updates it. When using PUT, the request body contains all previously provided data plus the added/modified data (to replace the current account application). If we use PATCH, it contains only the added/modified data.



CAUTION Enabling partial creation and updating of resources has a drawback: knowing whether data is complete and which data is missing can be tedious. Section 10.4.5 discusses and solves this problem. Ensure that saving data in any order is not impossible because of business or implementation constraints; see section 14.1.

10.4.4 *Carefully aggregating saving operations*

As discussed in section 10.3.6, we can challenge operation granularity to simplify the flow if that makes sense for the subject matter. Figure 10.11 challenges the need for separate operations for each account application element. Combining operations around the account application concept can make sense. Consumers can initiate, update, read, and validate the account application, services, branch, card, and holder data with a single set of operations.

Update acc. application handles all elements				
OPERATION FLOW STEP	INPUT	OUTPUT	ERROR	
Initiate account application	Applicant email	Acc. applic ID	Invalid data	
Update account application	Acc. applic. data		Invalid data	
Validate account application	Acc. applic. ID		Missing or invalid data	
Read account application	Acc. applic. ID			

Figure 10.11 We merge all “Add” and “Update” operations of sub-elements (services, payment cards, holders) into the “Update account application” operation.

However, aggregating operations may not always be appropriate. For example, combining “List services,” “List branches,” and “List payment cards” in a single operation doesn’t make sense. They deal with unrelated concepts from the banking and account application subject matter perspectives and are separate information in the account application, and it will be complicated to handle filtering. Fortunately, keeping them separate doesn’t affect the consumer flow.

10.4.5 Smoothing validation and separating it from completion

It's essential to let applications consuming the API know whether they need to collect more data and allow them to let their end users do a final check before completing the flow. In section 10.4.4, we started with minimum data and updated it until completion. However, this approach has two drawbacks. After each update, consumers must call the "Validate" operation to determine the missing data, and end users can't double-check the complete and valid data they provided before confirming the account application. We can either add a "Complete account application" operation or use a query parameter flag for account application validation (POST /account-applications/{accountApplicationId}/validations?completion=false; the query parameter is a resource modifier: validation versus completion). Both require a call to "Validate" after each account application update to determine which data is missing.

A better approach would be to provide informative data within the account application, using the error feedback design from section 9.8. We still return a 2xx class code on "Initiate account application" and "Update account application" operations because an incomplete account application is not an error in that context. A `status` property and a `messages` list inform consumers about the account application status. The `status` can be `COMPLETE` (all required data is present) or `INCOMPLETE` (some required data is missing). The `messages` can use a data format similar to the one we put in errors on error responses. This way, consumers know precisely what's missing each time they save data. Once the `status` is `COMPLETE`, consumers can show all collected data to the end user for a final check before calling the `Validate account application` operation.



TIP Generalizing the data plus metadata structure introduced for list responses (section 9.6.7) would allow a clean separation between the actual account application data and the metadata, indicating what's missing or wrong.

10.4.6 Enabling full and partial data-saving flows

Consumers may or may not have end users. Either way, they may need to provide data incrementally, whereas others may not. Allowing for full (single-step) or partial (multiple-step) operation flow lets them choose the best approach for their use case.

We can consider adding the optional `completion` flag to the account application initiation to enable consumers to open an account in one call (POST /account-applications?completion=true; the query parameter is a resource modifier: partial versus complete account application); the operation errs if data is missing or invalid. This option could be helpful for a bank's partner using a batch application to open accounts. If it suits most consumers, we can make the one-call option the default (`completion true` by default). Therefore, a POST /account-applications?completion=false will start the multiple-step flow.



NOTE We're seeing many different design options. Check section 16.1 to learn not to feel overwhelmed and overcome doubts when deciding which to choose.

To be more explicit, we can differentiate a one-shot versus partial account application by creating `/account-applications` and `/partial-account-applications` resources and operations. Once a partial account application is validated (`POST /partial-account-applications/{accountApplicationId}/validations`), it becomes an account application (`/account-applications/{accountApplicationId}`). Check section 10.4.7 to see how to handle this switch. Consumers can perform a one-call account application with `POST /account-applications`.

10.4.7 Redirecting the consumer to the finalized resource

In section 10.4.6, we introduced the idea of working on an intermediary resource or operation dedicated to collecting data to create another resource; now, we need to connect the dots between the two. If the collected data is correct, `POST /partial-account-applications/{accountApplicationId}/validations` creates an account application that can be retrieved with `GET /account-applications/{accountApplicationId}`. The validation operation can return 201 Created with a `Location` header targeting the created account application (`/account-applications/{accountApplicationId}`) and its data.

We can also use the 3XX HTTP status class, which we haven't used yet. A 3XX class status indicates a redirection. The server tells the client that they should go elsewhere to get information. A 3XX response has a `Location` header indicating where to go; it has no body. Consumers may follow the redirection automatically if configured to do so or send a GET on the URL indicated in `Location`. The “Validate” operation can return 303 See Other with the `Location` header (`/account-applications/{accountApplicationId}`) and no account application data. It's up to consumers to decide whether to follow the redirection to get all data; the `Location` header may be enough for later use.

Summary

- An operation flow is a sequence of calls to API operations a consumer performs to achieve a use case.
- A user-friendly, interoperable operation flow uses user-friendly, interoperable data and operations, but it must also be seen as a whole.
- Design concise flows with minimum steps.
- Design flexible flows that can be used in various contexts and limit constraints on UI flows; flows should ideally enable starting from steps later than the initial one.
- Ensure that operations used within the flow meet user needs in the flow's context, helping to call other operations in the flow.
- Ensure the reuse of similar patterns across similar flows to create consistent and standardized flows that are guessable and easy to use.
- Optimize operation flows after enhancing data and operation user-friendliness and interoperability.

- To find potential optimizations, question the need for certain steps, check whether errors and data processing can be prevented, and reconsider how data is saved during the flow.
- To optimize a flow, add use-case-specific features or data, create specific operations, and aggregate operations.
- Ensure that any optimizations, especially aggregations, don't come at the expense of clarity and scope of data and operations.
- Enable partial data-saving to allow consumers to collect data as needed without forgetting to collect the minimal vital data when initiating the flow.
- Split validation and completion in a data-saving flow to inform consumers about missing data and enable end users to verify data before finishing.
- Allow for a full (single-step) or partial (multiple-step) saving operation flow that lets consumers choose the best approach for their use case.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 10.1

In a healthcare appointment scheduling API, the flow for scheduling an appointment with a doctor is as follows:

- 1 "List doctors" (GET /doctors); returns all doctors
- 2 "Get a doctor's slots" (GET /doctors/{doctorId}/slots); returns occupied and free slots in a doctor's schedule (date and time)
- 3 "Update slot with patient details" (PATCH /doctors/{doctorId}/slots/{slotId})

What are the problems with this flow, and how can you fix them?

Exercise 10.2

The Renovation Quotes API allows for a quick and raw estimation of costs and delays for a renovation project, as follows:

- 1 "Initiate quote" (POST /quotes); expects no data and returns a quote ID
- 2 "Indicate the number of rooms for a quote" (PUT /quotes/{quoteId}/room-count); fails if already set
- 3 "Add a room to quote" (POST /quotes/{quoteId}/rooms); expects no data, returns a room ID, and fails if the previously added room does not have all its data completed or the room count has been reached
- 4 "Set room size" (PUT /quotes/{quoteId}/room/{roomId}/size)
- 5 "Set room current condition" (PUT /quotes/{quoteId}/room/{roomId}/condition); fails if the room has no size or the condition has already been set

- 6 “Set room accessibility” (PUT /quotes/{quoteId}/room/{roomId}/accessibility); fails if the room has no condition or the accessibility has already been set
- 7 “Add service to a room” (POST /quotes/{quoteId}/room/{roomId}/services); includes painting, flooring, plumbing or electrical work, material quality, etc.; fails if the room has no size, condition, or accessibility
- 8 “Describe the project” (PUT /quotes/{quoteId}/description); fails if all rooms are not added to the list and fully described
- 9 “Estimate” (POST /quotes/{quoteId}/estimate); returns the estimate for cost and delay; fails without a description

What are the problems with this flow, and how can you fix them?

11

Designing user-friendly, interoperable APIs

This chapter covers

- Creating one or multiple APIs
- Naming APIs
- Enabling interoperable API browsing

We are not easing developers' work if an application must use different APIs but their search operations have completely different designs or similar resources have different IDs in different APIs. But before that, developers must find the right APIs and understand their capabilities and how to use them to meet their needs. But suppose an organization's information system is divided into two dozen blocks, each exposing an API just named "API" comprising hundreds of operations. Anyone (human or AI), including those who have created them, will have difficulty finding the right APIs and operations they need.

Ensuring the design of globally user-friendly, interoperable APIs is essential, as we've covered in previous chapters. But we must see an API as a whole and understand how to facilitate its discovery, understanding, and usage. This chapter examines what makes an API globally user-friendly and interoperable. We then explain how to choose between one or multiple APIs and how to select API names to help developers find and understand our APIs. Finally, we discuss enhancing API

interoperability and enabling runtime discovery by adding browsing capabilities and creating a hypermedia API.

11.1 What makes an API user-friendly and interoperable?

As shown in figure 11.1, we're almost done with the user-friendly, interoperable layer of the API design process introduced in section 8.1. After learning what makes data, operations, and flows user-friendly and interoperable, we can focus on the API level. A user-friendly, interoperable API has a clear purpose that meets focused needs and may help consumers discover available operations and relations at runtime.

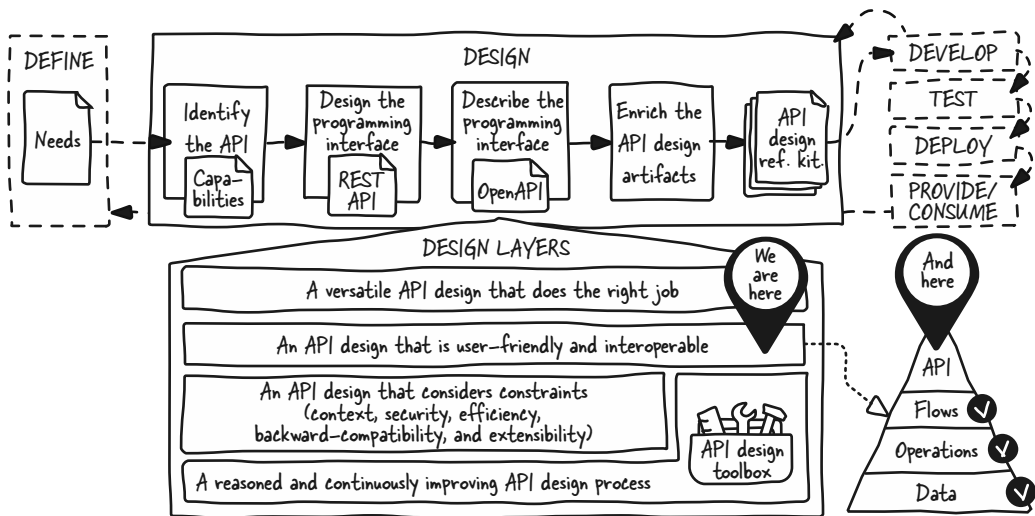


Figure 11.1 We're almost done with the user-friendly, interoperable API design layer.

11.1.1 Having a clear purpose that meets focused needs

A user-friendly, interoperable API uses user-friendly data (section 8.2), operations (section 9.1), and flows (section 10.1) that meet user needs. However, as is the case for operation flows, the sum of user-friendly, interoperable individual elements may not be a user-friendly, interoperable whole. So that they don't overwhelm developers and are easy to understand and easy to use, user-friendly APIs don't meet all possible needs; they have a clear, focused purpose.

For example, the Rent-A-Vehicle company has an information system covering accounting, HR, cloud infrastructure management, and car rental. Its developers will be overwhelmed if the company builds a single internal API named "API" that centralizes all possible operations across these domains. Creating an API per domain may still not be fine-grained enough to offer clear purposes. Although its name seems clear, a single Car Rental API covering the car rental domain actually covers different subdomains, such as rental management, customer management, and car fleet

management. Each can become a separate API with a clearer purpose that can be used in isolation or conjunction with others.

11.1.2 Enabling discovery and navigation

Using HTTP correctly and ensuring consistent and standardized data, operations, and flow design makes our APIs user-friendly and interoperable, simplifying discovery and navigation because consumers can guess how things work. But we may consider adding standard features to remove the guesswork and enable runtime discovery and navigation.

By using HTTP, a consumer of the fleet management API can know at runtime whether a specific car can be updated and how by verifying whether the `PUT` or `PATCH` method is available on the car resource. Using standard hypermedia data formats and their metadata, a consumer of the rental management API that retrieves the data of a car rental contract can determine the relationship between a contract and a vehicle and get a ready-to-use vehicle path instead of building it by concatenating a `vehicleId` to `/vehicles`.

11.1.3 How to create user-friendly, interoperable APIs

Based on previous sections, we can say that in addition to ensuring that we use user-friendly, interoperable elements when designing an API, we need to

- Ensure that the API is clearly bounded (do-it-all API versus fleet, rental, and customer management APIs)
- Ensure that the API has a meaningful name (“API” versus “Fleet management”)
- Consider making the API browsable (discovering whether an update is possible)

The rest of this chapter details when and how to consider these concerns.

11.2 Creating one or multiple APIs

It’s essential to analyze the set of operations we have designed to determine whether we’re creating one or more APIs. Although a single all-in-one API comprising all identified operations can meet user needs, it can be challenging for users, designers, and other stakeholders to understand the purpose of the API if it contains heterogeneous operations. Organizing the operations in different APIs can help better represent their intent.

This topic also has implications at the information system level. Organizing an information system or a subpart of it in different blocks is a discipline in itself that exists independently of API design. It is usually a task for architects (typically an enterprise architect in a big organization) or tech leads. However, we may have to work on such questions at a smaller scale as API designers. This section gives a few tips to achieve this task from the API design perspective, but consult with your tech leads and architects.



NOTE Security (section 12.3.5) and organizational or system constraints (section 14.1) can also drive the creation of one or more APIs. Typically, we’ll

separate business operations (such as transferring money or listing accounts) from system monitoring operations (health check operations). An API's granularity also affects its extensibility (section 15.6.1).

11.2.1 When to discuss API granularity

Regardless of the number of operations, we must always question whether one or multiple APIs are needed to avoid creating gigantic, meaningless APIs. We can investigate this question once we have a detailed view of the data, resources, and operations. That means we must wait until after optimizing the data, operations, and flows (section 10.2), as we may discover new resources and operations; what comes out of the API capability identification (section 2.1) may be incomplete.

If unsure about the division, we may keep everything in a single API and split it later, after expanding the API, when we have a better view of the domain(s) we are working with. This strategy has a minor drawback: consumers must slightly modify their configuration or code to call the new APIs (section 15.2.3); however, that's manageable.



TIP Remember that an API is an interface to an implementation. A single application may expose multiple APIs.

11.2.2 Identifying independent sets of operations

To evaluate whether the API needs to be divided, we can identify independent sets of operations and see how they are related. To achieve this, we use subject matter knowledge and operation flows.

Figure 11.2 shows the Banking API operations and how they relate once we optimize the flows. If we ask our SMEs, they can tell us the operations deal with three subdomains of their activity: account information (account, transaction, and holder-related operations), money transfer (beneficiary, source/destination, and transfer-related operations), and account application. Although resources may also give us some hints to identify these groups, we must look at operation flows to determine how they are related.

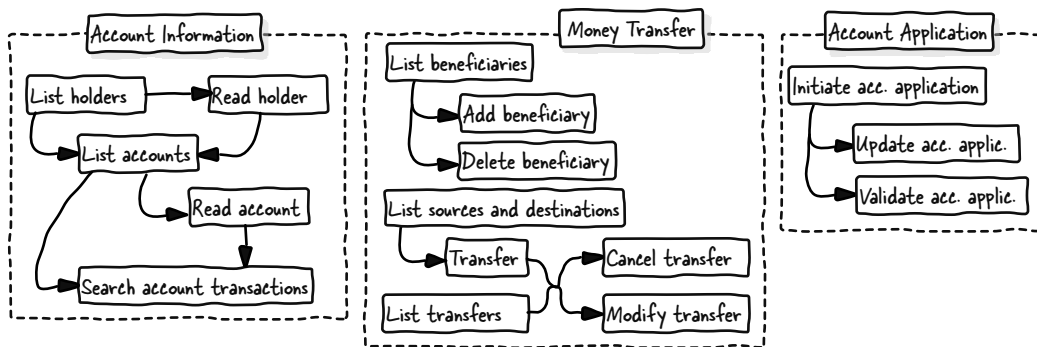


Figure 11.2 According to SMEs and confirmed by our analysis of the operation flows, the Banking API comprises three independent sets of operations ("Account Information," "Money Transfer," and "Account Application").

These three blocks can be seen by representing all operations in a diagram that connects them based on the flows identified in previous design process steps and listed in the API Capabilities Canvas. Divided this way, each sub-API can be used independently of the others. Consumers can trigger a money transfer without needing account information or account application operations, which means we can split the API into three parts.



TIP Consider using existing business data models to avoid unnecessary reinvention. Your architects or tech leads may have already divided the business into smaller domains that you can use to organize operations into different APIs. You can also use standard industry models such as ACORD in insurance, BIAN in banking, or ARTS in retail. However, although accurate, these enterprise models may not be user-friendly or suitable for public APIs or new practices.

11.2.3 Keeping in mind that sub-APIs can be related

Subdomains are not always clearly separated; they can share connections by using interoperable identifiers, which can sometimes result in the creation of different resources tied to similar business concepts.

Before optimizing the money transfer flow in section 10.3, we used “List accounts” to list sources for money transfers. Because of that, we might have been tempted to keep account information and money transfer operations in the same “Account” set of operations, as they are related to accounts, which wouldn’t be a clear way to organize them. However, the money transfer flow optimization highlighted the need for a money-transfer-dedicated “List source accounts” operation. “Account information” and “Source account for a money transfer” are different resources related to the concept of an account and share an interoperable ID: an IBAN (section 5.4.5).

This design allows us to create independent but related sets of operations with more precise purposes. “List accounts” and other account-related operations can go in an “Account Information” set, and “List source accounts” and “Transfer” go in a “Money Transfer” set. Consumers can trigger money transfers by solely relying on the “Money Transfer” set operations or by providing IBANs retrieved with the “List accounts” operation or from another data source they own; the origin of the IBAN is unimportant.

Domain-driven design bounded context

If you’re familiar with domain-driven design (DDD) methodology, you can use the concept of bounded context, which lets you split a domain model into smaller models that are independent but may share concepts. For more information, I recommend reading Martin Fowler’s post on the topic at <https://martinfowler.com/bliki/BoundedContext.html>.

11.3 Clarifying the API's purpose with its name

An API's name is crucial for setting the API's boundaries and helping potential users understand what the API does. It also ensures smooth evolution (section 15.6). This section covers when and how to choose an API name and add it to the API paths.

11.3.1 When to choose an API name

API names, like data names (see section 8.8.1), can vary during the design process. An API name is initially based on expressed user needs and is finalized only after we have discussed granularity and developed a detailed view of the API(s) needed to fulfill those needs (see section 11.2).

11.3.2 Choosing an API name

API names should focus on the subject matter domain, business concept, or use case. Avoid generic, project- or context-related, or meaningless names.

Generic names like “API” and “Web Service” don't tell what the API does and can't differentiate one API from another. Developers will be confused and have difficulty picking one “API” among many. Also avoid naming an API after a project, such as “New Banking Architecture”; such names often don't describe the API's purpose, and the API will outlive the project. In this example, “Banking Architecture” describes an internal reason or the context for creating the API, not its purpose. Once this architecture is set up, the project ends, but the API remains. Additionally, the API won't be “new” forever. Non-meaningful names like “Pink Fluffy Unicorn” are usually wrong, but they can be used for public or partner API branding. A name such as “Online Money Transfer,” although meaningful, should be used cautiously because it may limit the API's use cases; we designed it to be used by either a mobile application or a batch server application.

Plain, simple names like “Banking,” “Account,” and “Money Transfer” work well and describe the API's primary domain, resource, or use case. We can also suffix the name with “API”: “Banking API,” for example. However, a “Banking” API that only provides account information without other banking capabilities may confuse potential users, especially if it's public; a name focusing on the actual offered capabilities is best.



TIP Use what you learned about user-friendly names in section 8.8 to craft an API name.

11.3.3 Adding the API name to the API base path

To help users identify the API used, we can add it to the base or root path that prefixes all paths. For instance, the Banking API, now split into Account, Transfer, and Account Application APIs, can have paths like `/account/accounts`, `/transfer/beneficiaries/{beneficiaryId}`, and `/account-application/partial-account-applications`. If there's an API suffix (“Account API”), it is usually not present in the base path. As

illustrated in listing 11.1, in an OpenAPI document, the API name (appearing in `info.title`, “Account”) can be added to the API base path by defining a server and its `url` in `servers` (`/account`). This `url` is a prefix to all paths defined under `paths`.

Listing 11.1 Defining a server to have the API name in all resources

```
openapi: 3.1.0
info:
  title: Account          ← API name
  version: "1.0"
servers:
  - url: /account         ← API base path including
                           the API name
paths:
  /accounts: ...         ← Resource path
```



NOTE The base path is also a common location for indicating the API version (section 15.4.4). API gateways often rely on the base path to differentiate the API called and route traffic to the appropriate API server.

11.4 Enabling interoperable API browsing with HTTP and hypermedia APIs

We don’t have to guess URLs and what is possible when browsing websites. For example, on a banking website, the page providing information about a money transfer has a hyperlink to the source account and shows whether the transfer can be canceled. If so, there is a link or button to cancel it. We can provide similar features with HTTP and interoperable hypermedia API data formats, which help consumers discover possible actions and relations between resources and operations at runtime. Despite their usefulness, these features are rarely used in APIs, but it’s worth knowing they exist because they can be useful in some contexts.

This section discusses

- Listing a resource’s operations with the `OPTIONS` HTTP method
- Providing pagination, format, and other resource links with the `Link` header
- Using hypermedia formats for relations and actions
- Proposing a plain JSON and hypermedia format with content negotiation
- Ensuring that subject-matter data isn’t hidden in hypermedia metadata
- Considering browsing capabilities

Hypermedia API, REST, and HATEOAS

The term *hypermedia API* means consumers/clients can dynamically interact with the API/server via hypermedia metadata and a generic understanding of HTTP. It is described as “hypermedia as the engine of application state” in Roy Fielding’s dissertation, which defines the REST architectural style (www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). It is often referred to as the unpronounceable HATEOAS acronym.

11.4.1 Listing a resource's operations with the OPTIONS HTTP method

The OPTIONS HTTP method lets consumers know which HTTP methods are available on a resource without retrieving it. For example, a money transfer (`/transfers/{transferId}`) can be read (GET), modified (PUT), and canceled (DELETE). However, depending on its status, the transfer may not be modified or deleted. To check what is possible, consumers can call `OPTIONS /transfers/{transferId}`, which indicates the available HTTP methods in the `Allow` response header. In figure 11.3, the transfer can be read (GET) and canceled (DELETE) but not modified (PUT is absent).

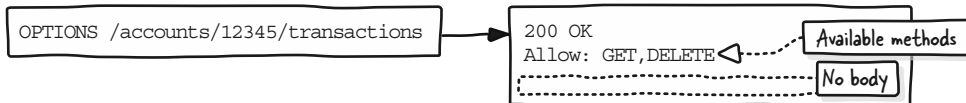


Figure 11.3 Using the OPTIONS HTTP method lets consumers know which operations are available on a resource.

11.4.2 Providing pagination, formats, and resources links with the Link header

Figure 11.4 shows how the `Link` response header, defined by the Web Linking RFC (<https://datatracker.ietf.org/doc/html/rfc8288>), can provide various types of links, including alternate formats, pagination links, and links to other resources. In section 9.7.1, we enabled the ability to request account transactions in CSV or PDF formats instead of JSON. The response to `GET /accounts/12345/transactions` can include a `Link` header listing these alternate formats. The `Link` header is a comma-separated list

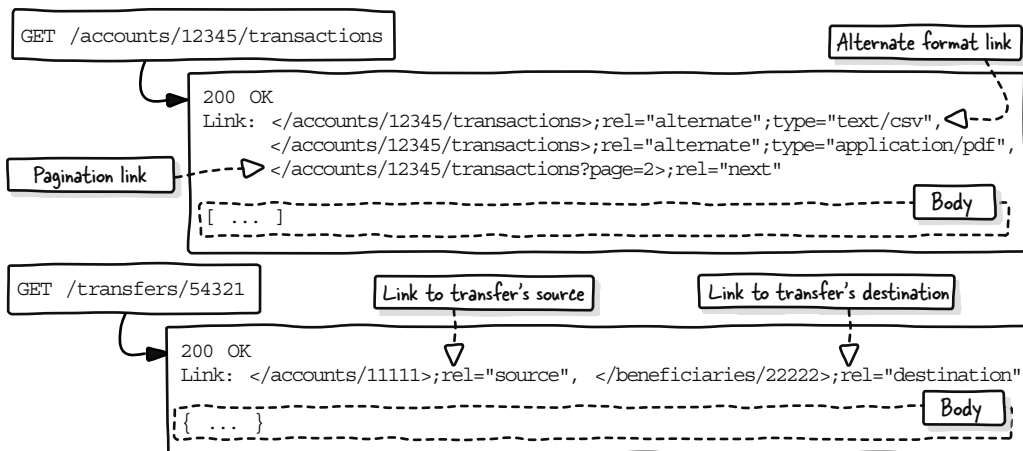


Figure 11.4 We can provide formats, pagination, and links to other resources with the `Link` header when listing transactions or reading a transfer.

of links, each composed of a URL enclosed in <>, rel, and optional type separated by ;. The rel describes the relationship between the current resource and the one targeted in the URL. The alternate relation indicates that the URL is an alternative resource representation. The optional type describes the media type of the targeted resource.

In section 9.6.6, we discussed transactions list pagination. Transactions pagination links can also be provided with a rel set to prev or next. The GET /transfers/54321 call shows that the Link header can contain any link, such as links to the source and destination of a transfer retrieved with GET.

11.4.3 Using hypermedia formats for relations and actions

Hypermedia APIs use metadata to create website-like features that describe links between resources; some may also describe possible actions on linked or current resources. To be effective, APIs must follow a standard format. The most popular is HAL, but JSON:API, Siren, and JSON-LD are also used.

Figure 11.5 compares money transfer representations in plain JSON, HAL, and Siren formats. How these formats describe relations is inspired by the Web Linking RFC, which defines the Link header described in section 11.4.2.

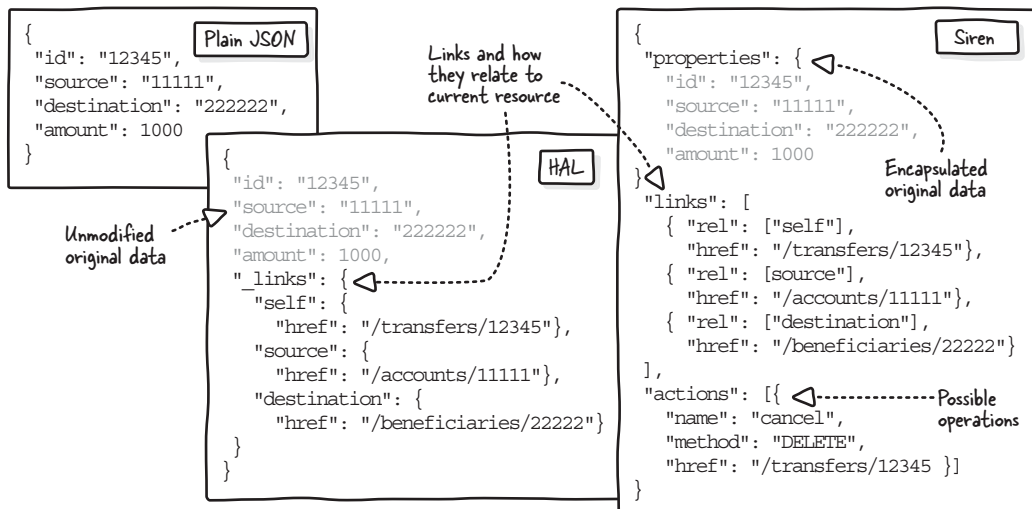


Figure 11.5 HAL and Siren propose standardized representations of links between resources. Siren adds the description of possible actions and separates original data and hypermedia metadata.

HAL integrates `_links`, a map of links, into the plain JSON data. Keys describe a relation; values are objects with an href targeting a resource. The `self` relation describes the current resource, and the `source` and `destination` target the source account and beneficiary. The rest of the data remains unmodified.

Like other formats, such as JSON:API and JSON-LD, Siren is a data format that modifies the original structure and stores the properties in properties. The links property is similar to HAL's `_links`, but it is an array in which each element has an href and a rel. Siren also defines an actions list to describe operations, such as to "cancel" the current transfer with an HTTP method and href (`DELETE /transfers/12345`). We can use the same mechanism to add pagination links like `prev` and `next` to collection resources.

Learn more about Hypermedia formats

This chapter only scratches the surface of hypermedia APIs. If you want to learn more about these formats, see https://stateless.group/hal_specification.html for HAL, <https://jsonapi.org/> for JSON:API, <https://json-ld.org/> for JSON-LD, and <https://github.com/kevinswiber/siren> for Siren.

11.4.4 Using content negotiation to select hypermedia or plain JSON format

We can offer hypermedia formats as alternate formats alongside our plain JSON data with content negotiation (see section 9.7.1). Hypermedia formats like HAL and Siren have their own media types: `application/hal+json` and `application/vnd.siren+json`. Consumers can request these formats by sending their media types in the `Accept` header instead of `application/json`. As we learned in section 11.4.2, we can advertise supported formats with the `Link` header.

11.4.5 Ensuring that subject matter data is always available

Although metadata provided through the `OPTIONS` HTTP method, `Link` header, and hypermedia format data can be helpful, we must ensure that essential information is not hidden within them. This keeps the API usable even when the browsing metadata and capabilities are ignored.

For instance, we can use `OPTIONS` or the Siren hypermedia format to describe the available operations on a resource. But if a piece of information is crucial, such as the possibility of deleting a transfer, we need to include it in the transfer's data with a cancellable flag. Similarly, although we may have a link to the source in HAL's `_links`, we shouldn't remove the source identifier from the data because it's essential information for consumers and end users. Conversely, the `Link` header may indicate available formats (CSV or PDF) on the transactions list, which shouldn't be included in the plain JSON response data. It would put us in the position of creating a custom, non-interoperable hypermedia format that nobody would use. When relying only on plain JSON data, consumers use their knowledge and documentation to know which formats are available on a resource, connect the cancellable flag to the delete operation, or build the path to the source resource (`/accounts/{source}`).

11.4.6 Considering browsing capabilities

Although hypermedia is in REST DNA (part of the uniform interface constraint introduced in section 4.8.1) and hypermedia APIs are powerful, they never caught on. Therefore, we should carefully consider whether to add such features to our APIs. As with any technology or design pattern, we must use it only if we actually need it.

The main benefit of hypermedia APIs is that they enable dynamic interactions between providers and consumers and may bring a certain level of standardization to our data and metadata organization. With a fully fledged hypermedia API and a client library that supports hypermedia formats, building a basic and even generic consuming application is easy and fast. It can also simplify the work of an AI agent because the agent can rely on runtime data instead of guessing which subsequent API calls could be made (which can be facilitated by a consistent API design). Additionally, hypermedia facilitates modifications, such as an API split (section 11.2.1), because consumers don't calculate paths but use the ones returned by the API (section 15.1).

If you're in a context where developers are eager to use these features and dynamic, generic, or AI-powered API clients could benefit from them, it may be worth investigating hypermedia APIs. If not, don't bother. No worries if you are unsure or the context changes; browsing capabilities can be added later through content negotiation (see section 11.4.4).



NOTE When selecting a hypermedia format, confirm the availability of client and server libraries that are compatible with your and your consumers' ecosystems. This will facilitate implementation and foster adoption.

Summary

- A user-friendly, interoperable API uses user-friendly, interoperable data, operations, and flows. It also has a clear purpose that meets focused needs and may help consumers discover available operations and relations at runtime.
- A single API can fulfill user needs but can be complicated to understand. After optimizing flows, evaluate whether one or more APIs are needed.
- To decide on the API split, identify independent sets of operations and see how they are related. The subdomains are not always clearly separated; they can share connections.
- An API name is crucial for setting the API's boundaries and helping potential users understand what the API does.
- When choosing an API name, focus on the subject matter domain, business concept, or use case. Avoid generic, project- or context-related, or meaningless names.
- Use the `OPTIONS HTTP` method to let consumers know which HTTP methods are available on a resource without retrieving it.
- Provide pagination, formats, and resource links on `GET` operations with the `Link` header.

- Use a standard hypermedia API format to provide interoperable metadata that enables navigation.
- Consider browsing capabilities only if they're actually needed, typically in a context where dynamic or AI-powered consuming applications can benefit from these features.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 11.1

A university is creating an API to support its operations. The following API operations have been identified:

- Search courses (GET /courses)
- Create a course (POST /courses)
- Read a course (GET /courses/{courseId})
- Update a course (PUT /courses/{courseId})
- Cancel a course (DELETE /courses/{courseId})
- Enroll a student in a course (POST /courses/{courseId}/students)
- Handle student withdrawal from a course (DELETE /courses/{courseId}/students/{studentId})
- Track course enrollment (GET /courses/{courseId}/students)
- Define course exam date (POST /exams)
- List exams (GET /exams)
- Add student (POST /students)
- Update student (PUT /students/{studentId})
- Delete students (DELETE /students/{studentId})
- Search students (GET /students)
- Record or update student grades for an exam (PUT /exams/{examId}/grades/{studentId})
- List grades for students, courses, or exams (GET /grades)
- Add employee time off (POST /employees/{employeeId}/time-offs)
- Update employee time off request (PUT /time-offs/{timeOffId})
- List employee time offs (POST /employees/{employeeId}/time-offs)
- List employees (GET /employees)

Do these operations fit in a single or different APIs?

Exercise 11.2

The PAX (Print Anything eXpress) company, which prints anything on any medium, has launched a ReGenesis initiative to rebuild all its internal applications. The company just finished refactoring the Zeus application that enables

- Calculating shipping costs for an order
- Generating shipping labels for an order
- Shipping an order
- Tracking an order shipment

PAX is considering the following names for this private API:

- Zeus API
- Logistics API
- ReGenesis Shipment API

Evaluate the pros and cons of each name. If none of these names are appropriate, propose one.

Constrained API design

We have designed a versatile, user-friendly, interoperable API that does the right job, streamlines consuming application development, makes developers feel fantastic, and enhances the end-user experience. However, this API design may be unrealistic and require adaptation to potential constraints. We must consider data and operation exposure and access, such as whether account reads should return sensitive information like payment card numbers, and who can read a specific account in a Banking API. We must avoid creating an API design that causes an excessive infrastructure load or drains smartphone batteries. We must integrate into our design the nature and usage of our data, the infrastructure, and business limitations: for example, we won't treat files like regular data, and the system or business may not be available 24/7. Finally, modifying an existing design involves caution; careless modifications risk making existing consumers crash, preventing our or our consumer's end users from accessing our services.

This part of the book focuses on the constraints layer of API design (section 1.7.3). Chapter 12 covers designing secure APIs; we discuss data sensitivity, secure operation behavior, data integrity, and controlling access. Chapter 13 focuses on efficient API design, discussing design optimization, cache, and processing multiple elements. Chapter 14 explores contextual constraints due to our data, architecture, or business. Chapter 15 discusses modifying an API, including how not to break consumers, versioning, and extensible design.

12

Designing a secure API

This chapter covers

- Exposing only the necessary data and operations
- Ensuring that implemented operations behave according to context
- Ensuring data integrity
- Preventing protocol or infrastructure-based data leaks
- Limiting access with security scopes
- Erroring securely

A versatile API that does the job and is user-friendly and interoperable is nice, but it's worth nothing if it's not secure. As APIs have grown in popularity, the number of API attacks has increased exponentially, making APIs the primary hacking attack vector in 2022. In 2023, 95% of organizations faced API security problems, such as distributed denial of service (DDoS) attacks, lack of authentication, API key leaks, shadow or zombie APIs that no one knows about, sensitive data overexposure, and business logic abuse. API security problems can cause reputational damage, financial losses, disruption to business operations, and data privacy threats.

API designers can't solve all API security problems, but they have a crucial role to play. API security must not be overlooked during the API design process, assuming that security experts will handle it later. How we design an API can significantly influence its security. For example, if we're not careful, a user buying products on our website via our Shopping API could discover the secret wholesale price (the discounted price we pay suppliers for bulk purchases) by inspecting network traffic through developer tools, accessing other users' orders, or changing the price of a product.

This chapter provides an overview of API security and how design contributes to it. Then we explain how to minimize the API surface by exposing only what is necessary, ensuring that operations behave according to the context, understanding how data can leak over HTTPS, and partitioning access to operations and data. Additionally, we discuss ensuring data integrity and how to design errors from a security perspective. In doing all this, we'll also learn how to describe the API in an OpenAPI document to ensure that implementation developers have all the information needed to develop a secure API.

12.1 Overview of API security

Securing any API is crucial. API providers must ensure that only relevant operations and data are accessible to authorized consumers and end users. Although APIs exposed on the internet are de facto at risk of being used and abused by anyone, internal nonsecure APIs become vulnerable once a network is breached. Additionally, inadequate security on internal APIs can lead to the accidental use of an API deployed in the production environment instead of the development environment, potentially corrupting the system.

To effectively contribute to creating a secure API, we must understand how API security works and how it connects with design. This section provides an overview of typical API security mechanisms and discusses common design-driven API security problems. We use the Shopping API example introduced when identifying API capabilities in section 2.3, which allows users to search for products and buy them.

12.1.1 What happens during an API call?

API security happens at three levels during an API call:

- Communication between the consumer and provider takes place over a secure channel.
- An API operation can only be called by known consumers with appropriate scopes or permissions and used by known end users (if any).
- The implementation performs the requested action based on the consumer or end-user context.

Figure 12.1 presents a simplified overview of what happens when an end user browses our shopping website, which uses the Shopping API secured with a mechanism similar to that used with OAuth 2.0 (www.rfc-editor.org/rfc/rfc6750.html) or OpenID Connect (www.openid.net/developers/how-connect-works/).



NOTE To learn more about API security (including the OAuth 2.0 and OpenID Connect frameworks), I recommend reading *API Security in Action* by Neil Madden (www.manning.com/books/api-security-in-action) and *Secure APIs* by José Haro Peralta (www.manning.com/books/secure-apis).

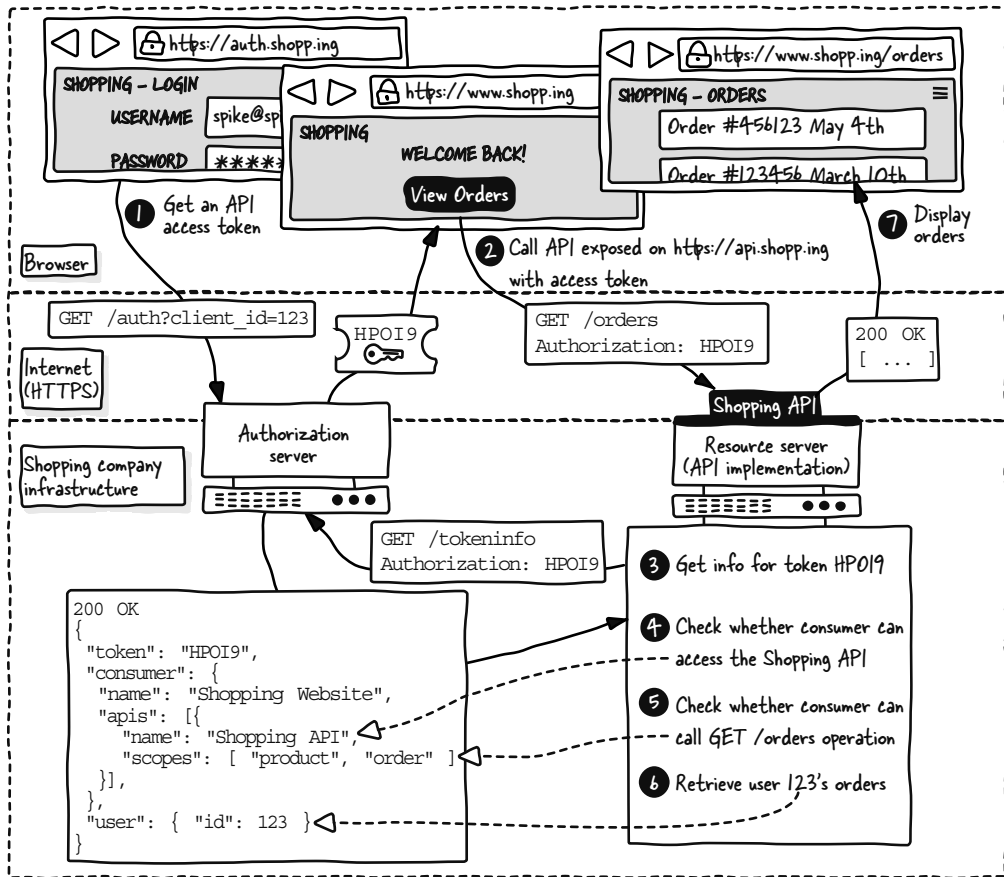


Figure 12.1 The shopping website gets an access token to call the Shopping API to retrieve an end user's orders. Based on the token, the Shopping API implementation verifies that the consumer has appropriate permissions before returning the user's data.

The process in figure 12.1 is as follows:

- 1 The website contacts the authorization server (AS), indicating its `client_id`, which identifies it. When the AS recognizes this ID, it authenticates the end user and returns an access token. This token allows the website to call the Shopping API exposed on the resource server (RS) in the end user's name.
- 2 To list the user's orders, the website sends a `GET /orders` request to `https://api.shopp.ing` with the standard `Authorization` header set to the token value.

The `https` URL scheme indicates that the HTTPS protocol is used. This secure version of HTTP secures communication between the consumer and the RS by encrypting requests and responses.

- 3 When the RS (and hence the API implementation) receives the request, it requests the information attached to the token by contacting the AS.
- 4 The RS checks the returned information to ensure that the consumer can access the Shopping API.
- 5 The RS also checks that the returned information has the `order` scope that allows calling the `GET /orders` operation.
- 6 The RS retrieves the orders for the user indicated in the token information.
- 7 The website displays the orders.



NOTE Like API design, API security applies to any API, including server-to-server communication, typically within a microservice architecture. The mechanism for obtaining the token to call an API and retrieve attached data may vary depending on the context. But the API implementation will always check this token to decide whether to respond, and the response will be based on information attached to this token.

12.1.2 *Uncovering design-related API security problems*

API security problems cover a wide range of topics beyond the perimeter of API design; for example, it's not up to us, API designers, to ensure that the authorization server securely stores access tokens or that our API servers can handle a DDoS attack. However, there are some essential API security problems we can help prevent during the design stage of the API lifecycle:

- Excessive data or operation exposure
- Business logic delegation
- Compromised data integrity due to concurrent updates or request replay
- Information leaks due to protocol or architecture
- Inappropriate consumer scope
- Insufficient implementation-level controls

An API may expose more data or operations than it should. For example, the data returned when reading a product with the Shopping API contains the wholesale price. Although it's not shown on the shopping website, it can be seen by looking at network traffic with developer tools in any browser. Also, an unexpected server error message may divulge critical infrastructure information.

An API may delegate business logic to consumers, risking the underlying system's integrity. As we saw in section 2.8.2, if updating the end user's address requires the consumer to deactivate the current address before adding a new one, there's a significant chance that the end user will end up without an active address or with multiple active addresses.

Data integrity may also be compromised. If the Shopping administration website updates a customer's VIP level and the selling website updates the same customer's email address with `PUT /customers/123`, the last update wins, and the VIP level or email address may be reverted. If a consumer thinks that an order (`POST /orders`) failed due to an absence of server response, they may retry the order and create duplicates.

Sensitive information can leak based on the architecture or how we use protocols. For instance, the “List orders” operation allows us to filter orders by payment card number (`GET /orders?card=12345`). This makes the sensitive card number visible in many logs.

A scope may cover an overly wide perimeter. The shopping website that targets end users buying products has `product` scope, which allows it to call any “Product” resource-related operations, including the “Update product” operation. This must be accessible only to the application targeting the administrator of our product catalog.

The implementation may also lack some controls. The Shopping API implementation may allow access to any order with `GET /orders/{orderId}` because it doesn't check that the provided `orderId` belongs to the end user.

Learn more about software and API security with OWASP

The Open Web Application Security Project (OWASP) is a worldwide not-for-profit organization focused on improving software security. Its website (<https://owasp.org>) provides many insights about software security, especially API security. I recommend reading the API Security Top 10, which describes the most typical API security problems (check the most recent edition in the top menu at <https://owasp.org/API-Security/>).

12.2 When and how to handle security during design

Now that we've looked at API security and how API design can cause security problems, we can discuss when and how to manage API security when designing an API.

12.2.1 When to consider security during API design

As illustrated in figure 12.2, we're now working on the constraints layer of API design introduced in section 1.7.3. API designers contribute to making an API secure (or insecure) throughout most steps of the design process. Security concerns affect capabilities identification, HTTP representation of operations, data modeling, and describing the API with the OpenAPI document. Although we must always consider security when designing an API, we must separate concerns to streamline our work and related discussions. We can work on security once we have designed a versatile API that does the right job, and after making it user-friendly and interoperable, which may add new data and operations; that way, we'll work on a complete design. However, security is not a one-shot concern; if any modifications are made, we must review them from the security perspective.

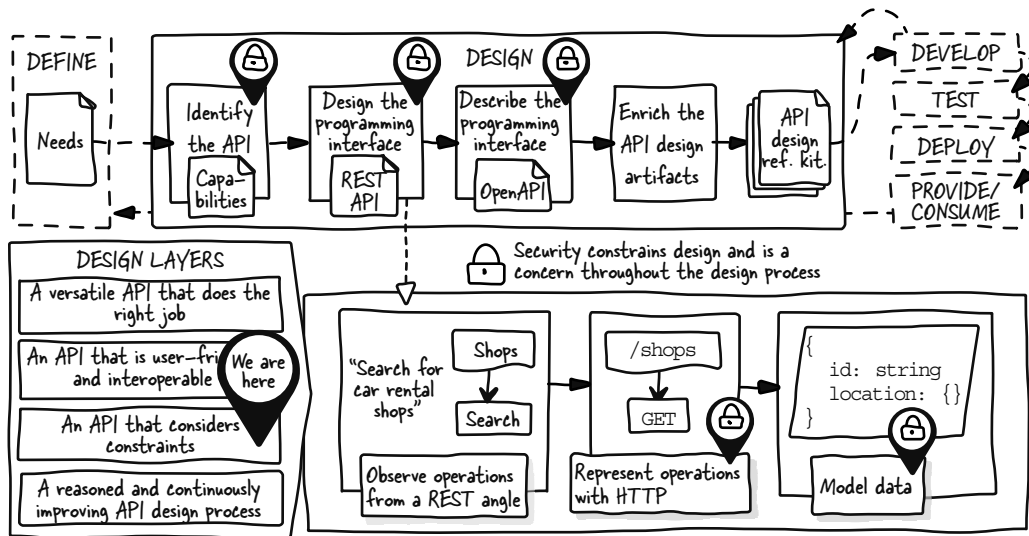


Figure 12.2 API security is a concern throughout the design process, affecting capability identification, HTTP representation, data modeling, and describing the API with OpenAPI.



NOTE Securing an API design will become easier with experience and the help of API design guidelines we'll craft to facilitate our work (section 16.3).

12.2.2 How API design contributes to API security

To avoid or at least limit the risk of the problems discussed in section 12.1.2, we must

- Be sure not to expose the provider's perspective
- Minimize the API surface
- Ensure data integrity
- Ensure that the implementation and its developers have all the needed information

Exposing the provider's business logic to consumers leads to nonsecure APIs that allow consumers to corrupt underlying systems, as we learned when identifying API capabilities (section 2.8.2). Focusing on user needs is a pillar of API security.

To minimize the API surface, we identify sensitive data and operations and adapt our design to expose only the necessary ones securely, based on who consumes the API and how it is exposed. We must also know how to use HTTP to avoid leaks and to design scopes that allow us to partition the API securely.

To ensure data integrity, we must condition updates to use fresh data and ensure that the implementation has the information it needs to perform such a check. We must also ensure that a call to an operation creating elements can't be replayed.

API security problems often arise because insufficient information is provided to implementation developers, and security concerns are ignored during design. To

prevent this, we thoroughly describe all security errors in the interface contract, identify security needs, adapt the design as necessary, and document security requirements for operations and responses.

The rest of this chapter discusses these concerns further, using the Banking API example from previous chapters. We'll learn to

- Expose only the necessary data and operations
- Ensure that operations behave according to the context
- Ensure data integrity
- Prevent protocol- or architecture-based security problems
- Limit access with scopes
- Err securely

12.3 *Exposing only the necessary data and operations*

A secure API design exposes a minimal surface that comprises only necessary data and operations. To achieve that, we can exclude unnecessary sensitive or non-sensitive elements or modify the design to handle sensitive ones securely. After discussing what we mean by sensitive data and operations, this section discusses how to achieve our objective by

- Challenging the presence of sensitive and non-sensitive data and operations
- Modifying data to make it less or non-sensitive
- Splitting an operation to separate concerns
- Separating sensitive operations in dedicated APIs

12.3.1 *What are sensitive operations and data?*

Sensitive data is information that, if leaked, can lead to security threats or loss of advantage for the organization that owns the data and related individuals. A sensitive API operation carries similar risks as it manipulates sensitive data or provides access to critical business or technical processes. Let's look at a few examples related to the Banking API.

Payment card details like the number and expiration date are highly sensitive; leaking them can lead to fraudulent payments. Account owner details such as name, phone number, and address are also sensitive; leaking this information can lead to scams and identity theft. Additionally, such personally identifiable information (PII) is subject to regulations like the European General Data Protection Regulation (GDPR) and the Privacy Act in the United States, which (rightly) impose high fines for abusing or failing to protect this data.

Listing banking services to choose from for an account application may seem harmless, but revealing their internal costs can be risky. Such business-critical information should only be accessible to bank employees who require it. Also, some money transfers are unusual and require extra validation from specific bank employees. Inadvertently letting other users, such as account owners, perform this validation creates a security vulnerability.



CAUTION As API designers, we cannot determine sensitivity on our own. Consult with your Chief Information Security Officer (CISO), Data Protection Officer (DPO), legal department, or another qualified expert. Sensitive data and operations are often already identified, so gathering that information before designing the API streamlines the process. Sharing an exhaustive OpenAPI document with security stakeholders will also help.

12.3.2 *Challenging sensitive and non-sensitive data and operations*

When identifying sensitive data or operations, our first reaction should be to question their presence. For instance, we can challenge adding the list of payment cards to an account data model or expose the unusual transfer validation operation on the internet or to partners. However, the presence of this data can be legitimate in context or due to requirements. These elements may make sense in a private API exposed on an internal network or in a partner API exposed to trusted third parties that need payment card information to perform specific tasks. In those cases, we must take the steps described in the following sections to minimize the risks.

Non-sensitive data and operations should also be questioned from a broad security perspective. Minimizing the software and API surface is recommended to limit security risks. The most secure data or operation is the one that doesn't exist. There must be a genuine reason for exposing non-sensitive data or operations via an API. Focusing on user needs (as we learned in part 1 of this book) helps create APIs that expose only the necessary elements. For example, if no use case involves reading an account owner profile, we must not include this operation in the API.

12.3.3 *Modifying data to make it less sensitive or non-sensitive*

We can modify the model of the sensitive data we wish to keep to make it less sensitive or non-sensitive by removing or replacing elements. Figure 12.3 illustrates these options.

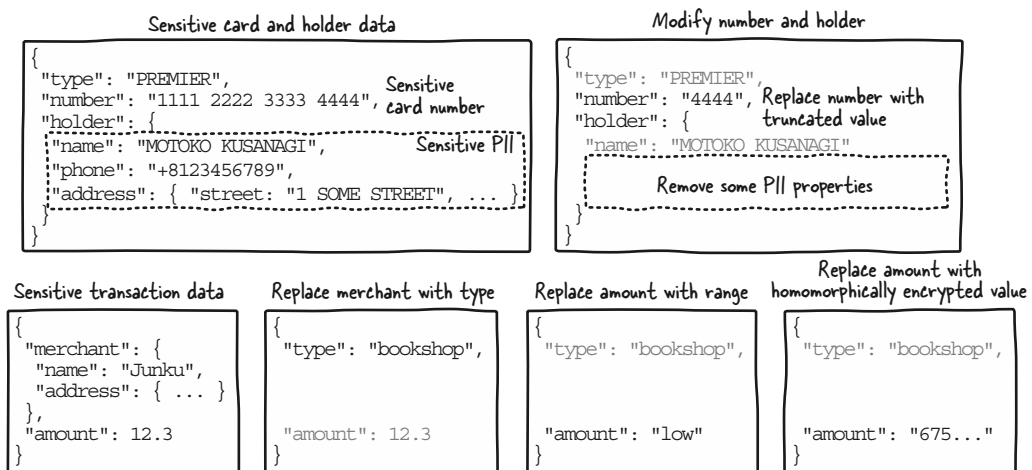


Figure 12.3 Card, holder, and transaction-sensitive data become less sensitive by removing or encrypting data.

When reading an account, we decided to return the list of payment cards attached to it. A card has a type, number, and holder. We replaced the sensitive card number value (1111 2222 3333 4444) with the last four digits (4444) to make it less sensitive. The holder data properties are sensitive when all together. We kept only the holder's name and removed their phone number and address to fix this.



CAUTION Always check whether any security standards provide recommendations when handling sensitive elements. For example, when processing payment card data, comply with PCI security standards. Visit www.pcisecuritystandards.org for more information.

The transaction amount, merchant name, and address may be considered sensitive when sharing account transaction data with a third party for statistical analysis. To address this, we can replace the merchant information with a category ("restaurant" or "bookshop", for example). Similarly, we can replace the amount with a fuzzy range ("low", "medium", or "high", for example). However, this method limits computation. Alternatively, we can use homomorphic encryption to hide actual amounts while keeping them computable. Our partner can perform calculations on encrypted amounts and return the result to us, and we can decrypt it to get the actual result. Check <https://homomorphicencryption.org> for more information. Also see section 12.6.4, which discusses other uses of data encryption.

12.3.4 Splitting an operation to separate concerns

An operation can have a purpose that addresses sensitive and non-sensitive concerns. As we learned in section 9.9.2, splitting such an operation can help make the API more secure. In section 12.3.1, we discovered that some money transfers may require extra validation from a bank employee. Suppose this validation is performed via a status modification with the "Update a transfer" operation, which also allows the modification of the execution date or amount. In that case, the operation mixes sensitive and non-sensitive actions. We can create a dedicated "Validate unusual transfer" operation, which separates concerns, makes the sensitive action more visible, and simplifies access control.



NOTE We may also consider letting the implementation tweak what can be done (modifying the date of a transfer, for example) or which data is returned (merchant address in a transaction) at runtime by using consumer or end-user contexts (section 12.4.4) and scopes (section 12.8.5).

12.3.5 Separating sensitive operations in dedicated APIs

Splitting an API to separate sensitive and non-sensitive operations reduces the risk of giving sensitive access to the wrong consumers or users. However, as we learned in section 11.2, ensuring that the resulting APIs are helpful and make sense is essential. Additionally, such security-driven splits can highlight gaps in our needs analysis and help fix them.

The Money Transfer API provides operations such as “List source and destination,” “Transfer,” “List transfers,” “Modify a transfer,” “Cancel a transfer,” and “Validate unusual transfer.” The last one is sensitive and should allow access only by selected bank employees. We should move it to a dedicated Money Transfer Admin API.

However, the new API doesn’t work alone and requires the Money Transfer API’s “List transfer” operation to validate an unusual transfer. To address this, we perform a quick needs analysis and search the API Capabilities Canvas for use cases that use the “Validate unusual transfer” operation. We identify the necessary operations to make the API a standalone solution.

The resulting APIs may share identical or slightly different operations. In both APIs, the “List transfer” operation returns the transfer the end user can see. A bank customer will see the transfer related to their bank accounts. A bank employee will see the transfer related to the bank accounts they manage, which require their validation. However, the data returned may differ; the admin version can contain extra data such as the identity, IP address, and user agent of the user who initiated the transfer. Additionally, some operations may exist only on one side. For example, the admin API doesn’t have the “List source and destination” operation.



NOTE Remember from section 5.4.5 that the same concept may be represented by different resources and thus different data models depending on the context in which it’s used.

12.4 *Ensuring that implemented operations behave according to context*

Despite only integrating the necessary operations and data in our design, data leaks and security-related errors can happen at runtime because the implementations of operations don’t behave according to the context. Typically, Alice shouldn’t be able to access Bob’s accounts unless Alice is a bank advisor managing them. To prevent this, we can

- Describe what consumers and end users can see or do
- Narrow access by design

12.4.1 *Describing who sees or does what*

API operations must behave according to the security context attached to the access token, which indicates the consumer and end user. But the resource paths, HTTP methods, and data alone may not convey the expected security-related checks and data processing. We must provide implementation developers with explicit, clear information about expected runtime security measures to avoid misses or hazardous guesswork in the implementation, which can compromise API security. The following sections cover the following:

- Describing what list or search operation returns
- Describing how inputs narrow access
- Describing all expected implementation checks and behaviors

12.4.2 Describing what list or search operations return

A typical runtime security problem is a list or search operation returning more data than it should. To clarify expectations, we can use the description of the operation or its 2XX response in the OpenAPI document.

Our OpenAPI document defines the “List accounts” operation as `GET /accounts`, which returns a list of `Account` elements that include `iban`, `balance`, and other carefully selected properties. However, without more precision, the implementation risks exposing all bank accounts to any application calling the operation. This may not be a problem with an internal application that is performing global financial checks. But it would be a colossal data leak for the mobile application used by account owners, who are supposed to see only the accounts they own. Fortunately, most developers will ask how they should filter the accounts, but some may not or may make wrong assumptions based on incomplete information that we provided. The following listing shows how we can prevent such problems by clarifying what is returned in the 200 response description of the “List accounts” operation in the OpenAPI document.

Listing 12.1 “List accounts” response description

```
openapi: 3.1.0
...
paths:
  /accounts:
    get:
      summary: List accounts
      ...
      responses:
        "200":
          description: |
            The returned account lists vary depending on the
            ➤ security context:

            - Account owner user: returns accounts owned by the end user.
            - Bank advisor user: returns accounts owned by the account
              ➤ owners that the bank advisor manages.
            - Partner application: returns accounts linked to partner
              ➤ identifier.
```



TIP All OpenAPI description fields support Markdown format, allowing clear text formatting.

12.4.3 Describing how inputs narrow access

Narrowing access according to the consumer or end-user context is a concern for any operation. It is done based on what the implementation knows about the consumer or end user and all input data.

For instance, `GET /accounts/{accountId}` must only return the account to account owners who own it or bank advisors who are managing the account’s owners.

A call to `POST /owners/{ownerId}/addresses` must be checked similarly. A `POST /transfer` has a source account. If the end user is an account owner, the implementation must check that the end user owns this account. If the end user is a bank advisor, the implementation must check whether this account belongs to an account owner managed by this bank advisor. See section 12.10 to learn how to handle related errors.



CAUTION Such controls must be performed even when using opaque UUIDs (`ff0d991d-6fcf-4349-baa2-6875a69790f8`); opacity isn't security. Input data must always be checked against the security context data, which is the only input data the implementation can trust.

12.4.4 *Describing all expected implementation checks and behaviors*

As seen in previous sections, the consumer's or end user's context attached to the token contains information that allows the implementation to identify them and narrow their access. It also contains their roles or permissions, which may grant them access or prevent them from seeing or doing something. Clarifying any related checks and behaviors the implementation must perform is essential.

For example, some bank advisors can perform a transfer of more than \$100,000 to an external account, but others are not allowed to. If they try, the implementation must return an error. The "Money transfer" operation description can state this limitation. Account owners can see all transaction data, including the merchant address (section 12.3.3), whereas bank advisors can't. Therefore, when an account transaction is listed for a bank advisor, the merchant data can be stripped out. The description of the `merchant` property can state, "Only returned to account owners."



NOTE Use the operation, response, and property descriptions in the OpenAPI document to exhaustively describe who sees and does what.

12.4.5 *Narrowing access by design*

When clarifying and discussing who sees or does what, we must consider adapting the design to return only elements needed to achieve a specific objective, although the consumer or end user could access more. This is not a strong security measure, but it can limit errors that would harm our company, partners, and customers.

When a bank advisor executes a money transfer on behalf of an account owner, it's best that `GET /source` retrieves the possible transfer sources for that specific account owner instead of sources for all the account owners the advisor manages. This ensures that the source is owned by the account owner who requested the transfer. To accomplish this, we can add the owner ID to the resource path and modify the operation into `GET /owners/{ownerId}/sources`.

To avoid bothering consumers used by account owners with extra input, we can accept the magic owner ID `me`, leading to `GET /owners/me/sources` or keep `GET`

/sources. Either way, check section 12.8.4 to see how to use scopes to ensure that each application type only accesses the proper “List sources” operation.



NOTE To prevent accidents, discuss with your security experts whether it’s possible to have a security token indicating that the user (human or application) is acting on behalf of someone: for example, that Alice, the bank advisor, is performing a money transfer on behalf of Bob. The implementation can perform stronger checks based on the information attached to the token.

12.5 Ensuring data integrity

Ensuring that consumers and their end users only do what they are supposed to do does not ensure total data integrity. Data integrity can be compromised due to the replay of a request; duplicates can be created. There’s also a risk of losing past modifications on updates because an authorized consumer used old data. This section discusses corrupting data with regular API calls and how to prevent it by

- Correctly implementing HTTP methods
- Preventing request replay
- Enabling conditional updates



NOTE This section discusses design and implementation considerations, which are essential to understand when designing an API and describing the expected behaviors. These generic behaviors can fit into API guidelines (section 16.3).

12.5.1 Corrupting data with regular API calls

Data corruption may occur when replaying a request and when updating data. Consumers or HTTP intermediaries may encounter unexpected problems due to infrastructure or network glitches, such as 500 errors or no response. Replaying an unexpectedly failed request, such as a money transfer, can be risky for data integrity, as the server may already have fulfilled it and would thus send money twice.



CAUTION Data corruption can have broader consequences that reverting data in a database may not solve.

Additionally, we only need to update based on stale or outdated data returned by a previous read to corrupt data, as illustrated in figure 12.4. Suppose two consumers read an account owner with `GET /owners/123`; one modifies the email address and then the other the job title with `PUT /owners/123`. The email address change is reverted because the second update is based on outdated data. The same applies to a `PATCH` request resending nonmodified data.

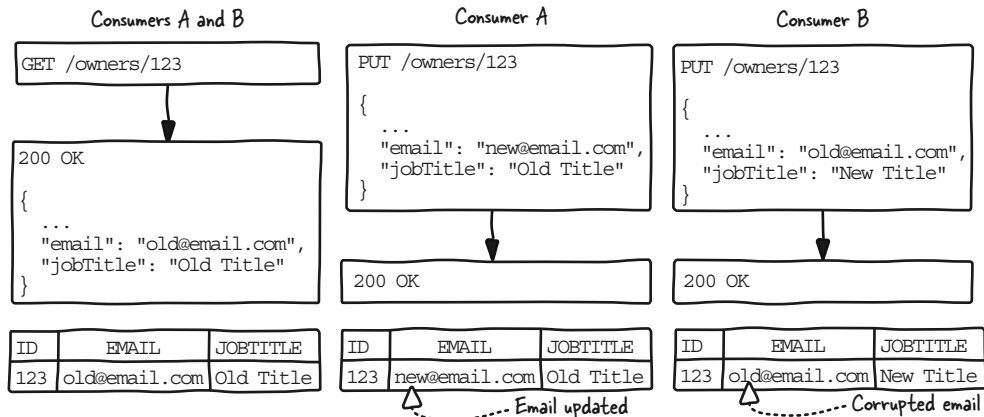


Figure 12.4 An update based on outdated data corrupts data.

12.5.2 Correctly implementing HTTP methods

We can rely on HTTP methods to determine when to replay a request without jeopardizing data integrity. Depending on the method, an HTTP request can be safe and idempotent.

A safe HTTP request does not alter the server state and is used for read-only operations (GET), but the server may log calls or update stats. We can safely replay a GET /accounts/12345 request; the server will add a new log line.

HTTP idempotency means identical requests produce the same server effect. GET, PUT, and DELETE are idempotent, whereas POST and PATCH are not. For example, repeatedly calling DELETE /files/12345 results in the same server outcome: the file is deleted after the first call, and subsequent calls may return an error if it's not found, but the server state remains unchanged. In contrast, replaying POST /transfers creates multiple money transfers, changing the server state each time and emptying the end user's bank account.



CAUTION Replaying non-idempotent POST and PATCH jeopardizes data integrity. However, replaying an idempotent PUT doesn't ensure data integrity, as data may have been modified; see section 12.5.1.

Therefore, we can only safely replay GET and DELETE requests if their implementation is idempotent.

12.5.3 Preventing request replay

We can identify unique requests with a technical ID or based on data to prevent a server from reprocessing a replayed request, typically POST, PATCH, or PUT. We can add a request header with a consumer-generated unique identifier, typically a UUID, to block the reprocessing of requests. That way, a consumer can automatically replay a request safely.

An IETF Internet-Draft (<https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-idempotency-key-header-06>) suggests naming this header Idempotency-Key; we could also use Transaction-Id or Request-Unique-Id, for example. When receiving a POST /transfers request, the server checks whether the consumer has already sent a request with the same identifier and returns a 409 Conflict status if so.

We may also consider adding business-level checks based on request data. End users of our mobile application may retry money transfers when the UI shows an unexpected technical error; the server may or may not have processed the request. The server may view these retries as separate instances if each attempt has a new idempotency key. But if the server detects an identical transfer (same amount, source, destination) within the last minute, it can respond with a 409 Conflict. The duration for blocking duplicates varies by context. However, if the consumer or end user wishes to repeat the transfer, resend POST /transfers with a Confirm-Duplicate: true header to force it.

12.5.4 Enabling and enforcing conditional updates

We can use HTTP conditional requests to prevent data corruption on an update. It's a standard HTTP mechanism that lets us send a request that says, "Execute this request if a condition is met." In our current context, it ensures that the update is based on fresh data, as illustrated in figure 12.5.

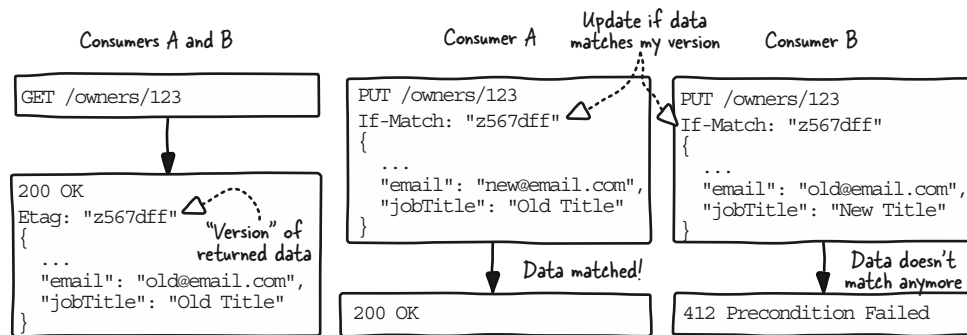


Figure 12.5 By returning the resource data version in an Etag header on reading, we can condition updates with the If-Match header and ensure that data stays uncorrupted.

The "Read owner" operation returns an Etag: z567dff header. The Etag standard header contains an entity tag (z567dff), which identifies the data's version based on changes over time and content negotiation; JSON and XML representations of the same data have different entity tags. An Etag can be based, for example, on a hash of the modification date or all the data; it's up to the implementation to decide what's best.

Consumers send this value in an If-Match: z567dff request header on the update operation to make it conditional. Based on this, the implementation checks before

performing the update that no modification has been made since consumers retrieved the data. If the resource data doesn't match the `Etag`, it means the data has been modified since it was read. In that case, the operation returns a `412 Precondition Failed` error. Use the usual error format to inform the consumer, such as an application problem. The consumer must read the owner to get the latest version to fix this and perform the update.

Alternatively to `Etag` and `If-Match`, the read operation may return a `Last-Modified: Sat, 29 Jun 2024 07:28:00 GMT` header that contains an HTTP date. Consumers can use this date with the `If-Unmodified-Since` header to condition their update request. Check the Conditional Requests section of RFC 9110, HTTP Semantics (<https://www.rfc-editor.org/rfc/rfc9110.html#name-conditional-requests>) for more information. To ensure the safety of all updates (`PUT` or `PATCH`), we may consider always marking the `If-Match` (or `If-Unmodified-Since`) request header as required in our OpenAPI document.



NOTE Section 13.4 discusses further how consumers can safely and efficiently cache data and use conditional requests to update their cache only when necessary. Ensuring that we systematically design updates to be conditional is a typical topic to include in API design guidelines (section 16.3).

12.6 *Avoiding protocol- or architecture-based security problems*

To securely design an API, it's essential to know how the protocol we use for our API and the architecture used to expose and consume it can lead to data leaks. This section first discusses what may not be secured on an API call over HTTPS and then explains the following:

- Dealing with sensitive search parameters in URLs
- Dealing with sensitive resource IDs in URLs
- Integrating data encryption or signing in the design

12.6.1 *What may not be secured on an API call over HTTPS*

Understanding what HTTPS secures and how API data can leak on the consumer and provider sides is essential to adapting our design and limiting sensitive data leaks. Figure 12.6 illustrates a typical architecture with the Banking API exposed via an API gateway, an API-security-focused HTTP proxy, and consumed by a server, web, or mobile application. The architecture on the consumer side can be unknown.

No one can read the encrypted data on the secured HTTPS wires between the consumer and the API gateway, as well as the API gateway and the implementation. However, these components between HTTPS wires may not be secured. Applications that expose or consume APIs often log calls, potentially exposing URLs and request/response bodies (although it's less common due to storage limitations). Also, although we may know what happens on the provider side, an API call and its data can pass through

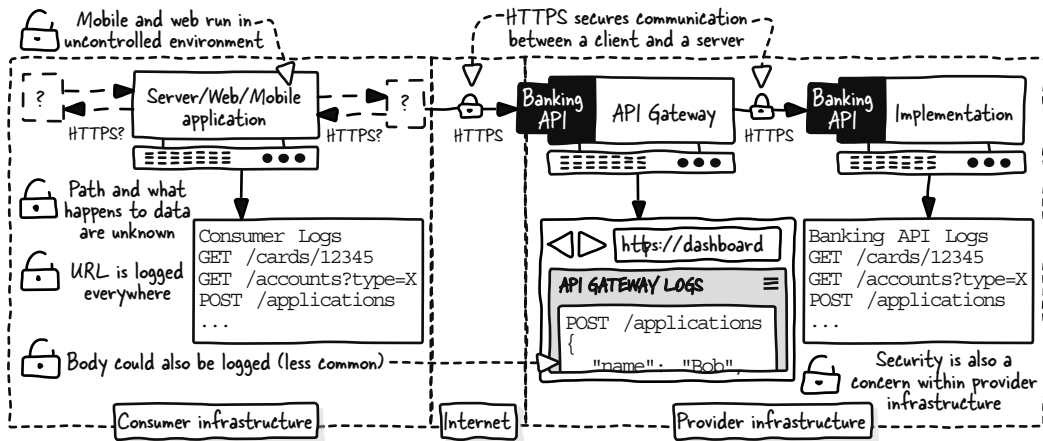


Figure 12.6 HTTPS secures communication between client and server. The URL and even the body can be logged in many places on the consumer and provider sides. How security is handled on the consumer side is unknown.

multiple intermediaries, some of which may be uncontrolled. Our very own banking mobile application runs on uncontrolled devices owned by our customers.

To mitigate these protocol- and architecture-based security risks, an API must

- Never have sensitive data in URLs
- Encrypt data if necessary to ensure that no intermediary can read it
- Sign data if necessary to ensure that no intermediary can tamper with it

The following sections discuss these concerns further.

12.6.2 Dealing with sensitive search parameters

Search filters can contain sensitive data, but we're used to putting them in query parameters. The `POST /resources/search` pattern, which moves sensitive query parameters from the URL to the request body, is a typical solution to this problem.

We manage account holders with our Banking API. Suppose we must enable partners to search them using sensitive data, such as names. We can't have a request such as `GET /holders?name=Bob`; sensitive information will be logged everywhere. In such a case, a usual design pattern is to use `POST /holders/search` and put the search filters in the body (if someone complains about this being "not REST," refer to section 4.7.1). That way, no sensitive data is logged. We can't use `POST /holders` because it's used to create an account holder. We could use a named resource such as `/holders-searches` to stick to our usual "Resource list plus optional resource identifier" path pattern (section 9.3.2), but doing so would be less intuitive. The `/search` option modifies our pattern into "Resource list plus optional resource identifier or action resource."

12.6.3 *Dealing with sensitive resource IDs*

Interoperable IDs can be sensitive data. Unfortunately, we can't use them as resource IDs because doing so would mean adding a sensitive path parameter to a URL. But we can use the `POST /search` pattern from section 12.6.2 to work around this constraint and keep our API interoperable.

If any possible account holder can be identified with a unique Social Security number known by many different systems outside ours, we might be tempted to read an account holder with `GET /holders/{socialSecurityNumber}` in our Banking API. But we can't put such sensitive data in URLs, so we'll use our internal holder ID, known by all of our systems, instead (`GET /holders/{holderId}`). That makes our API at least internally interoperable. But we can also propose a certain level of interoperability with the outside world by proposing to search account holders by Social Security numbers with `POST /holders/search`, as in the previous section. If a matching holder is found, the consumer gets our holder ID and can use it with any Banking API operation that needs it.

12.6.4 *Integrating data encryption or signing in the design*

Security experts may consider HTTPS insufficient and decide that the consumer or provider must encrypt or sign the request or response data. Encrypting data ensures that only authorized parties can read it. Signing data ensures that it comes from a trusted source and hasn't been tampered with by an intermediary; it leaves the data readable. Both mechanisms can be used together. As API designers, we won't discuss the details of these. But we can ask security experts a few questions to ensure that we correctly adapt our design to security needs and complete our documentation for consumer and implementation developers:

- Do we encrypt or sign all data or only the sensitive element(s)?
- Should data be encrypted or not, depending on consumers?
- How do consumers get or provide the elements to encrypt/decrypt or sign data?

Encryption or signatures may cover the whole request or response or only part of it. We must indicate what is encrypted and how at the operation, request, response, or property-level description. If a signature is used, we need to indicate which elements are covered so the consumer and provider know how to generate or validate the signature.

The need for encryption may vary depending on consumers or the network used. We may use scopes to tweak returning raw or encrypted data (see section 12.8.5).

Encrypting/decrypting and signing/validating data may require elements of configuration or information to be exchanged out of band or within the API calls between the consumer and provider. All this information must be included in the descriptions in our OpenAPI document or in the design itself when relevant.

Learn more about data encryption and signing in APIs

If you want to learn more about data encryption and signing in APIs, I recommend looking at standards such as HTTP Signature header (RFC 9421), Encrypted Content-Encoding for HTTP (RFC 8188), JWE encryption (RFC 7516), and JWT tokens (RFC 7519). The documentation is available at www.rfc-editor.org.

12.7 Limiting consumer access with scopes

An API that exposes only necessary data and operations still allows access to all of them for any consuming application authorized to use the API. The security principle of least privilege must be applied, and access must be granted only to what the consuming applications need. We can use scopes to define which parts of an API an application can access, thereby limiting access to what is essential. Before designing scopes (section 12.8), it is important to understand what scopes do and why their design is significant.



NOTE Scopes are primarily associated with the OAuth and OpenID Connect security frameworks (section 12.1.1). They can be compared to roles or attributes of other security frameworks, such as Role-Based Access Control (RBAC) and Security Assertion Markup Language (SAML).

12.7.1 Limiting access to an operation with a scope

To design scopes, it's essential to understand how they are used during an API call; having the appropriate scopes grants access to applications but does not ensure execution, as business rules and end-user permissions can still influence execution.

Figure 12.7 illustrates what happens when different applications used by various end users call the “Transfer” operation, which requires the `transfer` scope. It shows a typical architecture with the Banking API implementation exposed via an API gateway, an API-security-focused HTTP proxy that controls access to the API and operations. The gateway and implementation use the token in the `Authorization` header to identify the consumer and end user (as seen in section 12.1.1).

The Admin Website and Mobile Banking applications have the `transfer` scope that grants access to the “Transfer” operation (`POST /transfers`), and the third-party SpendAdvize application doesn't. Therefore, the API gateway that controls access to the API blocks the `POST /transfers` call of SpendAdvize. Although the API gateway lets the Mobile Banking application call pass, the implementation refuses to execute it as the user, Bob, doesn't have the “unlimited transfer” user permission required to perform a money transfer exceeding \$100,000. Alice, who has this permission (and uses the Admin Website, which has the `transfer` scope), can transfer money because the account has a sufficient balance. If this were not the case, the implementation would refuse to execute it.

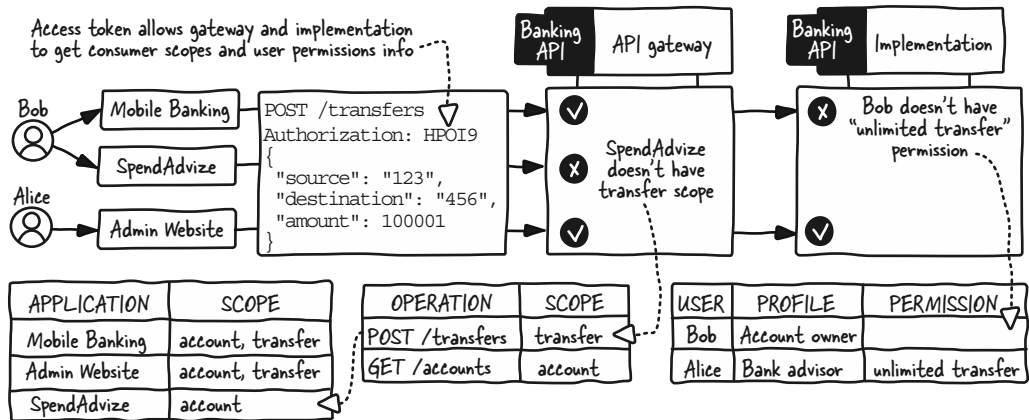


Figure 12.7 Scopes are usually handled at the API gateway level, and the implementation handles user permissions.

12.7.2 Measuring the importance of scopes and their design

Scopes effectively reduce the accessible API surface to what consumers strictly need only if they are correctly designed and understood by the people who choose them. With no scope defined, any authorized application can call all API operations. For example, if an end user authorizes the SpendAdvice application to use the Banking API, it can call operations it doesn't need, such as "Transfer". This exposes a wider API surface than necessary.

We must define scopes to limit the operations consumers can call. For instance, we can require the account scope to list accounts and transactions and the transfer scope to perform transfers. As discussed in section 12.7.1, the SpendAdvice application would only be granted the account scope, preventing it from transferring money.

If we don't carefully design scopes, we may grant access to unneeded and possibly sensitive operations. For example, suppose we define a write scope granting access to all write operations, such as "Update transaction" and "Transfer". If the SpendAdvice application needs to mark transactions as "checked," the end user grants it the write scope so it can call the "Update transaction" operation, but doing that also grants it access to the "Transfer" operation it doesn't need.

Deciding what a scope grants access to is essential, but how we name and describe it is also crucial. API providers, consumers' developers, and end users must be able to understand it clearly. As shown in figure 12.8, the Banking API owner decides which scopes the MyBk App application may use, such as account, transfer, and transaction. MyBk App's developer can request all or a subset of scopes (only transfer, for example) when requesting an access token. End users who want to use our Banking API with MyBk App verify what it can do by reading the scope name (transfer) or description ("Transfer money") shown during the

authentication process. End users don't need to perform such verification when using our Mobile Banking application.

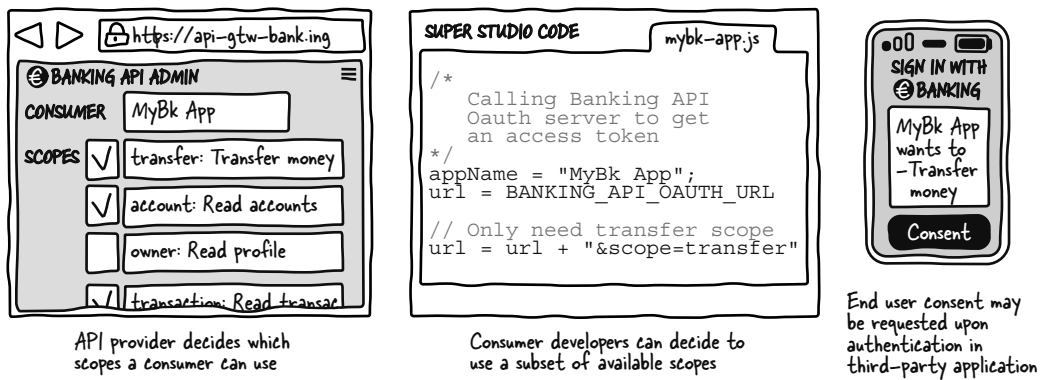


Figure 12.8 Scopes are visible to providers and consumers but also end users.

12.8 Designing scopes

We can design scopes once we have designed the API operations and their data. We must design scopes to offer the proper access to consumers while ensuring that API providers, consumers, and their end users can easily make the right decisions when choosing them (section 12.7.2). This section shows different scope design options illustrated in figure 12.9 and discusses choosing them. These options are as follows:

- Operation-based scopes granting access to one operation
- Resource- or concept-based scopes granting access to all operations strictly or loosely related to a resource
- Use-case-based scopes granting access to operations needed for a use case
- Read-write-based scopes that enable separating access to read and write operations
- End-user- or consumer-based scopes granting access to all operations needed by a user, profile, or application
- Behavior-tweak scopes that modify the behavior of one or more operations

12.8.1 Creating operation-based scopes

We can define one scope per operation and name them `operation_name` or `singular_resource_name:action`, for example. For instance, the `list_transfers` and `transfer:list` scopes grant access to “List transfers” (GET /transfers). Fine-grained operation-based scopes offer the most precise access configuration. However, selecting the right ones can take time and effort for providers and consumers if there are many. They may also overwhelm end users who need to validate them.

SCOPE LIMITING ACCESS TO OPERATIONS					Multiple options are possible in the same API				
OPERATION	SINGLE OPERATION	RESOURCE/CONCEPT	READ/WRITE	USE CASE	END USER/PROFILE/CONSUMER				
List destinations	destination:list	destination:all	destination:read	money transfer	Transfer as a concept could include source and destination				
List sources	source:list	source:all	source:read		owner:all	owner:transfer:all	owner:transfer:read		
List transfers	transfer:list	transfer:all	transfer:read						
Read transfer	transfer:read								
Transfer	transfer:create		Global read or write scope rarely used transfer:write					Combine options to refine coarse-grained scopes	owner:transfer:write
Modify transfer	transfer:modify								
Cancel transfer	transfer:cancel								
Validate uncommon transfer	transfer:validate	Sensitive advisor-only operations		Not for account owners					
List accounts	account:list	account:all	account:read	Not in money transfer use case	owner:all	owner:account:all	owner:account:read		
Read account	account:read								
Create account	account:create		account:write		Not for account owners				
Close account	account:close								
List transactions	transaction:list	transaction:all	transaction:read	SCOPE TWEAKING OPERATION BEHAVIOR					
				transaction:sensitive_data	Could also be sensitive data and apply to all operations				

Most precise option (but can be overwhelming) ---

Less overwhelming coarse-grained scope (but may grant access to operations with different sensitivities)

Figure 12.9 Choose scope design options according to your context. You can use multiple options in the same API and also combine them to refine coarse-grained scopes. Behavioral scopes may apply to one or more operations.



NOTE The API world can be inconsistent: resource names in the path are usually plural, whereas scopes often use singular. We use snake case to name scopes, but other options are also used. Feel free to use your preferred naming conventions, but be consistent and user-friendly.

12.8.2 Creating resource-, concept-, or use-case-based scopes

To simplify understanding and configuration, we can create coarse-grained scopes covering multiple related operations. A resource-based scope grants access to all operations related to a resource (`/resources` and `/resources/{resourceId}`). For example, the `transfer` or `transfer:all` resource-based scope grants access to all operations related to transfer collection and unitary resources (GET and POST `/transfers` and GET, PUT and DELETE `/transfers/{transferId}`).

The resource approach may lead to an incomplete and unhelpful group of operations. In our case, a consumer with the `transfer:all` scope can't perform the money

transfer use case, which requires selecting a source and destination. To address this, we can create use-case-based scopes (Use Case column of the API Capabilities Canvas). The `money_transfer` scope grants access to all operations related to `/transfers` and `/transfers/{transferId}` resources but also the “List sources” (GET `/sources`) and “List destinations” (GET `/sources/{sourceId}/destinations`) operations, which are necessary to perform the money transfer use case.

We can also consider resources less strictly and see them as broader concepts. For example, we can decide that “transfer” is a wider concept, which includes all related operations, such as “List source,” “List destinations,” and “Validate uncommon transfer.” However, in that specific case, we mix operations with different sensitivities; uncommon transfer validation is only for bank advisors. Check section 12.8.4 to discover how to fix this.

12.8.3 Creating scopes for read or write operations

We can create scopes to separate read and write operations. Global `read` and `write` scopes could grant access to all read and write operations, but as our API covers many different concepts, we prefer more focused scopes. We can work at the resource or concept level. The `account` and `account:all` scopes grant access to read and write operations such as “List accounts,” “Create account,” “Read account,” and “Close (delete) account.” However, create and close operations are sensitive because they modify data. To address this, we can create an `account:all:read` or `account:read_only` scope that grants access to list and read, along with `account:all:write` and `account:write_only` to covers create and close.

12.8.4 Creating end-user- or consumer-based scopes

We can create end-user- or consumer-based scopes because different end users or consumers (identified in the API Capabilities Canvas User column) may work with the same resource or have similar use cases but require slightly different operations. We can have broad scopes that grant access to all operations a specific user needs. For example, the `owner:all` scope grants access to all operations needed by an account owner. We typically grant this scope to our Mobile Banking application and Banking Website. However, this approach grants access to many operations and hides the details of what is granted. It can be a problem with third-party applications, especially if end users need to validate scopes. We can narrow such scopes to address this with concept/resource and read/write considerations. The `owner:transfer:all` grants access to all transfer-related operations, excluding “Validate uncommon transfer,” which is reserved for bank advisors. The `owner:transfer:write` scope grants access to the “Transfer,” “Modify transfer,” and “Cancel transfer” operations.

Similarly, we can have scopes defined for specific consumers. The `third_party_server_application:account_application:all` scope grants access to all operations a partner needs for account applications in the context of a server application.



NOTE We end with the `{user or consumer or wildcard}:{resource or use case or wildcard}:{action or wildcard}` scope naming pattern, which allows us to use different approaches altogether. Feel free to use another one, but remember what we’ve learned about user-friendly names and resource paths: it must be simple, organized, and intuitive.

12.8.5 *Tweaking operation behavior with scopes*

Scopes can also modify an operation’s behavior; the implementation usually handles such scopes, as an API gateway doesn’t handle business logic. Section 12.3.3 discusses how to make account transaction data non-sensitive so that it can be shared with a third party. This makes the “List transactions” operation specific to a type of consumer. To avoid this, we can condition the return of sensitive data to the `transaction:sensitive_data` scope. A third-party application with the `transaction:all` scope can call “List transactions” but get the non-sensitive data only. Applications like our trusted web and mobile applications with `transaction:all` and `transaction:sensitive_data` scopes will get the merchant’s name and address.

12.8.6 *Deciding which scope types to use*

This section summarizes the previous ones to help you decide among the various options. Fine-grained operation-based scopes are the most precise but may make the configuration complex and overwhelming, especially if third parties or end users are involved. To address this, use coarser-grained resource-, concept-, use-case-, user-, or consumer-based scopes. However, be careful not to grant access to operations with different sensitivities or create big, unclear scopes that third parties or end users may have difficulty grasping. Combine different options, including separating read and write operations, to prevent this. Feel free to use multiple options within the same API to match all user needs; multiple scopes can cover the same operation.



NOTE With API gateways, modifying scopes that limit access to operations is relatively simple; it’s only a configuration often based on an OpenAPI document. Modifying scopes that tweak behavior is more complex, as it most likely requires modifying the implementation. Modifying scopes after deployment may have consequences for consumers already using the API; refer to section 15.3.

12.9 *Describing scopes with OpenAPI*

We can use the OpenAPI format to describe scopes and the operations to which they apply. This section discusses defining scopes, organizing them in groups, and applying them to the relevant operations.

12.9.1 *Defining scopes*

As illustrated in figure 12.10, API security is defined through *security schemes* under `components.securitySchemes`. The security scheme name is a key in the object

(OperationBased). If you're unsure about which type and flow to use, ask a security expert; you can temporarily use the `oauth2` type and `implicit` flow. You need to define URLs for a flow (`authorizationUrl`, for example); use dummy values if unknown. Scopes go under the `scopes` object; names are keys ("`beneficiary:create`") and description values ("Add a beneficiary"). Use quotes around names with `:` to avoid YAML errors. Multiple security schemes can be defined to separate scopes.

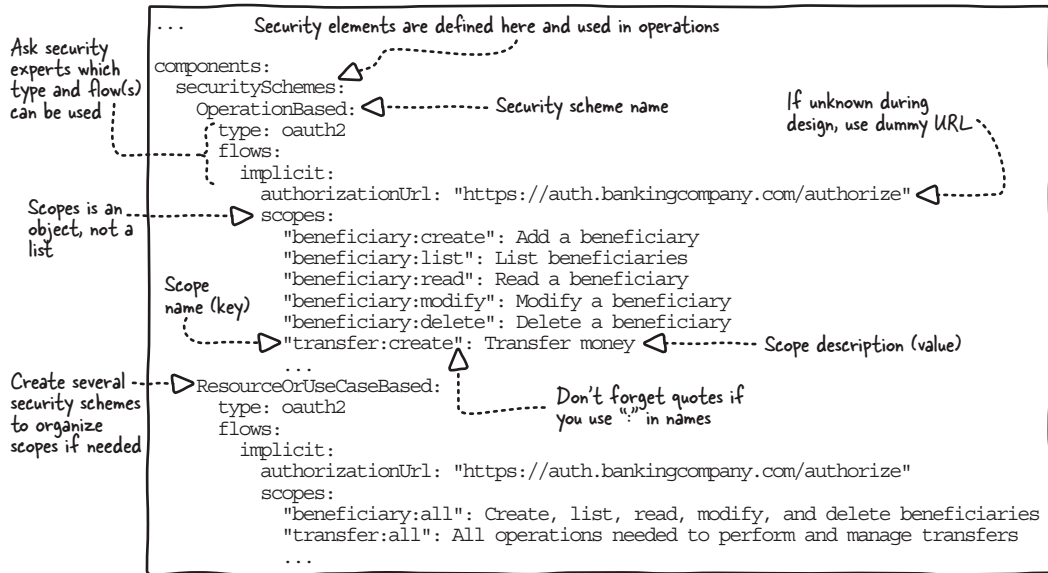


Figure 12.10 Define scopes in security schemes under `components.securitySchemes`. You can define multiple security schemes to organize scopes.

OpenAPI allows us to organize scopes into different groups. This can help differentiate fine-grained (OperationBased) and coarser-grained (ResourceOrUseCaseBased) scopes or scopes that vary depending on consumers, for example.

12.9.2 Using scopes

After defining security schemes and scopes, we can use them on relevant operations, as illustrated in figure 12.11. Add a `security` list where each element is an object with a property matching a security scheme defined under `components.securitySchemes`. The value should be a list with a scope corresponding to one specified under the selected security scheme's `scopes`.

The consumer must match one of the security list elements to use the operation. For instance, to call `POST /beneficiaries`, a consumer needs the `beneficiary:create` or `beneficiary:all` scope. OpenAPI supports more intricate combinations; refer to <https://spec.openapis.org/oas/v3.1.0#operationSecurity> for details.

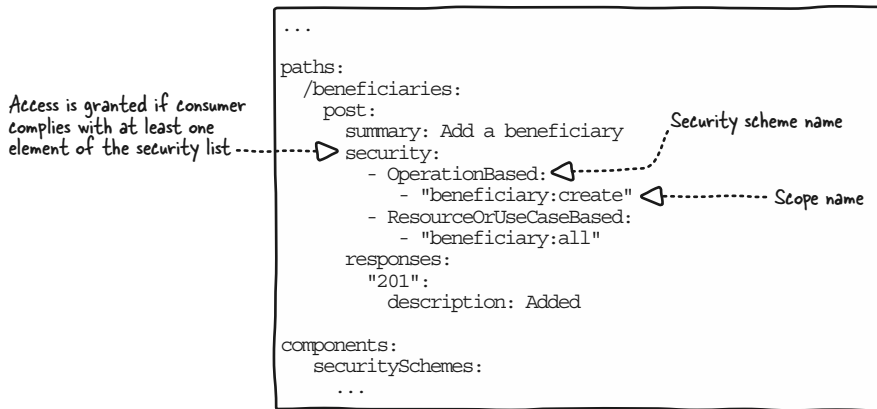


Figure 12.11 The security list contains the security schemes and scopes a consumer needs to use to access an operation.

12.10 Erroring securely

It's essential that API operations properly handle security-related errors and avoid leaking sensitive security or implementation information on errors. Using our learnings about HTTP (section 4.5.8) and user-friendly errors (section 9.8), we can contribute to that effort by doing the following in our OpenAPI document:

- Add and document token-related errors (401 Unauthorized)
- Add and document scope- or permission-related errors (403 Forbidden or 404 Not Found)
- Complete the documentation for unexpected errors (500 Internal Server Error)

12.10.1 Handling token-related errors

Any API call with a missing, expired, or invalid access token must result in a 401 Unauthorized response. Developers calling any operation of the Banking API who forget to provide the Authorization header, send it with an expired token, or use a token with a typo because of an incomplete copy/paste get a 401 Unauthorized HTTP status with a generic error message, such as “Missing, invalid, or expired token.”

12.10.2 Handling missing scopes or permissions

A call may be rejected despite a valid token due to a missing consumer scope or insufficient end-user permission. If the requested resource's existence can be disclosed and the problem can be resolved by requesting the appropriate scope or permission, the operation returns a 403 Forbidden HTTP status or 404 Not Found otherwise. Figure 12.12 shows an example.

If a consumer calls `GET /accounts/12345` but doesn't have a scope granting access to the “Read account” operation, the operation returns 403 Forbidden. The error

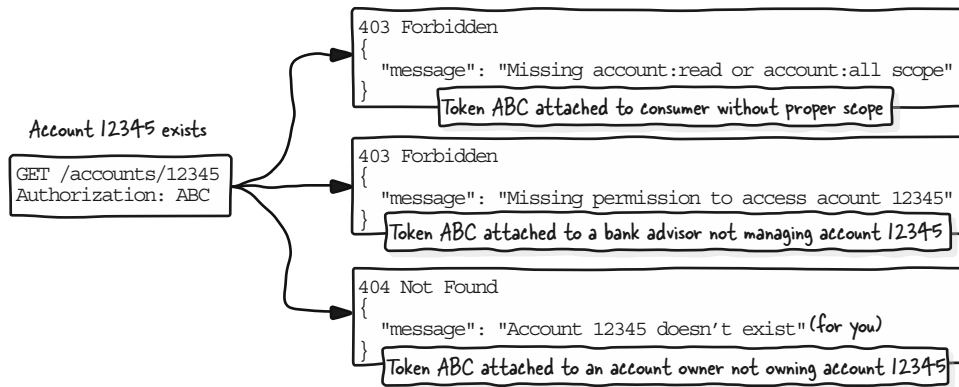


Figure 12.12 The “Read an account” operation returns “403” if the consumer doesn’t have the proper scope or if the end user is a bank advisor who doesn’t manage the account. It also returns “404” if the end user is an account owner who does not own the account.

message could be “Missing scope,” which doesn’t disclose which scope is missing, or “Missing account:read or account:all scope,” which discloses the options.

Suppose Alice is a bank advisor using the Banking Admin Website and tries to access the 12345 account, which isn’t her responsibility. The response to `GET /accounts/12345` is `403 Forbidden` with a “Missing permissions to access account 12345” error message. It indicates that she doesn’t have access to the 12345 account but may request additional permissions to get it. Similarly, Alice may not be allowed to trigger transfers exceeding \$100,000; calling the “Transfer” operation with an amount set to 100001 would result in a `403`.

If we don’t want to disclose that an accessed resource exists, or the consumer or end user can’t request an additional scope or permission to fix the problem, the operation must return `404 Not Found` instead of `403 Forbidden`. For example, if an account holder, Bob, triggers a `GET /accounts/12345` call for an account he doesn’t own, the API can return `404 Not Found` with an “Account 12345 not found” message. This way, the API communicates to Bob that the account doesn’t exist for him, even though it exists in reality.

12.10.3 Avoiding disclosing implementation details on server errors

In section 4.5.9, we added a `500 Internal Server Error` HTTP status to all operations to handle unexpected errors. It’s crucial to avoid leaking sensitive implementation details like server names or IP addresses, programming languages, or database engines in the error messages, as hackers can exploit such information. Instead, return a generic error message that conceals implementation details. You can also provide an opaque reference for further investigation if needed. Figure 12.13 contrasts `500` responses that do or do not disclose implementation details.

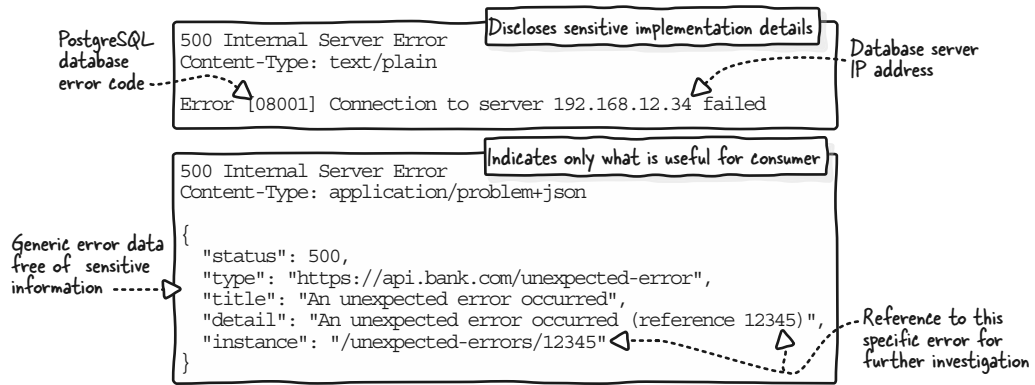


Figure 12.13 An unexpected server error must not disclose implementation details. Return generic information with an optional reference to this error’s occurrence for further investigation.

The Error [08001] Connection to server 192.168.12.34 failed message indicates that the PostgreSQL DB is used (error code 08001) and its IP address. The safer alternative uses the Problem Details for HTTP APIs standard we discovered in section 9.8.6, but you can apply the same principle to a custom error format. Compared to what we’ve seen previously, we’re using a new property, *instance*, which allows us to provide a reference to the specific error that just happened. It’s a URI (`/unexpected-errors/12345`), but it doesn’t have to be resolvable; it can just appear in the server logs, for example. We also put the reference number (12345) in the human-readable detail, which may be shown to end users.

12.10.4 *Providing implementation details in response descriptions in OpenAPI*

Using the HTTP status reason as a response description, such as “Forbidden” for a 403 response in the OpenAPI document, is the surest path to an improper implementation. We must provide details about security-related client errors (4XX class) for the implementation developers, as illustrated in figure 12.14.

There are often uncertainties about 401 versus 403 and what an invalid token is (“Does it include tokens with wrong permissions?”). To avoid this, we indicate that “invalid” means “nonexistent, typo” and that “scope-related errors are handled with 403” in the 401 response description. Check out section 17.5.2 to see how to do this once for all 401 responses.

An implementation-friendly 403 or 404 description clearly states the condition in which it happens. On the `GET /accounts/{accountId}` operation, the 403 description can state, “The access token is missing a scope, or the identified user is a bank advisor who doesn’t have access to the account.” Its 404 indicates, “The account doesn’t exist, or the identified user is an account owner who doesn’t have access to it.”

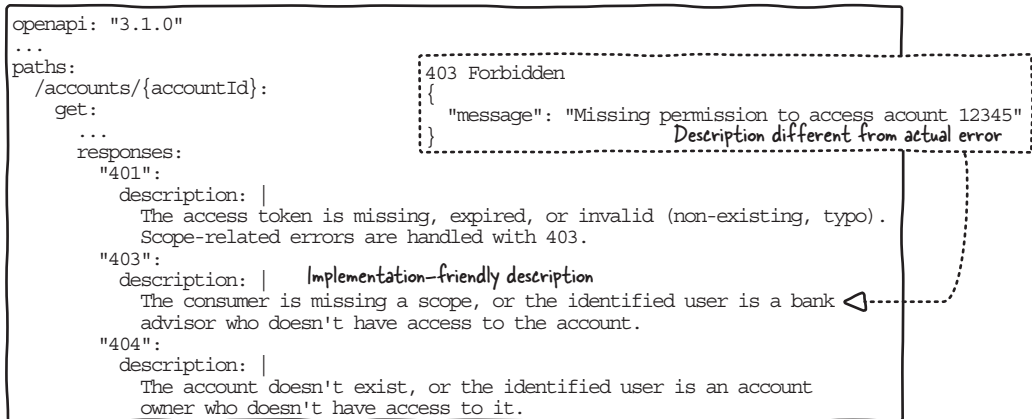


Figure 12.14 Each response description provides information that is essential for the implementation. The response description is different from the actual error message.



CAUTION Do not confuse the response description in the OpenAPI document (“The consumer is missing a scope, or the identified user is a bank advisor who doesn’t have access to the account”) with the API’s error response (“Missing permission to access account 12345”). The OpenAPI document may provide more information than the error message.

12.10.5 Enforcing expected error data with JSON Schema

It’s common not to define a schema or to use the same generic schema for all error responses in an OpenAPI document. That can lead to the implementation returning an improper data format and, worse, a sensitive information leak. To ensure that the error data the implementation returns contains precisely what is expected, we can use JSON Schema, as shown for a 500 error using the `application/problem+json` format in figure 12.15. You can apply this to 4XX errors, too.

In the `UnexpectedError` JSON schema defined under `components.schemas`, we use the `const` keyword to indicate the only possible value for an element. For example, we constrain the possible value for `title` to “An unexpected error occurred.” The `detail` property contains the title plus the error reference; we constrain the value using the `pattern` keyword, which is a regular expression describing the expected string. Regular expressions can be complex to read; we provide an `example` value to clarify expectations. Afterward, we use the schema with `$ref` set to `#/components/schemas/UnexpectedError` in the 500 response of all operations. If developers of the implementation respect this contract, no sensitive information should leak.

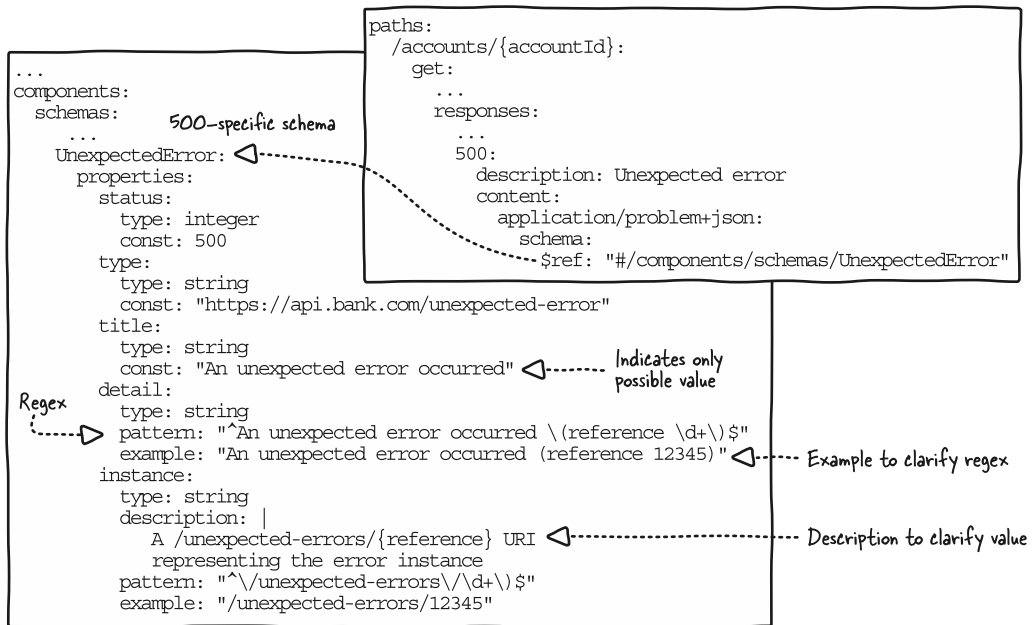


Figure 12.15 The UnexpectedError schema describes precisely what is expected on 500 errors, which avoids disclosing sensitive information.

Summary

- Secure any API, even if it is private and not exposed on the internet, to ensure that only relevant operations and data are accessible to authorized consumers and end users.
- To ensure security, minimize the API surface, avoid revealing the provider perspective, and provide all necessary information to the implementation and developers.
- Consult with competent experts to identify sensitive data and operations and define how to protect them; share an exhaustive OpenAPI document with the experts.
- Challenge any sensitive or non-sensitive data and operations, remove or replace sensitive elements, split operations, or separate sensitive operations into dedicated APIs based on subject-matter relevance, to expose only the necessary elements.
- Describe what consumers and end users can see or do in the operation description in the OpenAPI document, and adapt the design to allow returning only the needed data. This will limit runtime data leaks and security-related problems.

- Use the `POST /resources/search` pattern to move sensitive search parameters from the URL to the request body and enable retrieving non-sensitive resource IDs from interoperable but sensitive data.
- Use conditional requests (`If-Match: <ETag>`) to ensure that updates won't revert data.
- Discuss encrypting or signing data with security experts to adjust the design as needed and include relevant details in the documentation.
- Design user-friendly scopes so that API providers, consumers, and their end users can easily make the right decisions when choosing them.
- To partition access to operations with scopes, choose one or more approaches (operation, resource, use case, read/write, end user, or consumer).
- Use scopes to tweak operation behaviors such as returning sensitive data.
- Define scopes under `components.securitySchemes`, and use them on each operation under `security` in the OpenAPI document.
- Return 401 Unauthorized for token problems, 403 Forbidden for fixable scope or permission problems, and 404 Not Found for non-fixable ones.
- Add information to OpenAPI to ensure that 500 Internal Server Error data doesn't disclose implementation details.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.ba/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 12.1

You're designing an API for a health and fitness tracking application. Listing 12.2 shows the data returned to a specific user when reading their profile, and listing 12.3 shows the data returned when reading one of their workouts. Stakeholders are considering allowing consumers to use these operations to share information across all platform users; any user can read any profile or workout. Are there any security problems? How can you avoid them?

Listing 12.2 Reading user data

```
{
  "id": "spikey34"
  "phone": "+33123456789"
  "name": "Spike Spiegel",
  "email": "spike@bebop.com",
  "birthDate": "2044-06-26",
  "fitnessLevel": 123
}
```

Listing 12.3 Reading workout data

```
{
  "id": "1234",
  "date": "2066-09-01",
  "type": "running",
  "duration": 45,
  "calories": 350,
  "gpsCoordinates": [
    {
      "timestamp": "2066-09-01T10:00:00Z",
      "latitude": 40.748817,
      "longitude": -73.985428
    },
    ...
  ]
}
```

Exercise 12.2

A company specializing in data extraction creates an API to determine whether a string format matches any ID format of any country. For example, `GET /identifiers?id=123-456-789&country=BRA` could determine whether 123-456-789 matches a Brazilian driver's license ID or any other Brazilian ID. Does the design of this operation have security problems, and if so, how can they be fixed?

Exercise 12.3

In a hospital, different applications, such as Patient Folder, Treatment, and Schedule, use the same medical API. This API has a `GET /patients/{patientId}/conditions` operation whose data must be accessible only to the doctors treating the patient on the Patient Folder and Treatment applications. Can this be handled with scopes or implementation checks?

Exercise 12.4

An API for managing library systems has the following operations and scopes:

- `GET /books`, scope: books
- `POST /books`, scope: books
- `GET /books/{bookId}`, scope: book
- `PUT /books/{bookId}`, scope: book
- `DELETE /books/{bookId}`, scope: book
- `GET /members`, scope: members
- `POST /members`, scope: members
- `GET /members/{memberId}`, scope: member
- `PUT /members/{memberId}`, scope: member

- `DELETE /members/{memberId}, scope: member`
- `POST /members/{memberId}/borrowed-books, scope: borrowed books`

Could the scope design cause security problems? If so, how could you fix them?

Exercise 12.5

An event management API has a `GET /events/{eventId}/attendees` operation that an event organizer uses to check the people coming to an event. Suppose users A and B are event organizers, and event 1234 is organized by A. Which HTTP status should be returned when B sends a `GET /events/1234/attendees` request?

13

Designing an efficient API

This chapter covers

- Optimizing design when necessary
- Enabling caching and conditional readings
- Optimizing data volume
- Processing multiple elements with bulk operations
- Considering an optimization-specific API layer

Since the Shopping company implemented its newly redesigned API with additional features, the number of product purchases has dropped. Website and mobile analytics indicate that users are experiencing wait times of over 500 ms when searching for products. This problem is due not to a missing index in the database or inadequate infrastructure but rather to the inefficient design of the API. Searching for products necessitates firing multiple API requests for each product found to gather all the necessary data and display it to users.

Concern about efficiency is not limited to the implementation level. An efficient API design can lead to benefits including optimized smartphone battery usage, reduced wait times for end users, and minimal effects on 5G data plans. Furthermore, it can enhance provider infrastructure efficiency, reducing cloud infrastructure costs through reduced CPU usage and data downloads.

This chapter provides an overview of common mistakes and oversights that lead to inefficient APIs and explains how to design an efficient API. We discuss aspects to consider beyond design before considering drastic optimizations. Then we revisit past learnings from the efficiency perspective. We discuss new considerations to make our API efficient: cache and conditional requests, data volume optimization, and performing multiple operations with a single call. Finally, we consider creating a specifically optimized API layer so we can keep versatile and reusable APIs underneath.

13.1 An overview of API efficiency

After security, efficiency is the next most vital constraint that can affect API design. An efficient API enables consumers to accomplish their goals quickly with minimal energy or processing, even when communicating over a low-quality network, while using the provider's infrastructure effectively. As API designers, we must understand how inefficient APIs can negatively affect consumers and providers and how to help prevent this at the design level. This section presents an example of an inefficient API, discusses when to be concerned about efficiency during API design, and introduces the principles we can use to create an API that is efficient by design.

13.1.1 How an API can be inefficient

An inefficient API has direct and immediate consequences for both consumers and providers. Consumers experience global slowness and unnecessary data load, which affects their end users' experience and satisfaction. In addition, providers face an overloaded and costly infrastructure. Figure 13.1 illustrates these concerns.

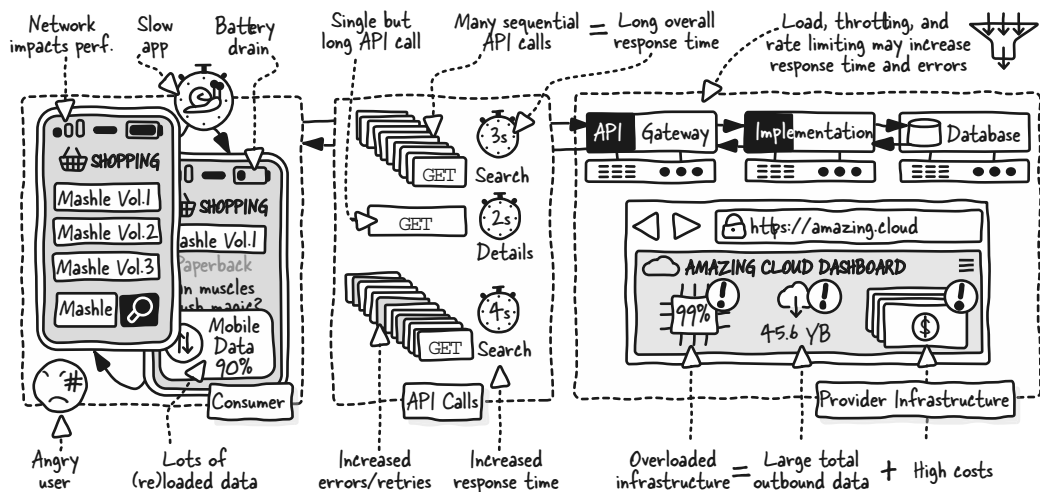


Figure 13.1 An API can negatively affect consumers with slow response times and high data volumes, and providers with costly, overloaded infrastructure and massive total outbound data.

Users are frustrated by the Shopping mobile application's slow performance, battery drain, and high data usage. API interactions are sluggish due to large amounts of returned data and infrastructure overload. Long sequential API call flows slow the process. A sequence of ten 300-millisecond calls lasts only 3 seconds, but the human brain perceives a delay for any time greater than 500 milliseconds. These call flows also increase the risk of errors and battery drain. Users with limited data plans quickly reach their limit due to repeated data loading. The experience worsens in areas with poor mobile connections or congested networks, where limited bandwidth makes everything slower.

The Shopping API provider is frustrated by losing customers and skyrocketing infrastructure costs caused by excessive requests overloading the system and high outbound data volume. Throttling, rate limiting, and optimizing the implementation and database have helped, but the system remains slow for consumers, and the outbound data problem persists with unnecessary repeated requests.

Applications using an API over a wired network are also concerned about efficiency. A batch server application that performs stats on the orders and runs on an on-demand infrastructure may increase costs by running longer than necessary. And when the API slows the internal administration web application, employees lose time.

13.1.2 When to be concerned about efficiency

Efficiency is a concern for all actors across the API lifecycle. Although consuming applications and API implementation and their dependencies (database or other APIs) can be optimized by architects, tech leads, or developers, some design elements will hamper these efforts. Efficiency constrains our design by affecting data modeling, operation design, and operation flows, as illustrated in figure 13.2. It's essential to

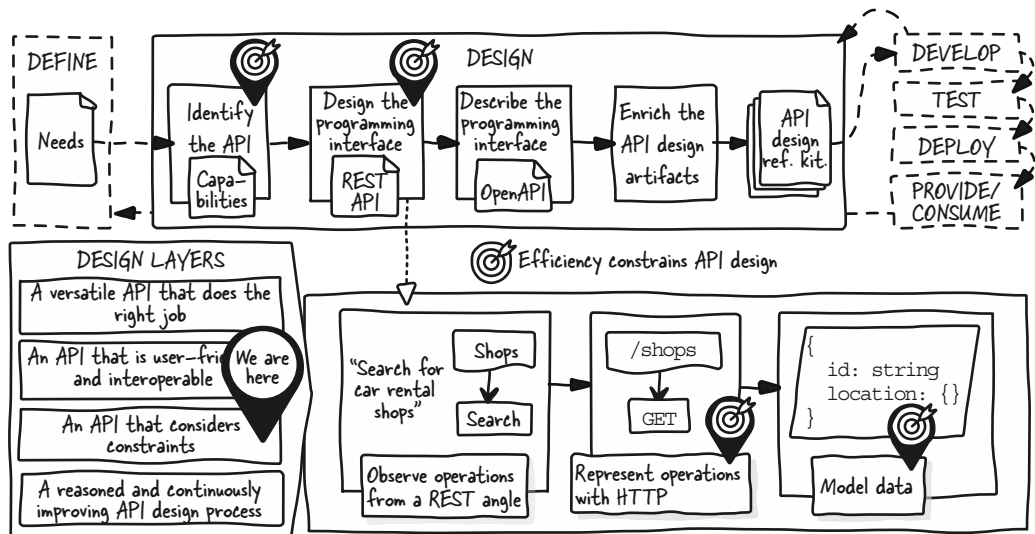


Figure 13.2 Creating a versatile API that does the job and is user-friendly and interoperable puts us on the right track. But we can improve data, operations, and flow design to be totally efficient.

continue separating concerns and consider efficiency after designing a user-friendly, interoperable API that does the job. Fortunately, such an API is already reasonably optimized. But there's always room for improvement.

13.1.3 How design contributes to API efficiency

To be efficient by design, our APIs should require the fewest possible calls, especially sequential ones, and handle the least possible data to achieve use cases. To do so, we can

- Optimize input and output data
- Optimize API call flows
- Enable cache and conditional requests
- Consider an efficiency-specific API layer

In the Shopping application, searching for products requires 10 calls to present the result to the end users. Our search returns too few products simultaneously, and product summaries miss essential information. Consumers need to make additional API calls sequentially to fill the gaps. But retrieving product details returns massive amounts of data.

We can optimize this to get only the necessary information in one call. The mobile application retrieves the same product or searches data each time users navigate. The application can cache data to reuse it if the API returns the proper information. Additionally, we can help consumers by retrieving data only if it has changed. If that's insufficient and no infrastructure configuration can help, we can consider more drastic design optimization. However, that could make the API specific to the mobile application. So in that case, we can consider creating a dedicated API that uses the regular Shopping API underneath. The following sections discuss these concerns in more depth.

13.2 Optimizing the design only when necessary

In this chapter, we'll discover patterns we can use to make our API efficient by design. But there are a few non-design-related concerns we need to be aware of as API designers to prevent injecting unnecessary complexity in our design and address performance problems: those we imagine at design time and actual ones discovered after deployment at runtime. This section discusses

- Ensuring HTTP configuration efficiency
- Limiting API usage with rate-limiting
- Enhancing response with rate-limiting headers
- Finding the true root cause of an API performance problem

13.2.1 Ensuring HTTP configuration efficiency

An appropriate HTTP server configuration can drastically enhance API efficiency, optimizing data volume and the network connection. HTTP versions can be divided into "1.1" and "2 and above." HTTP 1.1 represents a small portion of internet traffic

compared to HTTP 2 and later. Nonetheless, many APIs still employ HTTP 1.1. The choice of version doesn't alter our use of HTTP (requests, responses, methods, statuses, etc.). However, using HTTP 2 directly or via a proxy is more efficient.

HTTP 1.1 can compress request and response body data (not headers) to reduce data volume, but this may not be enabled by default. An HTTP 1.1 server keeps connections open, enabling consumers to reuse connections for other requests. However, HTTP 1.1 does not allow parallel requests over the same connection, requiring consumers to open multiple connections. Additionally, browsers and smartphone OSs limit simultaneous connections to a server, affecting overall response time.

HTTP 2 and later versions resolved HTTP 1.1's limitations and enhance performance. They offer improved compression for bodies and headers, use a more efficient binary format for data transmission, and support persistent connections and parallel requests. Consumers can send requests in parallel up to the server's capacity without being limited by browsers and smartphone OSs.



NOTE On average, HTTP compresses JSON or any textual data by up to 75%; the compressed data is reduced to 25% of its original size. Although additional compression is unnecessary, we may want to optimize the returned data to minimize volume (section 13.5).

13.2.2 *Limiting API usage with rate-limiting*

Rate-limiting lets us control how consumers perform requests over time by, for example, limiting the number of requests over a short period. This is crucial for protecting underlying systems from malicious (denial of service [DoS]) or unintentional (loop bug) floods of requests and ensuring fair distribution of resources among all consumers. On-premises infrastructure has limited capabilities and is often difficult to extend, and although cloud infrastructure can expand to meet demand, budget constraints may arise. Defining rate-limiting policies isn't our role, but we can at least ensure that they are in place and applied in a user-friendly, interoperable way.

The money transfer system handles 1,000 requests per minute. Increasing the server's CPU and RAM may enhance processing capacity, but limits remain. Each consumer must be granted a limited number of calls per minute to protect the system. On reaching their limit, the API gateway returns 429 Too Many Requests with a standard `Retry-After: Tue, 02 Jan 2024 14:12:00 GMT` header for the reset time. Additional error details may be provided (section 9.8), but they must adhere to the error structure shared by all APIs and outlined in the API design guidelines (section 16.3).



NOTE Rate-limiting can also restrict API usage based on payment tiers. Rate-limiting policies may need to evolve as the business, API usage, or infrastructure changes. Rate-limiting is usually implemented by an API gateway managed by a separate team; coordinate with them to maintain a consistent design across all exposed APIs.

13.2.3 Enhancing response with rate-limiting headers

Rate-limiting headers can be added to all responses to help consumers be more efficient by being proactive instead of waiting for a 429 error. For example, `X-RateLimit-Limit: 100` shows the total allowed requests, `X-RateLimit-Remaining: 90` indicates requests remaining, and `X-RateLimit-Reset: Tue, 02 Jan 2024 14:12:00 GMT` displays the reset time. Some APIs, such as GitHub's, use epoch timestamps (1704204720) instead of an HTTP date. However, I use HTTP dates in the examples because it's the standard date format in HTTP headers.

The `X-RateLimit` headers are custom and not defined by any RFC but have become a standard (with some naming variations). However, I recommend considering the `RateLimit` and `RateLimit-Policy` headers defined by the "RateLimit header fields for HTTP" IETF draft (which may become an RFC). The server returns `RateLimit: "default";r=90;t=20` to indicate remaining requests (`r=90`) and a reset in 20 seconds (`t=20`), and `RateLimit-Policy: "default";q=100;w=60` to state that it allows 100 requests (`q=100`) per 60 seconds. Check the documentation at <https://datatracker.ietf.org/doc/draft-ietf-httpapi-ratelimit-headers> for more details.



NOTE The `x-` header name prefix indicates eXperimental or eXtension; it aims to separate custom headers from standard ones. This practice was deprecated in 2012 with RFC 6648 due to the costs of leakage into standards (www.rfc-editor.org/rfc/rfc6648.html). Once widely adopted, names are difficult to change, which applies to any API data; this is discussed further with API modifications in section 15.1.

13.2.4 Finding the true root cause

Runtime efficiency problems sometimes lead to hasty blame on the API design. However, most API efficiency problems I've encountered over the years were caused by the implementation (in a broad sense), not the API design. I've encountered problems caused by undersized servers, misconfigured load balancers driving all traffic to a single instance, undersized networks, and bugged or sub-optimized implementations. However, the most predominant cause was databases; I can't count how often the solution was to add a missing database index.



CAUTION Before attempting to solve an efficiency problem (or any other), make sure its true root cause is identified.

13.3 Focusing on user needs and user-friendliness to be efficient

Our focus on user needs and user-friendliness has helped us build efficient APIs, balancing factors like API data and operation call flows. This section summarizes what we have learned and demonstrates how to optimize an inefficient flow, using the Banking API example from previous chapters.

13.3.1 What we've learned so far

Our past learnings seamlessly guided us in creating an efficient API that deals only with necessary data and operations. For instance, in section 5.5.3, we discussed ensuring that our data modeling fits our users' needs so we only request or return essential data. In section 9.4.4, we cut in half the data needed to create a money transfer by minimizing input data. In section 9.9.2, we also learned to rethink an operation's purpose, allowing us to deal with more context-focused data and possibly less data. These locally minor optimizations can make a massive difference at scale, especially when operations are called thousands of times per hour or second.

We also worked at the use-case level. For example, in section 10.3, we optimized the selection of the source and destination of a money transfer from three sequential generic steps to a one-step flow specific to the use case. A shorter flow with fewer or no sequential steps takes less time.

13.3.2 Analyzing an inefficient flow

Let's analyze an inefficient flow to understand the effects. The banking application's home screen shows account information for the user and related individuals, including total balances, pie charts of categorized transactions over the last three months, and alerts for transactions in a country other than the owner's. Figure 13.3 shows the operation flow the application uses to get this information.

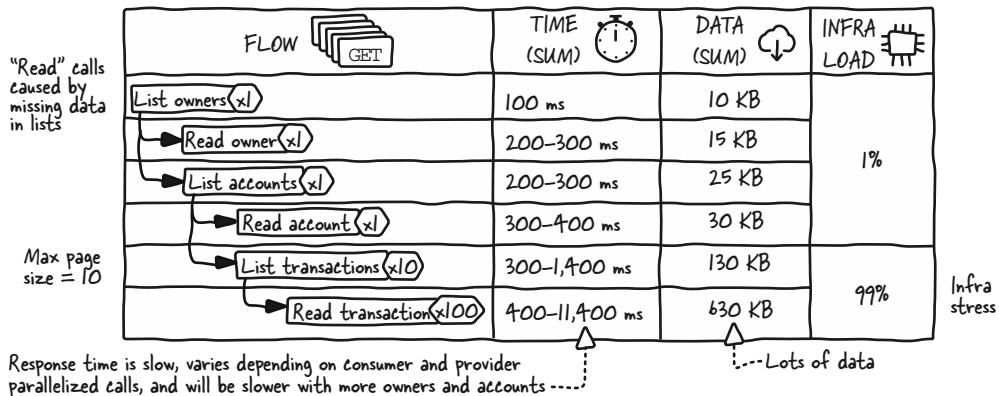


Figure 13.3 The home screen information flow is slow due to numerous sequential calls, which stresses the infrastructure with up to 100 parallel calls. Also, three months of transactions mean a lot of data.

It reads and lists owners (related to users), accounts, and transactions, with each element requiring individual reads because essential data is missing in the summarized models used in lists. Three months of transactions is an average of 100 transactions: 10 calls are required to list all transactions because the maximum page size is 10. That takes 400 ms and represents 630 KB of data on average. These stats apply to a simple

case with one owner and one account; they increase with more owners or accounts, which is common.

These numerous calls are triggered every time a user visits the application's home screen, which stresses the infrastructure, including the database. This leads to decreased responsiveness and fewer parallel requests. The flow could take up to 11.4 seconds on the worst days, not considering network quality concerns. Additionally, although 630 KB may seem small, 100,000 users viewing their dashboard daily adds up to around 1.8 TB of data each month.

This flow could also be more developer-friendly: developers have to implement many concurrent calls and deal with errors. Fortunately, we can improve it by using what we have learned, as shown in the following sections.

13.3.3 Optimizing each operation

As shown in figure 13.4, we can enhance response time and reduce infrastructure load by working on each operation, especially by rethinking data models and input data according to user needs and the subject matter.

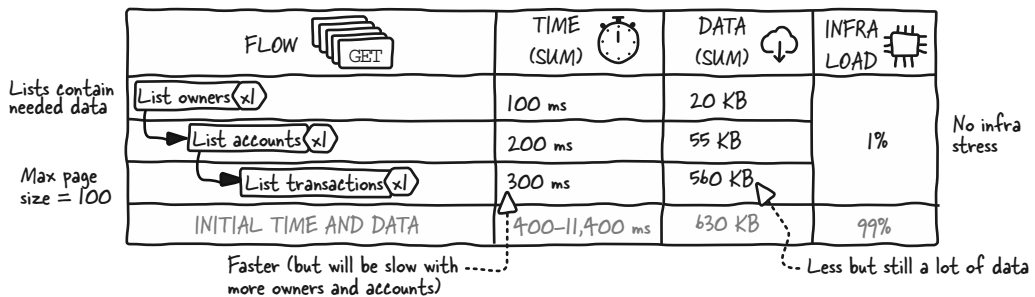


Figure 13.4 Due to relevant data additions and larger transaction list pages, the flow is faster and infrastructure-stress-free. However, the process may slow down with more owners and accounts, and the data volume remains substantial.

We have learned to design helpful list data models by summarizing essential data from complete data models or using these models entirely. Therefore, we can modify the list owners, accounts, and transaction operations so they return the relevant data. This approach eliminates the need to read each element returned by list operations, especially reading each of the 100 transactions returned by list transactions.






Additionally, setting the maximum page size to 10 when listing transactions is irrelevant to the topic of account transactions in general and our context. There are an average of 33 transactions per month, and most consumers in many use cases need more than that. Increasing the value to 100 aligns with actual requirements without affecting performance. It allows consumers to get all the data they need in one call most of the time; if they need less, they can set a smaller page size.


These optimizations simplify the flow to three steps and enhance its response time (300 ms versus 400 to 11,400 ms). However, it will still be slow if there are more owners and accounts (more calls to “List accounts” and “List transactions”), and it still deals with a significant amount of data (560 KB versus 630 KB).

13.3.4 Rethinking the flow

An inefficient operation or flow can indicate that we’re not solving the right problem. We must question why a consumer would need such a complicated flow. The application calls generic operations (“List owners,” “List accounts,” “List transactions”) to perform many calculations to present a dashboard. We’re delegating business logic to the application, which we must avoid at all costs. It is error-prone and not user-friendly, and we now realize it can also be inefficient.

Although it aggregates various kinds of information (owners, accounts, transactions) in a specific way, the dashboard is a valid business concept that can be reused in our customer- or advisor-facing applications. Figure 13.5 shows a dashboard-focused flow that prioritizes user needs and conceals business logic. It is faster (100 ms versus 400 to 11,400 ms), returns only necessary, ready-to-use data (20 KB versus 630 KB), and places minimal strain on the infrastructure because only one call is required. It would also be fine if there were more owners and accounts. The icing on the cake is that it’s super user-friendly. But the result will be efficient only if the implementation handles the workload efficiently. Suppose the implementation reproduces the initial unoptimized flow by reading each transaction individually: in that case, we’ll avoid the many round trips over the internet but still put considerable stress on the database.

	FLOW 	TIME (SUM) 	DATA (SUM) 	INFRA LOAD 	
Use-case-specific operation	Read dashboard <x> 	100 ms	20 KB	1%	No infra stress
	INITIAL TIME AND DATA	400–11,400 ms	630 KB	99%	

Always fast (if optimized implementation) 


Necessary data only and no business logic delegation 

Figure 13.5 The real problem is delegating business logic to the consumer. It’s more efficient to create an operation that fulfills actual user needs. It is fast and returns only necessary, ready-to-use data.



NOTE Such a specialized operation may be far from the actual subject matter and limit flexibility. We may want to put such an operation in a separate API dedicated to efficiency (section 13.8).

13.4 Enabling caching and conditional readings

The most efficient API call is the one that doesn't occur or retrieves data only when necessary. To achieve this, we can indicate to consumers whether and how they can save data for reuse (caching) and allow them to retrieve data only if it has changed (conditional request). This section examines these features and discusses

- Not letting consumers decide how to cache
- Defining caching policies according to data and context
- Returning cache directives
- Retrieving data only when modified

13.4.1 An overview of caching and conditional readings

When consumers cache data, they store it for later use, which reduces API calls and data downloads. However, there's a risk of using outdated data. Conditional requests help mitigate this risk by allowing consumers to retrieve data only if it has changed. This means downloading less data while still calling the API. Figure 13.6 shows the effect of cache and conditional requests when a user navigates the Banking mobile application.

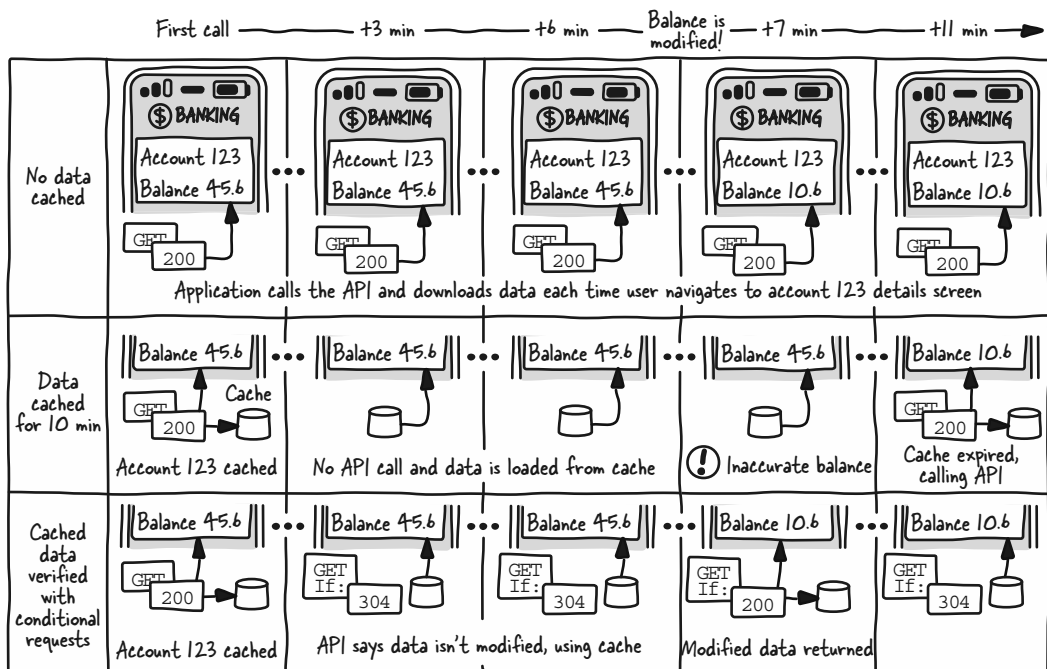


Figure 13.6 The application must call the API to show data when it isn't cached. When data is cached for 10 minutes, the application gets data from the cache, which may be inaccurate. Before using it, the application can call the API to ensure that its cache is up to date.

When no data is cached, the Banking mobile application calls `GET /accounts/123` each time the user visits the account 123 details screen. If the account data is cached for 10 minutes, the application calls the API to initiate the cache. It then loads the data from the cache for subsequent visits until the cache expires. However, the application may display an inaccurate balance if a transaction occurs during the 10-minute period. To address this, the application can send a conditional request that says, “Return account 123 data if modified” before using cached data, to ensure accuracy. If the data is modified, it uses the returned updated data and refreshes the cache.

13.4.2 Not letting consumers decide how to cache

Although caching data can reduce API calls and data volume, consumers must not decide how to do it. Outdated cached data can pose significant risks, from mildly irritating users to severe legal complications. For example, as illustrated in section 13.4.1, an incorrect balance could be displayed in our Banking mobile application due to outdated cached data. This could lead to users making wrong financial decisions, resulting in negative balances and fees that could result in complaints, legal action, and a loss of trust in our company.

13.4.3 Defining caching policies based on data and context

To ensure performance and accuracy, it’s crucial to explicitly define caching policies indicating whether and how long consumers can cache data and whether cached data must be validated before use (to ensure that it’s still fresh). To do so, we must consider the data’s composition (not all elements may have the same lifespan), events leading to its modification (leading to cache invalidation), volatility (short or long lifespan), and usage context (sometimes, using old data may be OK). We can also enhance the data to clarify its freshness.

We provide a list of available bank cards and their pricing during the bank account application process. The types of cards won’t change, but their pricing can be updated on the 15th of each month; this data can be cached until this date. The data returned by the “Read account” operation includes the account type, owner name, and balance. Although the account type and owner name are unlikely to change, the balance is affected by transactions. In the past, transactions were processed once a day at midnight, allowing us to cache account data until then. With transactions now processed in real time, the balance can change; the update frequency depends on how active the owner is and the type of account. We can consider allowing caching but enforcing cache validation before using cached data. Suppose the transaction pie chart categorization data shown on the mobile application’s home screen is returned by a dedicated operation. Despite being affected by any new transaction, the data is coarse grained and fuzzy; we can allow it to be cached for several hours or a day, and to mitigate problems, we can indicate when the percentages were calculated.



NOTE Don’t work alone on this; discuss it with subject-matter experts, architects, or tech leads. You must ensure that no security, business, or regulation

concerns prevent client-side caching. Add the defined caching policies in the operation or its 2xx response description in the OpenAPI document, where all stakeholders can easily find it; in particular, the implementation's developers will need it.

13.4.4 Returning cache directives

Operations can return HTTP caching directives to enable interoperable cache management that can be seamlessly used by any HTTP-cache-compliant tool. Figure 13.7 shows three responses with cache directives using the Cache-Control standard HTTP header.

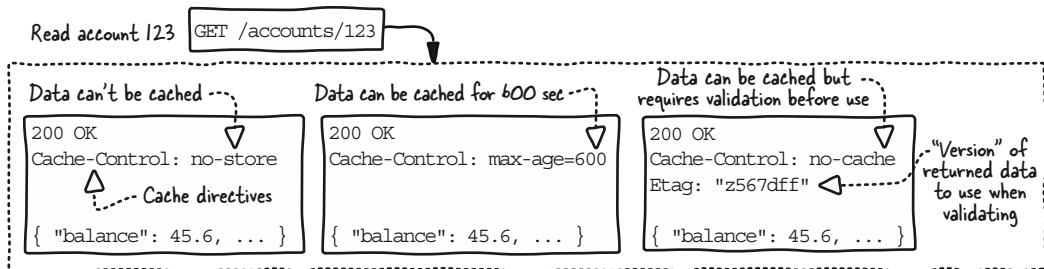


Figure 13.7 The Cache-Control response header contains directives indicating whether and how the returned data can be cached.

The `no-store` value means the account 123 data can't be cached, and `max-age` indicates that the data can be cached for 600 seconds. Contrary to what the name suggests, the `no-cache` value allows caching, but consumers must check its validity before using it. The `Etag` header contains an entity tag (`z567dff`), which identifies the data's version based on changes over time and content negotiation; JSON and XML representations of the same data have different entity tags. An `Etag` can be based, for example, on a hash of the modification date or the entire data; it's up to the implementation to decide what's best. We (or, more precisely, our cache) will validate the data by reading it conditionally with the `Etag` header in section 13.4.5. The `Cache-Control` header can contain multiple directives, and other directives are available. For more information, consult RFC 9111, HTTP Caching (www.rfc-editor.org/rfc/rfc9111.html).

By default, HTTP caches use the URL (including the query) as a key for the cached data. However, this may be insufficient when using content negotiation to propose different formats or languages (section 9.7). For example, data in JSON in French and XML in English would share the same cache key. In that case, the response must include the `Vary` header, with the header names contributing to the cache key calculation: for example, `Vary: Accept, Accept-Language`.

13.4.5 Retrieving data only when modified

Similar to the updates in section 12.5, we can condition readings with HTTP to validate cached data before use and get data only when modified. An HTTP-compliant cache can automatically use this feature.

Figure 13.8 shows how to use `Etag` for validation. When consumers want to check whether cached account data is still valid, they can make a conditional Read account call by including the `If-None-Match` header with the value of the previous `Etag` (`z567dff`). This conditional request means “Read account 123 if it is not modified compared to the `z567dff` version.” If the data is unmodified, the response is `304 Not Modified` with no data. If modified, the response is `200 OK` and includes the updated data and a new `Etag`.

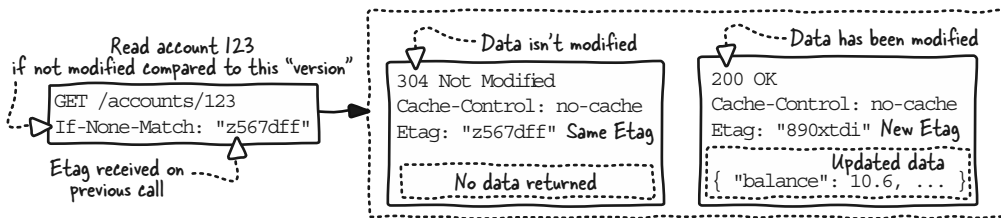


Figure 13.8 The `If-None-Match` header allows us to return data only if it has changed compared to the provided `Etag` value.

We can use `If-Modified-Because`, which expects a date instead of `If-None-Match` and an `Etag`. For more options, consult the Conditional Requests section of RFC 9110, HTTP Semantics (www.rfc-editor.org/rfc/rfc9110.html#name-conditional-requests).



NOTE Remember to add the request and response cache-related headers and new `304` responses in the OpenAPI document.

13.5 Optimizing data volume

Although HTTP compression can significantly reduce data volume (section 13.2.1), we can use various options to optimize data volume by returning more or less data according to the context. Returning more data allows consumers to make fewer API calls and possibly return less data overall. Returning less data means responding faster and reducing cloud infrastructure costs. This section discusses various options:

- Enabling resource model selection
- Toggling the return of the resource on creation or modification
- Enabling field selection
- Centralizing redundant data in dedicated operations
- Considering partial update over total replacement
- Contrasting JSON Merge Patch and JSON Patch for array updates

13.5.1 Enabling resource model selection

Once we choose a data model, especially for lists, we can't change it, potentially leading to too much or insufficient data. Allowing consumers to select the data model for each operation would improve our API's flexibility and efficiency. Figure 13.9 shows how to use content negotiation or the `Prefer` header to enable this choice.

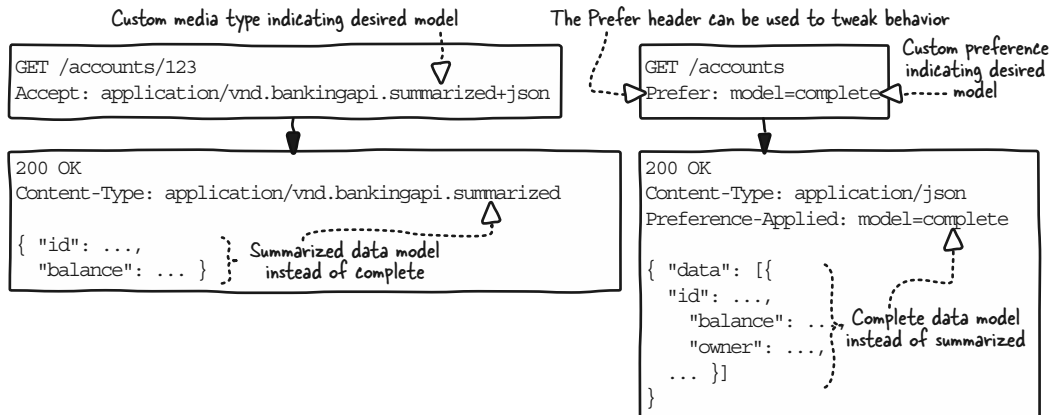


Figure 13.9 We can use a custom media type or preference to get the complete or summarized model on any read, list, or search operation.

We learned about standard media types, such as `application/json` and `text/csv`, in section 9.7.1. HTTP lets us create custom media types. Custom media types used in APIs are often named `application/vnd.something+json`, where `vnd` indicates a vendor-specific type and `+json` indicates that the type is JSON-based (use `+xml` for XML-based data). We can define a custom media type for each model type, such as `application/vnd.bankingapi.summarized+json` for the summarized data model or `application/vnd.bankingapi.complete+json` for the complete one. The API responds with the type of model corresponding to the media type the consumer sent in the `Accept` header. Its `Content-Type` indicates the model used.

Instead of custom media types, we can use the `Prefer` request header to tweak an operation's behavior (defined in RFC 7240, `Prefer Header for HTTP`; www.rfc-editor.org/rfc/rfc7240.html). We define a model custom preference so consumers can send `Prefer: model=summarized` or `Prefer: model=complete` depending on their needs. The response has the usual `Content-Type: application/json` header, but it also contains the `Preference-Applied` header indicating the returned model.



CAUTION Although these API design options are useful and aligned with HTTP specifications, they are uncommon, so use them cautiously. Using a custom media type may surprise developers. The `Prefer` header option allows the usual `application/json` media type to be used.

Both solutions apply to “Read” and “List” or “Search” operations. For example, calling `GET /accounts` with `Prefer: model=complete` returns a list of complete accounts instead of the usual summary. Calling `GET /accounts/123` with `application/vnd.bankingapi.summarized+json` media type returns the summarized account instead of the usual complete one. If no information about the desired model is provided, the operation returns a default model so it behaves as expected. For example, the complete model is returned by default when reading an account.



TIP The model selection helps with reading resources that have multiple IDs. For instance, if a consumer only has an account ID and the “Read account” operation is `GET /accounts/{iban}`, they usually need to search with `GET /accounts?id=123` to get the IBAN and then read the account. Selecting the complete model on search avoids the read call.

13.5.2 *Toggle the return of updated or created resources*

In section 4.6.2, we learned to return resource data on creation or update, but consumers may not always need that data. We can use the `Prefer` header we discussed in section 13.5.1 to toggle the return of the created or updated resource.

For instance, when creating an account application with `POST /account-applications` with `Prefer: model=none` or `Prefer: return-no-content`, the response will be 201 Created with the `Location` header holding the created resource URL but without data. I prefer to avoid returning 204 No Content to keep the information that something was created. The `model=none` option also allows `model=summarized` or `model=minimal`, respectively, leading to the return of the summarized account application model or just its ID (see typical models in section 5.4.1).

13.5.3 *Enabling field selection*

If tuning models is not fine-grained enough, we can allow consumers to specify precisely which fields they need. In section 11.4.3, we introduced hypermedia formats; some propose this feature out of the box. For instance, figure 13.10 shows how to get only the balance of an account with the JSON:API hypermedia format (<https://jsonapi.org/>).

A basic JSON:API document has some links and a data object containing a type (`account`), a unique `id` (123), and attributes holding the account data. By default, `GET /accounts/123` returns all attributes. Only the `balance` attribute is returned if we add the `fields[account]=balance` query parameter. See <https://jsonapi.org/format/#fetching-sparse-fieldsets> for more information. If you’re not using a specific hypermedia format, you can use `fields=balance` to achieve the same result with plain `application/json` data.



NOTE Other API formats like GraphQL and OData also propose field selection. GraphQL is discussed in section 14.8.

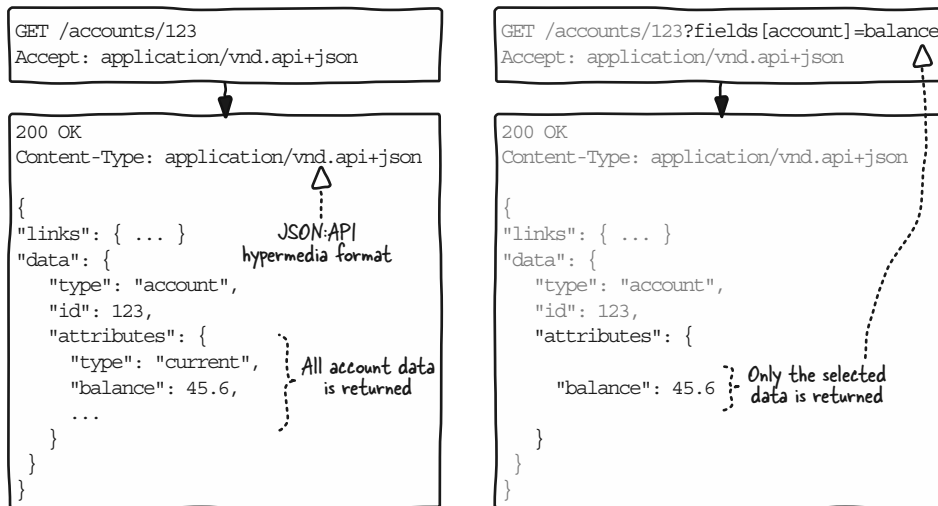


Figure 13.10 If field selection is enabled, consumers can, for example, get only the account balance instead of all the account data.

13.5.4 Centralizing redundant data in dedicated operations

Sometimes our data contains redundant information that appears many times in a response or across responses. We may want to deliver this data through dedicated operations.

In section 8.5.2, we learned to add human-readable labels for codes such as account types and transaction categories. Operations that use these labels may return the same occurrences multiple times; for example, many transactions may have the 123 code with the “Restaurant” label. Additionally, operations will always return them, even if consumers already know what the 123 transaction category code means. We can create a generic GET /codes/{codeName}/values operation to get the labels corresponding to codes, giving GET /codes/transaction-category/values in that case. Such an operation will benefit from being cached (section 13.4). We can also create a more specific GET /transaction-categories operation that returns more data, such as icons and short and long labels that consumers may use depending on the room available in the UI. The downside is that transaction data is a bit less ready to use, requiring an extra call when category codes need to be interpreted, but that may be worth considering at scale.

13.5.5 Considering a partial update over total replacement

Requesting less data means less time spent transmitting and processing it and less load on the infrastructure. Additionally, although data ingress is generally free, cloud providers may bill for it. So, using partial update (PATCH) over total replacement (PUT) can be useful in some cases. A replacement implies sending all of the resource’s data, and a partial update allows consumers to send only modified data. Partially updating a

transaction to mark it as “checked” only requires sending `{"checked": true}`, whereas replacing it requires sending all of its properties. It looks like a negligible optimization, especially when we know that data is compressed over the HTTP connection. But we must think at scale; we don’t have one user checking one transaction. We have hundreds of thousands, if not millions, of users checking 10 times more transactions weekly or monthly.

13.5.6 Contrasting JSON Merge Patch and JSON Patch for array updates

In section 5.3.4, we learned to use the `PATCH` HTTP method for partial updates by sending only the data to update. This technique is called JSON Merge Patch (RFC 7396). When modifying arrays, an alternative method called JSON Patch (RFC 6902) can be used for efficient updates, but I recommend careful consideration before using JSON Patch.

Figure 13.11 contrasts the two formats for updating the home address city of an account owner using a `PATCH /owners/123` request. The JSON Merge Patch version contains the modified elements of the resource: the entire `addresses` array with only the `city` property of the home address modified. The JSON Patch option describes a list of modifications to be performed sequentially on the resource. It uses the `replace` operation (`op`) to set the value of the element indicated by the `path` JSON pointer (`city` property of the first element of `addresses`) to `value`.

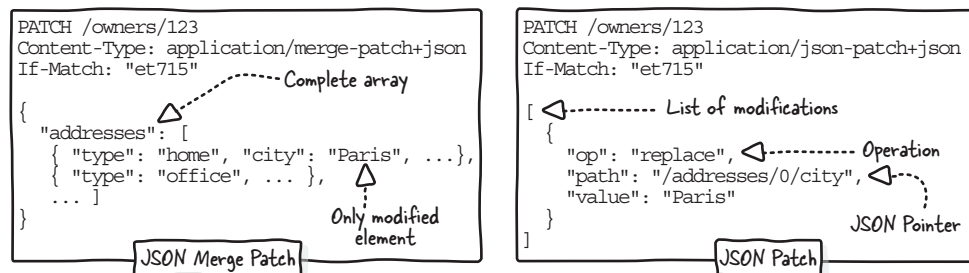


Figure 13.11 Updating the home address city with JSON Merge Patch requires sending the entire address array. With JSON Patch, we only describe the modification of the `city` property of the first element without needing to send the address array.

Although JSON Patch can be more efficient than JSON Merge Patch when updating elements in an array, I do not recommend using it to solve this particular problem or in general. JSON Patch is not commonly used and is complex, whereas JSON Merge Patch is widely adopted and intuitive. Documenting possible JSON Patch operations with OpenAPI and JSON Schema is complex, if not impossible. Regarding array update efficiency, if there are many elements in an array, they should be extracted from the resource and handled via dedicated operations. In our case, we have only two addresses that are not often modified, so it’s unnecessary.

But it's worth noting that thanks to content negotiation, both formats can be used, so users can use the one they prefer if necessary. A `PATCH` update with `application/merge-patch+json` or `application/json` or no `Content-Type` can be treated as a JSON Merge Patch, and one using `application/json-patch+json` can be treated as a JSON Patch. Users can check the JSON Merge Patch option in the OpenAPI-based documentation to know what can be updated and create their JSON Patch request accordingly.

13.6 Optimizing pagination

Enabling list filtering and pagination, as discussed in section 9.6, makes our API efficient by allowing consumers to retrieve tailor-made subsets of data. This prevents systematic retrieval of all data, which may cause the server to hang or crash due to out-of-memory problems and lead to costly cloud-provider bills due to the return of large amounts of data. However, we can do more by working on page size to optimize data volume and favoring cursor-based pagination for better performance.

13.6.1 Optimizing page size limits

We must carefully set page size limits to ensure that our search operations return only the needed data with minimum time and calls while maintaining user-friendliness. In section 13.3.2, the maximum page size of 10 caused an unnecessary load on our infrastructure when consumers needed to retrieve a large number of transactions. To address this, we've increased the maximum page size to 100, which seems reasonable based on the subject matter of account transactions, the use cases covered by our API, and the time needed to transmit the data.

Choosing the appropriate minimum page size is equally important. If it's set too high, such as 50 transactions, consumers who need only 10 transactions will receive 40 unnecessary ones. This can lead to inefficient data retrieval. On the other hand, setting the minimum page size to 1 ensures that consumers get exactly what they need. It allows them, for instance, to get the latest transaction.

Making the page size mandatory would ensure that we didn't return unneeded transactions, but that would make our API a little less user-friendly. We set the default page size to 10, which allows us to return limited but still interesting data for consumers, especially when they're in the discovery phase of the API. Afterward, they can specify a page size that matches their needs.



CAUTION More items per page means more output data and longer downloads, so extending the page size must be carefully considered. Most calls will use the default page size; setting it too high may result in returning unnecessary data and increasing infrastructure costs.

13.6.2 Choosing cursor- or index-based pagination

Choosing between an index and a cursor for pagination is not a question of preference; data, performance, and implementation constrain the choice, but we must not forget to carefully consider user needs. Index- or offset-based pagination uses a page index or offset number of records to retrieve a specific subset of results. A `GET /transfers?page=2&pageSize=10` or `GET /transfers?offset=10&pageSize=10` request returns the second page of 10 money transfers. Cursor-based pagination uses a pointer to the last item of the current page to fetch the next set of results. A `GET /transfers?next=xbzv237&pageSize=10` request returns the 10 transactions after the one identified by the `xbzv237` cursor. Only the server can understand the cursor's opaque value: it's based on unique and sequential data, such as an ID.

The volatility and nature of the data can drive the decision. For example, if a consumer retrieves the last 10 transfers and wants the next page, adding a new transfer causes the last transfer on the retrieved first page to appear on the second page. Additionally, if one of the last 10 transfers is deleted, the first transfer on the second page moves to the first page, and the consumer may miss it. Cursor-based pagination avoids this. However, nonsequential IDs, like UUIDs, can prevent this approach from being used.

From a performance perspective, cursor-based pagination is far more efficient than index-based pagination, which requires the database to load all previous pages. The effect on server resource use and response time is significant for big datasets and deep pages.

At the server-side implementation level, index-based pagination is simpler than cursor-based pagination, especially when search filters are involved. However, although implementing a cursor is complex, it is possible if unique, sequential data is available. Be cautious: the database may not support it. But if such a database is used, you likely won't need cursor-based pagination.

From the client-side perspective, the next page based on an index or cursor fundamentally makes no technical difference. But user needs must also be considered. If consumers must jump directly to page 10, index-based pagination is required. Cursor-based pagination allows consumers to go only to the previous or next page.



NOTE Consider cursor-based pagination for performance unless the requirements involve jumping to a specific page or the data or database is not cursor-compatible. Discuss this topic with architects or tech leads.

13.7 Processing multiple elements with bulk or batch operations

Optimizing call flows can be done by aggregating similar API calls into a single call. An API operation that allows consumers to create, update, replace, or delete multiple elements is often called a *bulk* or *batch* operation. We'll use the term *bulk* in this book. We need these operations for the same reasons we need "Search" and "List" operations. For example, it takes more time and resources to mark 100 accounts' transactions as "checked" with 100

“Update transaction” calls than a single call to “Update transactions.” This section discusses the design and optimization of bulk requests, choosing a bulk operation response policy, designing the response, and partitioning access to bulk operations.



CAUTION A bulk operation’s design may require adaptations because processing multiple elements may take a long time. See section 14.7 to learn how to integrate this constraint into its design.

13.7.1 Designing bulk operation requests

As illustrated in figure 13.12, a bulk operation request contains multiple times the information needed to process one element, including the resource identifier (in a broad sense, path and ID), HTTP method, and resource data. Our banking company collaborates with a partner that submits many bank account applications to us daily. Rather than making individual calls to our POST /account-applications operation for each account application, we modified the request body to allow them to submit 100 account applications in a single call, using a data array consistent with our “Search” or “List” operation response format.

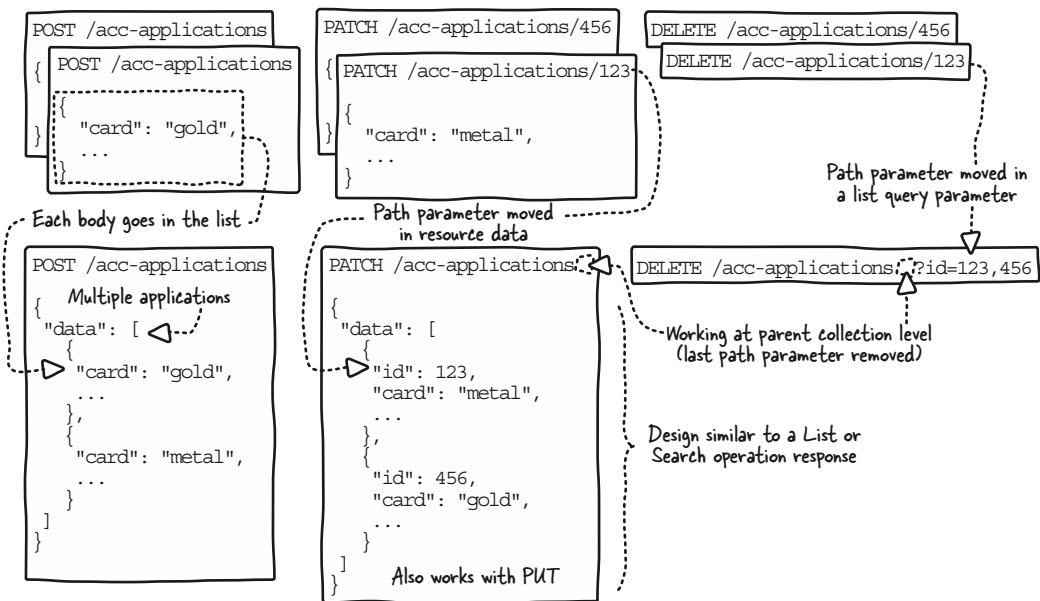


Figure 13.12 The bulk requests body contains the data of multiple resources (if any). If each element needs to be identified, the final path parameter is moved at each resource data level in the body (PATCH or PUT) or in a list query parameter (DELETE).

Suppose the partner needs to modify multiple account applications. In that case, they can work at the collection level and use a single PUT or PATCH /account-applications

instead of multiple `PUT` or `PATCH` `/account-applications/{accountApplicationId}` on individual elements. The body contains a data list where each account application object has an `id` replacing the `accountApplicationId` path parameter to identify it.

Similarly, if the partner needs to cancel account applications that are too old and unfinished, it can call `DELETE /account-applications?id=123,456,...` instead of calling `DELETE /account-applications/{accountApplicationId}` 100 times. Each `accountApplicationId` path parameter goes in the `ids` query parameter list.



TIP By combining `GET /some-resources?id=1,2,3` and model selection from section 13.5.1, we can bulk-read multiple resources.

13.7.2 *Optimizing bulk operation requests*

In the case of partial updates and deletions, we can optimize the bulk operation design so that the consumers send less data and possibly do fewer API calls. In section 13.7.1, partners can cancel old and unfinished account applications using `DELETE /account-applications?id=123,456`. Alternatively, they can cancel account applications based on filters other than IDs using the `DELETE /account-applications?created=lt:2024-05-04&status=IN_PROGRESS` request. This allows them to cancel account applications created before May 4, 2024, with a status of `IN_PROGRESS`.

Alternatively, suppose partners need to change an account application's status to `STALLED`. We could have a `PATCH /account-applications?created=lt:2024-05-04&status=IN_PROGRESS` whose body contains a single object with the new status (`{ "status": "STALLED" }`). Although the bulk `DELETE` operation design is common, the bulk `PATCH` with selection filters and a single input is uncommon but useful to avoid repeating the same data.

However, the best optimized call is the one that is not made. In that specific case, we may question the need for consumers to clean up old and unfinished account applications. Instead, the account application system could automatically handle this, optionally based on a specified timeframe provided by the consumer when initiating the account application.

13.7.3 *Clarifying a bulk operation error policy*

When designing a bulk operation, it's crucial to decide between the all-or-nothing and mixed approaches to handle errors when processing each element. Does a single errored element make the entire request fail? The API designer can't decide alone; subject matter, business, or implementation concerns may drive which approach to use.

For example, when designing the bulk account application, the architect mentioned that rolling back the 99 previously created account applications due to an error on the 100th is impossible without significant system fixes (check section 14.1 for more about such constraints). This forces us to use the mixed approach, which fortunately aligns with our company and partner needs. Our business growth and partner incentives are tied to valid account applications. This approach also

improves data efficiency by eliminating the need for consumers to resend valid account applications.

However, in the context of bulk money transfers, the all-or-nothing approach is not just a choice but a necessity. It's crucial to maintain the integrity of the processing for the set of provided transactions where elements may be interdependent, ensuring a secure, reliable bulk operation.

13.7.4 Designing a mixed response

If we use the mixed error policy, we can use the 207 Multi-Status HTTP status code defined by WebDAV, which is an extension of HTTP (www.rfc-editor.org/rfc/rfc4918). This extension defines a response data format containing a status and data for each processed element. Unfortunately, it's in XML, is specific to WebDAV, and lacks headers. No similar standard exists for JSON-based APIs, but we can use it as an inspiration to define our format. It contains the same data that the regular unitary operation would return for each element, including status, headers, and body data. This format is illustrated in figure 13.13.

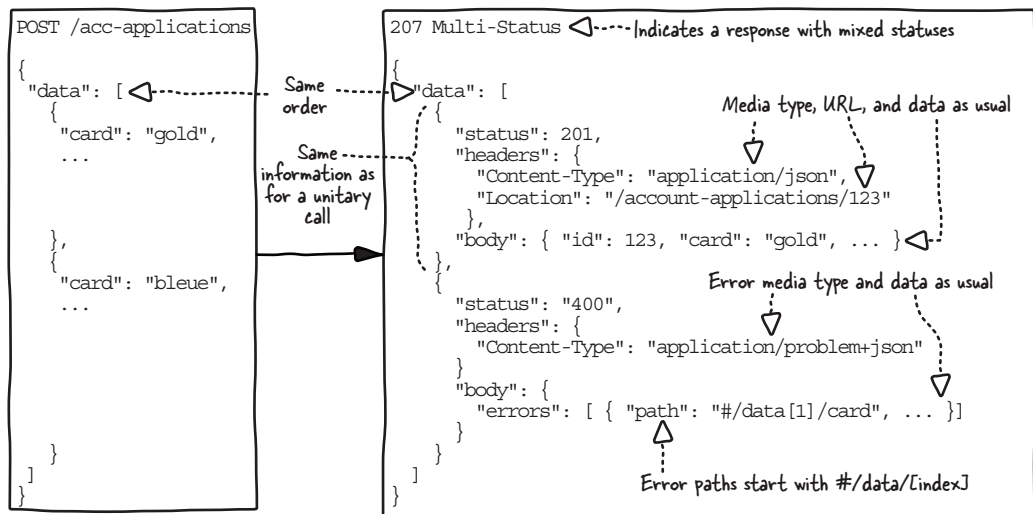


Figure 13.13 Each element of the bulk response contains the same data that a unitary call would return. Response elements are ordered as in the request.

The data array contains elements ordered as in the original request. Each element includes an HTTP status, a headers map, and a body. The first element had no problems, with a status of 201 (Created), standard headers (including Location), and the created resource data in the body. The second call had an invalid value in the card property, resulting in a status of 400 (Bad Request) and a Content-Type of application/problem+json in the headers. The body contains error information, with all paths

starting with `#/data[1]` to indicate the source of the problem. If all elements share the same status, such as 200 OK or 400 Bad Request, the operation returns that status instead of 207.

13.7.5 *Designing an all-or-nothing response*

If we use the all-or-nothing error policy, we can use the same design as the mixed approach so all our bulk operations are consistent. Also, we may be constrained by the fact that even in the case of total success or error, we may need different HTTP statuses and return 207 Multi Status. The response to a PUT can be 200 OK (update) or 201 Created. A specific error can be 400 Bad Request or 403 Forbidden. Additionally, we may need to return a Location header for each created element.

Figure 13.14 shows an alternative design to use if we don't care about element-specific headers or statuses. For success, we can return a data list containing the data for each element. In case of an error, we can return the usual error format, listing all errors of all elements in the same place. Such formats have the advantage of being simple and similar to regular operation responses; still, if we later need to handle a mixed status for another bulk operation and introduce the other design option, we'll end up with inconsistent designs. That can be a tricky decision; check section 16.1 to make it confidently.

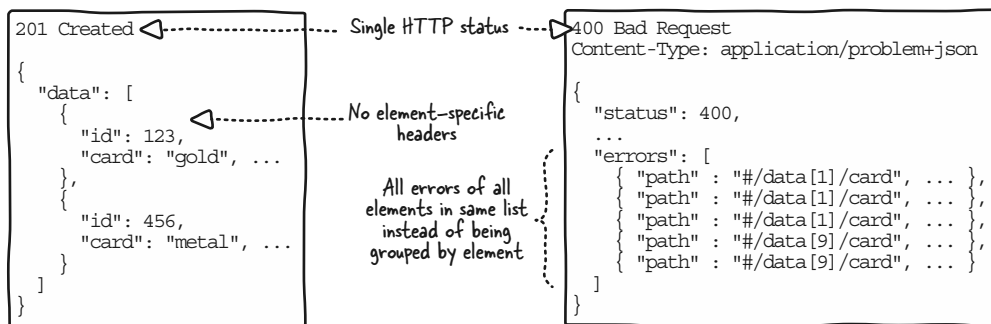


Figure 13.14 Possible design for all-or-nothing operations if we don't care about element-specific HTTP statuses or headers. The errors list contains all errors of all errored elements.

13.7.6 *Optimizing bulk request responses*

As in section 13.5.2, we can use model selection to indicate the desired model for elements returned under data. When we add `Prefer: model=none` to the request, no body property is returned. We can also request just the IDs (`model=minimal`) or summarized data (`model=summarized`). In the case of 207 Multi Status, this doesn't apply to errored elements in data, which will always have their body.

13.7.7 Partitioning access to bulk operations

We may need to reserve access to bulk operations for certain consumers. For instance, creating many account applications in one shot makes sense for a specific partner, but not all of them, or for our mobile application. We can use a behavior scope, such as `account-application:create:bulk`, to allow consumers to use `POST /account-applications` in bulk mode. Although both the partner and mobile applications have the `account-application:create` scope that allows them to call `POST /account-applications`, only the partner application has `account-application:create:bulk` that allows the partner to send more than one account-application. Alternatively, we can have a dedicated `POST /account-applications/bulk` operation accessible via the `account-application:create_bulk` scope.

13.8 Considering a separate optimized API

Suppose heavy optimizations are absolutely needed but lead to an ultra-specific API with resources and operations that make sense only for a specific consumer that needs them. In that case, we must consider not optimizing our versatile API but instead creating an optimized API on top of it. It's common to have different layers of APIs in a system. The typical layers are system, business, and experience APIs.

System APIs are exposed by commercial, open source, or old in-house software; they reveal inner workings and require considerable expertise. They shouldn't be provided to non-experts. The business API layer should ideally be the only layer. It represents the organization's business, subject matter, and domains and hides the system API complexities if needed. The experience layer holds specific APIs that rely on business APIs. Experience APIs are optimized for different purposes, such as new product experimentation or performance. Some experience APIs may evolve into business APIs over time.

In our efficiency case, it's typical to create an experience API called *backend-for-frontend* (BFF) that aggregates and optimizes business APIs for specific consumers, such as mobile applications or websites. Such APIs may use API formats other than REST, such as GraphQL (discussed in section 14.8). The team developing the consumers usually manages these APIs.



CAUTION As when we decide whether to have one or multiple APIs (section 11.2), creating different API layers is a question to discuss with architects. Creating layers and having many components communicating with each other requires discipline and organization to be sure what goes in each layer and component, ensure optimal paths, and limit duplication. The uncontrolled growth of such a distributed system can make it inefficient, brittle, and hard to maintain.

Summary

- Efficient APIs benefit consumers, end users, and providers (speed, minimal battery usage, low data volume, light infrastructure load, and cost-effectiveness).
- To ensure efficiency, use cases should require the fewest API calls, especially sequentially, and handle the least possible data.
- Separate concerns. Design a user-friendly API that does the job first, and then enhance its efficiency.
- Pay attention to locally minor optimizations; they matter at scale.
- Use caching and conditional requests to optimize calls and data volume.
- Define caching policies, indicating whether and how long consumers can cache data to mitigate risks associated with outdated cached data.
- Consider the data's composition, volatility, and usage context to define cache policies.
- Use interoperable HTTP cache and conditional requests so that consumers can save data to reuse it and update their cache only when necessary (Cache-Control, Etag, If-None-Match headers, and 304 Not Modified status).
- Enable the request or return of more or less data to optimize calls and data volume.
- Let consumers decide the model to return with content negotiation or Prefer header to optimize calls and data volume.
- Use hypermedia format field selection or a custom `fields` query parameter to minimize data volume.
- Consider delivering redundant information that appears many times inside or across response data through dedicated operations.
- Consider partial update (PATCH) over total replacement (PUT) to reduce data ingress.
- Favor the simpler and widely adopted JSON Merge Patch format over JSON Patch; use content negotiation to support both.
- Enable list filtering and optimize pagination boundaries from 1 to an efficient maximum to optimize calls and data volume.
- Favor cursor-based pagination for performance unless the requirements involve jumping to a specific page or the data or database is not cursor-compatible.
- Create bulk operations for consumers to efficiently process multiple elements in one call, optimizing call flows and reducing infrastructure load.
- Put resource IDs in resource data for bulk updates; use a query parameter for bulk deletes.
- Consider using search filters instead of IDs to optimize bulk deletes and same-input updates.
- Consider subject matter, business, and implementation concerns when deciding between an all-or-nothing or mixed error policy for bulk operations.

- Return a 207 `Multi-Status` HTTP status if the bulk response contains successes and errors.
- Use behavior scope or a dedicated operation to partition access to bulk operations.
- When a consumer hits its limits, return 429 `Too Many Requests` and a `Retry-After` header. Provide informative `RateLimit` headers on all responses.
- Ensure that HTTP 2 is activated or HTTP 1.1 compression and keep-alive connections are enabled before optimizing API design due to performance problems.
- Create an experience API on top of your business API to keep it versatile if heavy optimization is needed.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 13.1

An e-learning platform that proposes tens of thousands of courses that cover hundreds of domains is powered by an API. The users of the platform search for courses by domain, description, and star rating. Listing 13.1 shows the response of the “List courses” operation (`GET /courses`), which returns all available courses. Listing 13.2 shows the response of the “Read course” operation (`GET /courses/{courseId}`). Can these operations cause efficiency problems when users search for courses, and how can they be fixed?

Listing 13.1 “List courses” response

```
{
  "data": [
    ...
    {
      "id": "course_1005",
      "name": "Advanced Machine Learning"
    },
    {
      "id": "course_1006",
      "name": "Financial Accounting",
    },
    ...
  ]
}
```

Listing 13.2 “Read course” response

```
{
  "id": "course_1005",
  "name": "Advanced Machine Learning",
  "domain": {
    "code": "AI",
    "name": "Artificial Intelligence",
    "description": "The Artificial Intelligence domain focuses ..."
  },
  "stars": 4.9,
  "description": "Dive into the advanced techniques of ...",
  "lessons": [ ... ]
}
```

Exercise 13.2

You’re working on a Library API and have defined the caching policy for the “Search books” operation (`GET /books`) based on the following information:

- The operation returns books matching search criteria covering availability, title, author, genre, and description.
- New books are added on the 12th of each month at 00:00.
- Each book can be returned or reserved at any time, modifying the book’s availability.

What cache-related HTTP headers are returned for a `GET /books` call performed on January 11 at 11:00 a.m.?

Exercise 13.3

In a social network API, the “Timeline” operation returns the threads for a user; listing 13.3 shows its partial definition with OpenAPI. Is there an efficiency problem, and how can it be fixed?

Listing 13.3 “Timeline” operation

```
openapi: 3.1.0
...
paths:
  /threads:
    get:
      summary: Timeline
      parameters:
        - name: q
          in: query
          schema:
            type: string
        - name: next
          in: query
          schema:
            type: string
```



```
- name: limit
  in: query
  schema:
    type: integer
    default: 10000
    maximum: 1000000
responses:
  "200":
    description: Found threads
    ...
```

14

Adapting the API design to the context

This chapter covers

- Challenging or dealing with provider and consumer constraints
- Downloading and uploading files
- Notifying consumers about provider-sourced events with a webhook
- Handling long operations with polling and callbacks
- Considering other API types

Although they both measure time, an everyday watch does not have the same design as a deep-diving watch that's used under high pressure by a person wearing bulky gloves. An everyday object's design must consider various contextual factors to be entirely effective; the same goes for APIs. If the partners using the Shopping API need to be aware of any product price modifications in real time, requiring them to read all products every second to get up-to-date prices is likely not the best design option.

A design that only fulfills user needs and is user-friendly without embracing the full context surrounding the API will fall short. In previous chapters, we've seen

that security and efficiency affect API design, but these are not the only contextual constraints we must consider. The nature of data and services exposed, how they are used, how they are implemented, business considerations, existing systems, who consumes the API, and their limitations are some of the factors that can influence the design of an API. The effects on the API design can range from light modifications of data to significant consequences for operation flows. Consumers may be required to implement APIs for us to call, or we may need to consider types of APIs other than REST.

This chapter first examines the typical concerns that could influence our API design. Next, we illustrate consumer and provider habits and limitations. We then discuss managing files, notifying consumers about events, and handling long operations. Finally, we consider API types other than REST.

14.1 Integrating context into the API design

As shown in figure 14.1, we're still in the constraints layer of API design introduced in section 1.7.3. In addition to security (section 12.1) and efficiency (section 13.1), we must observe our API's full context to achieve an effective design; modification concerns will be discussed in section 15.1. This section examines contextual factors that can affect the design of an API and discusses seeking and challenging these elements before adapting the API design accordingly.

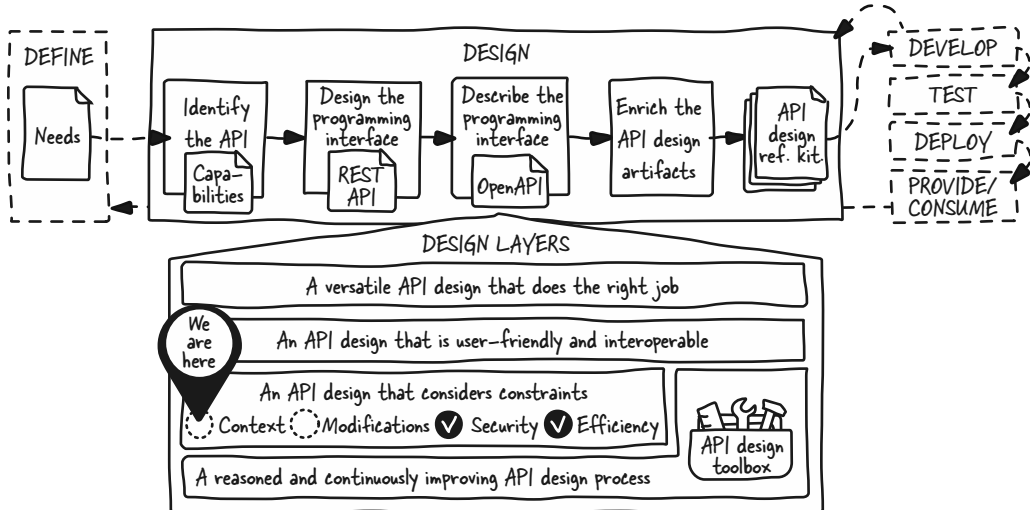


Figure 14.1 Many contextual factors other than security and efficiency constrain API design.

14.1.1 *How context can affect the design of an API*

The design of an API can be affected by the following contextual factors:

- Consumer limitations
- Domain or subject-matter constraints
- How consumers and the provider must communicate
- Nature of the data
- Nature of the operations
- Implementation limitations
- Infrastructure and business constraints

Consumers may have limitations. For example, in the banking world, it's typical to see old systems that can only use the `POST` HTTP method and XML data. A perfectly designed REST API that cleverly uses all HTTP methods and only accepts and returns JSON is unusable for such systems.

The domain or subject matter we work with may constrain us. Banking systems usually use the ISO 20022 financial XML message standard to communicate with other systems via file transfers or APIs. This format constrains API data, operations, and flows. The usual consumer-to-provider communication may not be adapted to user needs, such as “Real-time chat with bank advisor” or “Notifying partners of any new bank account transaction.”

The nature of the data may require adapting the design. Future account owners must provide supporting documents during the account application process, such as ID documents and proof of residence; file uploads can't be handled like sending JSON data. Depending on its location, data may need to be adapted; typically, if our existing transaction ID contains a slash `/`, it will break our URLs when used as a path or query parameter.

An operation can last more than a few seconds because the processing is inherently complex or involves a human being. The automated validation of supporting documents provided during an account application can take up to one minute. In some cases, human validation may be required.

Technically, everything is implementable, but it can be complex. For example, developers at our banking company complain about implementing partial updates in Java, which is “impossible.” We may (or may not) want to make some design trade-offs to work around this limitation.

We may also need to manage infrastructure and business constraints. A single system can't offer the identified capabilities, possibly requiring the creation of multiple APIs instead of just one. The old money transfer subsystem is stopped every night between midnight and 1:00 a.m. to run some internal processing. Additionally, there are a few days during the year when the payment networks between banks are closed, and no money transfers can be performed then.

14.1.2 Seeking constraints and limitations during design

We must identify elements that could influence the design of our API throughout the design process to ensure that our final design will work for the targeted consumers and is implementable. Some elements can be identified from the beginning, and others can be identified later as our understanding of user needs and the design grows. Based on the previous section, here are some typical questions we can ask before the needs analysis and during the entire design process to help determine what constraints and limitations may affect the design of an API:

- Do the identified consumers have limitations compared to how we usually create APIs? (Consumer limitations)
- Are standards or practices defined and used in the industry, organization domain, or subject matter? (Domain constraints)
- Do we need another way of communicating than the usual consumer-to-provider synchronous request and response mechanism offered by HTTP? (Communication)
- Is the data compatible with URLs? (Nature of the data)
- Are files involved? (Nature of the data)
- Are there processes that may take more than a few seconds? (Nature of the operations)
- Are humans involved? (Nature of the operations)
- Is “this” implementable? (Implementation limitations)
- Do the capabilities cover multiple systems, domains, or teams? (Infrastructure and organization limitations)
- Does the system or business run 24/7? (Infrastructure and business limitations)

With experience, you may discover more precise questions or new ones adapted to your environment.

14.1.3 Challenging constraints and limitations

Once an element that may affect the API design is identified, it’s essential to challenge it, as it may be solvable or avoidable. However, not all constraints are “problems.”

We must remember the provider and overly specific consumer perspectives (section 2.6) and how we challenged efficiency problems (section 13.2.4); we must find the root causes of elements that affect the design, but we may not find a problem. For example, is it really “impossible” to implement partial updates in Java? (Spoiler: no.) Or can those consumers actually only use `POST` and XML? (Old banking systems have evolved to support more options.)



NOTE System API-fication often requires revisiting old business processes involving manual controls. These processes may be incompatible with providing APIs to the outside world unless an organization has enough employees to meet all demands, hoping that human processing delays will not be a problem.

Considering the context is not always a question of solving “problems” but simply using appropriate design patterns or tools. For example, to better meet user needs, we may need to consider a type of API other than REST.

14.1.4 **Making trade-offs**

We can adapt the API design and make trade-offs once we’re sure they’re unavoidable. For example, if capabilities span different teams, a single shared API can lead to fragmented ownership and coordination problems. The resulting design may be less than ideal, but separated APIs that ensure clear ownership are best. This example is related to Conway’s law, introduced in section 2.8.3, which states that organizations and IT systems are aligned (for the best or the worst).



NOTE Some design trade-offs may evolve into standardized solutions if similar challenges appear in different places (like `POST /search` to handle sensitive query parameters). We must apply them consistently, which the guidelines will help us do (section 16.3).

The rest of this chapter illustrates and discusses adapting the design to the context (or not):

- Dealing with provider and consumer constraints
- Adapting the design for file downloads and uploads
- Notifying consumers about provider-sourced events with a webhook
- Handling long operations with polling and callbacks
- Considering other API types

14.2 **Dealing with consumer and provider constraints**

We may need to work around or challenge habits and limitations when designing an API. This section discusses a few typical cases:

- Consumers who can only use a subset of HTTP methods
- Consumers who are used to different data formats
- Provider systems or businesses that don’t operate 24/7
- Data and URL compatibility
- Partial update implementation concerns

14.2.1 **Working around consumer HTTP method limitations**

It is now rare to encounter systems that can’t use specific HTTP methods. But it’s still worth discussing this problem to illustrate typical design discussions. Suppose our Banking API is consumed by an old application that needs to create and update money transfers. The application can send a `POST /transfers` HTTP request without difficulty, but it can’t send a `PATCH /transfers/{transferId}` because it doesn’t support the `PATCH` HTTP method.

We can tell this application team that we won’t change our design. They can handle the problem by creating a backend-for-frontend to transform their `POST`-only

request. But many consumers in our system are experiencing similar problems, so we should find a way to fix this once and for all.

We can opt to avoid using PATCH and prioritize PUT instead. This requires sending all data for updates, but we don't anticipate performance problems at scale. Unfortunately, we discover that other consumers cannot use the DELETE HTTP method, so we need to find an alternative approach.

The standard solution in such a case is to support the clearly named X-HTTP-Method-Override custom header (previous options are purely rhetorical). Although not officially defined by any RFC, it's a de facto standard that many HTTP servers support. Instead of sending PATCH /transfers/123, consumers can send POST /transfers/123 with the X-HTTP-Method-Override: PATCH header to update a transfer. To delete it, they send X-HTTP-Method-Override: DELETE.

14.2.2 Accommodating consumers who are used to different data formats

Different consumer categories may want the same API, but the ideal design may differ for each. Sometimes, we can accommodate everyone; other times, we must prioritize a design that suits most consumers, even if it takes extra effort for others.

The bank company plans to offer an API for validating identity and bank account information, which helps corporations and startups prevent fraud. The “Verify account information” operation requires an IBAN, name, and address, returning the account's status and validity; figure 14.2 shows two design options. We can use the ISO

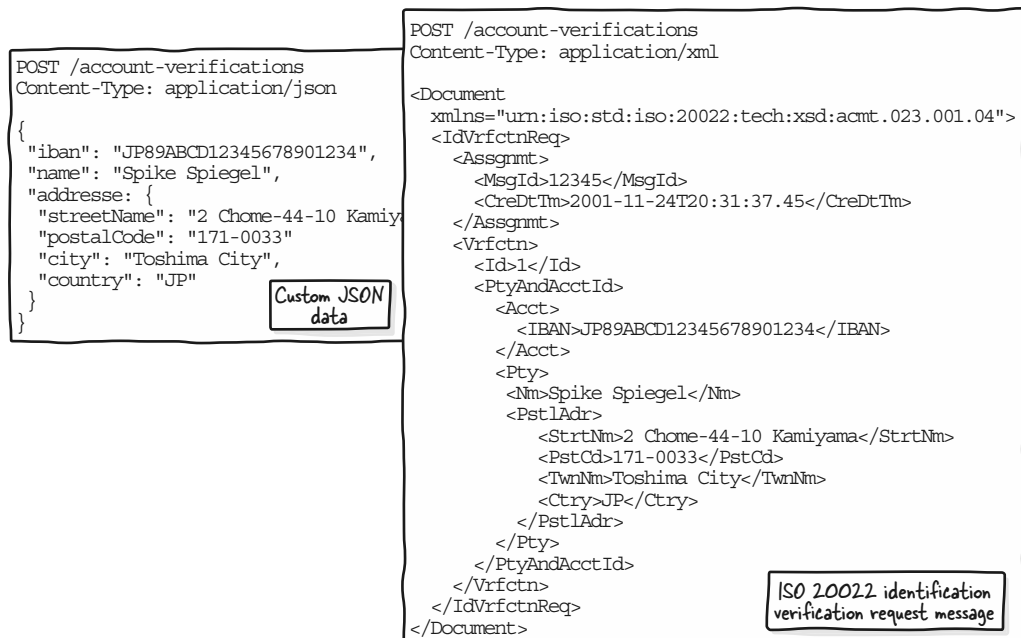


Figure 14.2 The Verify Account Information operation request with custom JSON and standard ISO20022 data

2002 XML standard for requests and responses; it's commonly adopted by large organizations, although it isn't user-friendly for startups accustomed to JSON APIs. We can use content negotiation to allow users to select their preferred format with an `Accept` header for either `application/xml` or `application/json`, pleasing both targets. If other use cases arise, we may need to reconsider, because ISO 20022 could constrain the design and capabilities.

Creating separate APIs for different markets and creating a single API using custom JSON are possible solutions. We must return to the Define phase of the API lifecycle to clarify our objectives with stakeholders.

14.2.3 *Managing planned interruptions*

Although APIs are expected to be constantly available, they may need maintenance, or the business behind them may not be available. If we can't avoid such unavailability, we must adapt our design.

The bank cannot process money transfers between midnight and 1:00 a.m. due to scheduled batch processing of account data. Additionally, interbank payment networks are down for maintenance on holidays like New Year's Day (January 1). Modifying the transfer system may help with nightly problems, but interbank downtime is unavoidable.

If the money transfer capability is unavailable, we can return a `503 Service Unavailable` response for `POST /transfer` calls. To improve the user experience, we include the `Retry-After` header to indicate when transfers will resume. For instance, if a transfer occurs on a banking holiday, we can set `Retry-After` to `Tue, 02 Jan 2024 00:00:00 GMT` (HTTP date) or `3600` (duration in seconds). We can also provide an application problem for more information about the error (section 9.8.6).

Alternatively, we can accept valid money transfer requests for later processing without returning an error. In this case, we return `202 Accepted` with `status` as `DELAYED` and `processingDate` set to January 2, instead of the usual `201 Created`. This requires all the necessary checks before executing the transfer, which our tech lead confirms is feasible. Although there's a risk of an insufficient balance at execution, this is similar to a scheduled transfer, and we are willing to accept the possibility.

14.2.4 *Ensuring data and URL compatibility*

When adding segments, path parameters, and query parameters to URLs, we must keep in mind that not all characters are allowed, and URLs can't be too long. Path and query parameter values with special characters like `/` and `?` must be URL encoded. Most programming languages that support HTTP have built-in encoding functions, such as JavaScript's `encodeURIComponent()`. However, it's best to avoid these characters in API data used in URLs, particularly IDs. For instance, if `GET /transactions` returns "123/456" transaction IDs, consumers must encode them to access the transaction. URL-encoding IDs in responses may cause interoperability problems, making them unusable if not decoded. If the IDs are internal and system-specific,

we can transform them into a format like “123456” or “123-456” at the API level, as existing data may not be fixable.

HTTP doesn’t set a maximum URL size, but implementations do. URLs over 2,000 characters can be problematic because browsers, servers, or proxies may reject them. For our Banking API, a `GET /transactions?transactionId=ID1,ID2,ID3` filtering 200 transaction IDs could approach this limit if the IDs are 10 characters long. If necessary, we can use the `/search` pattern discussed for security in section 12.6.2. A `POST /transactions/search` can handle 200 IDs easily. However, we may also decide that consumers can make two separate calls.

14.2.5 Implementing partial updates

In rare cases, some developers may propose using `PUT` or JSON Patch over JSON Merge Patch, believing the latter is unimplementable due to their language’s inability to differentiate between `"key": null` and a missing `key` in JSON. This concern is unfounded; any language can handle it natively or with JSON libraries. Furthermore, using a less user-friendly JSON Patch instead of the nearly standard and user-friendly JSON Merge Patch would unnecessarily complicate our API, deviating from what people are used to. However, remember that we can offer both with content negotiation (section 13.5.7).

14.3 Handling data and files

It’s common to need to handle files such as images, PDFs, and videos in web APIs. In our Banking API, we may need to provide pictograms for account transaction categories, return bank account statement PDFs, or require a photo as proof of residence to open a bank account. Although HTTP doesn’t differentiate between a JSON or PDF body (section 9.7.1), we must treat files slightly differently than the data we’ve been working with. Security, architecture, and efficiency concerns often constrain how we handle files in our API, affecting data, operations, and flow design.



CAUTION File uploads must always be checked for viruses and malware. Your organization has likely defined and standardized secure file uploads. If not, this must be defined and standardized. In both situations, contact your security team.

This section focuses on handling data and files in operations and flows:

- Collecting data and files in a flow
- Sending and retrieving data and files with a single call
- Describing files and a mix of data and files with OpenAPI



NOTE Managing files requires efficiency (section 14.4) and is often delegated to another system (section 14.5).

14.3.1 Collecting data and files in a flow

Flows often include uploading files in addition to collecting data. In section 10.4, we designed a flexible data-saving flow for opening a bank account, incorporating `POST /account-applications` and `PUT /account-applications/{accountApplicationId}`. These operations allow for creating an account application in one step or gathering information incrementally; let's modify it to collect documents, such as an ID document or proof of residence. This is typically done by

- Uploading files directly attached to a business object
- Uploading generic files and attaching them to a business object



NOTE These two options imply that all data (including files) can be collected gradually, which may not be the case (section 14.3.2). These flows may be affected when file management is delegated to a different system (section 14.5).

We can directly attach a file to the “Account Application” business resource. In this case, a proof of residence PDF document can be sent in the request body of `PUT /account-applications/12345/documents/residence`, where `residence` is a document type used as an identifier. Calling `PUT` again replaces it. Checking the uploaded file can be done with `GET` and removing it with `DELETE`.

Alternatively, we can use a generic file resource, possibly hosted in a dedicated File API. A `POST /files` request saves a file and returns an ID, which can be used as the `fileId` to add a document to the account application via an operation such as `PUT /account-applications/{accountApplicationId}` or `PUT /account-applications/{accountApplicationId}/documents/{documentType}`. To verify an uploaded file, a consumer can call `GET /files/{fileId}` with the document's `fileId`. Replacing a document means uploading a new file and updating the document list. Deleting a document removes it from the account application data. The File API implementation cleans up orphan files.



NOTE The generic file option allows files to be reused for different purposes. For example, the same uploaded video can be processed to detect kittens or generate a textual summary without reuploading it. This also helps centralize all file-related functions.

14.3.2 Sending data and files with a single call

Collecting information gradually may not be possible. Consuming applications may only be able to send a single call with all data, or the implementation may not be able to store resources in an intermediary state. Such limitations must be challenged but may not be solvable (section 14.1.3). Mixing files and data on upload is feasible but has downsides. Figure 14.3 shows two options: a Base64-encoded file in JSON data and a `multipart/form-data` media type.

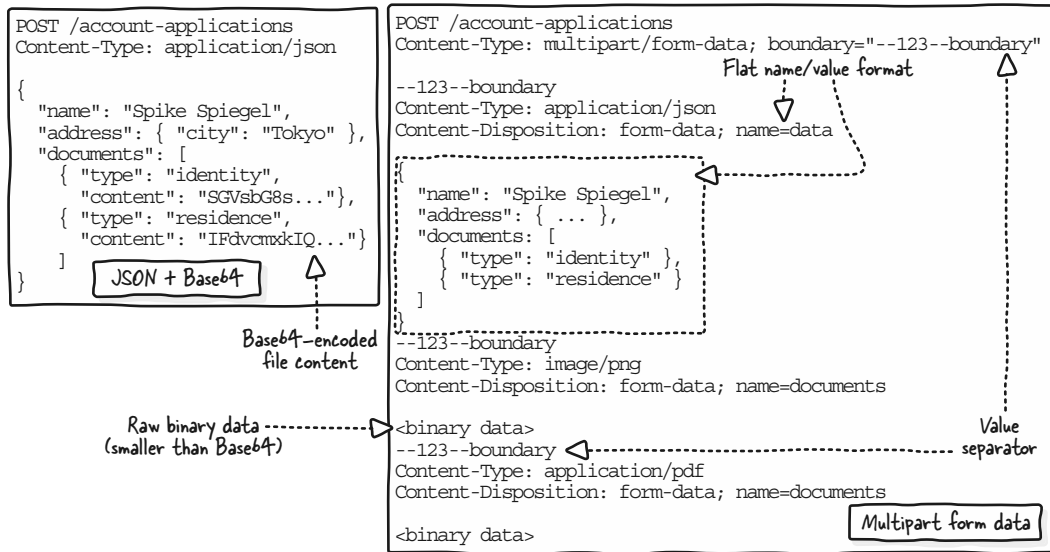


Figure 14.3 We can send a mix of data and files by encoding file content in Base64 strings in JSON or using the multipart/form-data media type.



CAUTION Base64 turns binary data into text but increases the content size by 33%. HTTP compression limits this over the network, but the increase will be around 5 to 10% for JPG files, for example.

Consumers can send all account application data and files with a single `POST /account-applications` request that includes a `documents` array. Each document has `type` set to “identity” or “residence” and a `content` property for Base64-encoded content.

Alternatively, we can use a more standard `multipart/form-data` to mix data and raw binary files (smaller than Base64-encoded content). Each element is separated by the boundary indicated in the `Content-Type` header and handled by HTTP libraries. A part usually represents an HTML form field, such as an applicant name (“Spike Spiegel”) or a file (residence PDF). However, parts can have any media type, allowing complex data structures, such as the data object, which is `application/json`. If two parts share the same name, such as `documents`, they are considered an array. Here, the implementation will map it with `data.documents` to get each file type (“identity” or “residence”).

These approaches have downsides. Data and files need to be re-uploaded if errors occur (such as missing properties in the data). Uploading multiple files can quickly hit request size limits. Some legacy systems, like our old API gateway and banking apps, may not handle multipart or Base64 content. However, combining files and data works in simple cases, like sending a file with metadata that doesn’t fit in the URL or headers; I suggest the multipart option in this case.

14.3.3 Retrieving data and files with a single call

We can mix data and files in responses if we consider them inseparable or if consumers can't retrieve files independently. This must be challenged (section 14.1.3) because embedded files are always retrieved and may not be in the most helpful format. Typically, we embed Base64-encoded files as strings in JSON, as in section 14.3.2. Although we could use the `multipart/mixed` format similar to `multipart/form-data` for responses, I do not recommend it as it's uncommon and complicates client-side implementation.

Figure 14.4 shows the “Read account” operation (`GET /accounts/123`) returning account data, including an icon URL. Because the icon is a small, user-defined file, we could return a Base64-encoded `icon` string instead of `iconUrl`. Consumers could decode it rather than make an additional API call: but was the extra call, which could be cached, really a problem? The downside becomes more apparent with multiple generic files. If we add a category icon to each transaction, consumers will download redundant Base64-encoded files, as many transactions share categories. Also, remember that Base64 increases the content size.

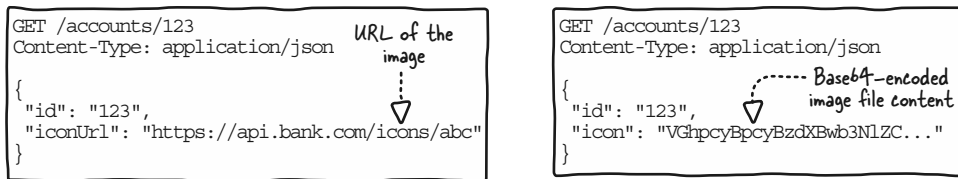


Figure 14.4 Base64 encodes any binary or textual data into a string; it can be embedded into a JSON string.

Separating files from data allows each element to be retrieved individually and, if cache and conditional requests are enabled (section 14.4.1), only when modified. Consumers can obtain images in the desired size and format using an `Accept` header and `width` and `height` query parameters. But using a resizable format like SVG when embedding images with Base64 can address sizing limitations if consumers support it.



TIP Use content negotiation (section 9.7.1) to retrieve a file or data. A request with the `Accept: application/json` header returns data; one without it returns the raw file content.

14.3.4 Describing files with OpenAPI

Describing files in OpenAPI is straightforward. Just indicate the relevant media type(s) under `content` without any other information (empty object) when the request or response body accepts raw files. It's also possible to use wildcards such as `image/*`. For an arbitrary binary file, we can use `application/octet-stream`. Listing 14.1 shows

that the `POST /files` operation accepts PDF, PNG, and JPG files. In listing 14.2, reading a file (`GET /files/{fileId}`) returns the same media types plus `application/json` to return file metadata.

Listing 14.1 Sending a file in a request body

```
paths:
  /files:
    post:
      requestBody:
        content:
          application/pdf: {}
          image/png: {}
          image/jpeg: {}
      responses:
        "201":
          description: File uploaded
```

PDFs, PNGs, and JPGs are accepted.

Returns the Location header and file ID as usual

Listing 14.2 Returning a file or metadata

```
paths:
  /files/{fileId}:
    parameters: ...
    get:
      responses:
        "200":
          description: File content or metadata
          content:
            application/pdf: {}
            image/png: {}
            image/jpeg: {}
            application/json:
              schema:
                type: object
              ...
```

Raw file data

File metadata



TIP Libraries like `file-type` in NodeJS and TypeScript can help check that binary content matches the media type, which type is `application/octet-stream` content, and which media type should be sent for a binary file.

14.3.5 Describing mixed data and files with OpenAPI

This section illustrates how to describe bodies mixing data and files in OpenAPI 3.1 when we are

- Embedding Base64-encoded files in JSON
- Mixing raw files and data with a multipart request body



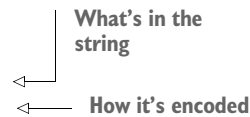
NOTE Handling files and mixed content varies slightly between OpenAPI 3.0 and 3.1. For more information, refer to the “Considerations for File Uploads”

section of the OpenAPI documentation: <https://spec.openapis.org/oas/v3.0.3#considerations-for-file-uploads> (3.0) and <https://spec.openapis.org/oas/v3.1.0#considerations-for-file-uploads> (3.1).

When mixing data and Base64-encoded files, use JSON Schema's `contentType` and `encoding`. In listing 14.3, the `icon` property of the `Account` schema is a PNG image encoded in Base64. Similarly, we would set `contentType` to `image/svg+xml` for an SVG image that uses a text-based image format (XML).

Listing 14.3 Embedding a file in JSON

```
...
components:
  schemas:
    Account:
      properties:
        ...
        icon:
          type: string
          contentType: image/png
          encoding: base64
        ...
```



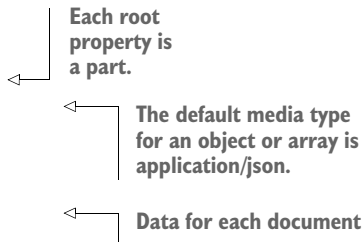
What's in the string

How it's encoded

We could use a similar approach to define each part's media type in a multipart/form-data request body, but we'll use an OpenAPI encoding object instead. In listing 14.4, each root-level property of the request body schema becomes a part and has a specific media type. The data object defaults to `application/json` (whereas an atomic value is `text/plain`). The documents array of binary elements results in parts sharing the documents name and a generic `application/octet-stream` media type. We include an encoding object beside schema as a comma-separated list to specify allowed media types for documents. Additionally, a documents array in data allows consumers to access details for each document, such as their type (ID document or proof of residence), by using item indexes.

Listing 14.4 Defining a multipart request body

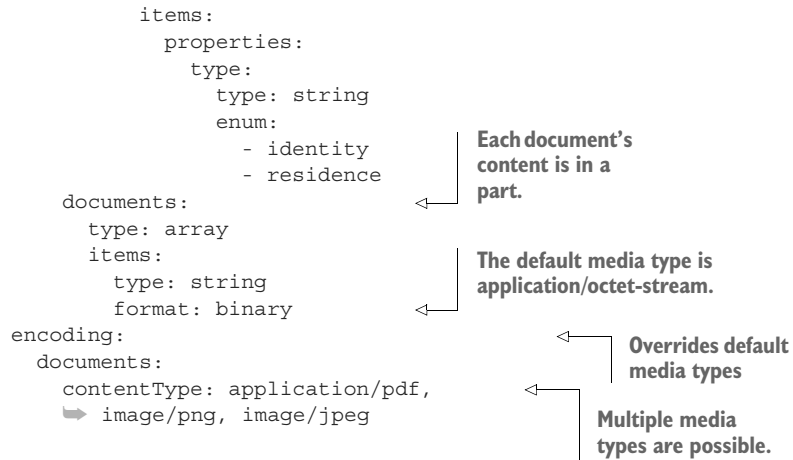
```
paths:
  /account-applications:
    post:
      requestBody:
        content:
          multipart/form-data:
            schema:
              properties:
                data:
                  properties:
                    name:
                      type: string
                    documents:
                      type: array
```



Each root property is a part.

The default media type for an object or array is `application/json`.

Data for each document



14.4 Providing efficient file management features

Efficiency was already a concern for us (section 13.1), but it is even more important when retrieving or sending files. They can be larger than the usual data and change less often. This section discusses the following:

- Returning file data only when necessary
- Enabling partial downloads and uploads
- Preventing unnecessary uploads



CAUTION This section illustrates typical file management features based on our previous learnings and using HTTP. However, some of them may be complex to implement. Using a cloud or on-premises file management or storage system helps implement secure, standard (or at least familiar) file upload and download features (section 14.5).

14.4.1 Returning file data only when necessary

As seen in section 13.4, we should enable cache and conditional requests on file reading. This can prevent the January 2024 PDF statement of account 12345 from being downloaded again if it hasn't been modified. Following the recommendations in section 13.5, we won't return the uploaded file on creation or update. When uploading the proof of residence file during the account application, the response only acknowledges the upload and returns an ID for later use if needed.

14.4.2 Enabling partial downloads and uploads

Consumers may need to resume interrupted downloads, progressively load media, or download multiple file parts in parallel. We can use the `Range` standard request header (discarded for pagination in section 9.6.6). For example, by adding `Range: bytes=0-10485759` to its request, a consumer can retrieve the first 10 MB of a zip archive containing all available account statements. The server responds with 206

Partial Content with the request bytes if the range is valid or 416 Range Not Satisfiable if it isn't.

Similarly, consumers may need to resume interrupted uploads or upload multiple parts simultaneously. We can use the Content-Range request header. A POST /files with a Content-Range: 0-500/1000 header indicates a partial upload of the first 500 bytes of a 1,000-byte file. The server responds with 202 Accepted and a Location header with the partially created resource URL. The consumer uses the returned URL to send a subsequent PUT /files/12345 with a Content-Range: 501-1000/1000 header, ending the file upload.



NOTE Partial downloads and uploads make sense when dealing with large files. However, implementing them can be complex. Storage solutions usually handle these features but may use custom solutions instead of HTTP features.

14.4.3 Preventing unnecessary uploads

If the server rejects the request, a file could be uploaded for nothing. To prevent this, we can inform the consumers about our expectations and enable prechecks before the file upload.

It would be inefficient to upload a 200 MB proof of residence GIF image with POST /account-applications/12345/documents, only to realize that the authorization token has expired or the consumer lacks a relevant scope (Authorization header), the path doesn't exist or is inaccessible to the consumer for security reasons, GIF files are not allowed (Content-Type header), or the maximum allowed size is 100 MB (Content-Length header).



NOTE The operation can return a 413 Content Too Large status if a file size exceeds the server limit. This can be used for any request's content, such as a PDF file or JSON data.

When enhancing the account application flow for flexibility, we likely added a use-case-specific operation indicating the expected documents for an account application (section 10.3.5). We could add the accepted file types and sizes to prevent related problems. However, the upload can fail if consumers don't use this operation or because of security problems. As a last resort, we can use the 100 Continue HTTP status as follows:

- 1 The consumer sends an initial request with an Expect: 100-continue header and other usual headers without body data.
- 2 The server checks the request based on available data (method, URL, headers).
- 3 If the checks are OK, the server responds with a 100 Continue status; if not, the server responds with the adapted 4xx status.
- 4 On receiving 100 Continue, the consumer sends the body data.
- 5 Once the body is received, the server sends the final HTTP status, an adapted 2xx or 4xx, depending on the processing of the file.



NOTE As for partial downloads and uploads, enabling prechecks on upload makes sense when dealing with large files.

14.5 Delegating file downloads and uploads

File uploads and downloads are commonly delegated to a file management system for security, performance, or architectural reasons; doing so typically prevents reinventing the file management wheel (section 14.4). Ideally, the implementation should hide this from consumers. However, consumers sometimes need to interact directly with the file management system, which may have security mechanisms different from our APIs. This section illustrates why and how to manage such a constraint on downloads and uploads.

14.5.1 Downloading files from another system

As shown in figure 14.5, to work around limitations preventing direct download, we can redirect consumers. Our Banking API platform team forbids binary file uploads or downloads through our on-premises API gateway. This gateway reads requests fully before forwarding them. This isn't a problem with JSON data, but it requires too many resources for larger files.

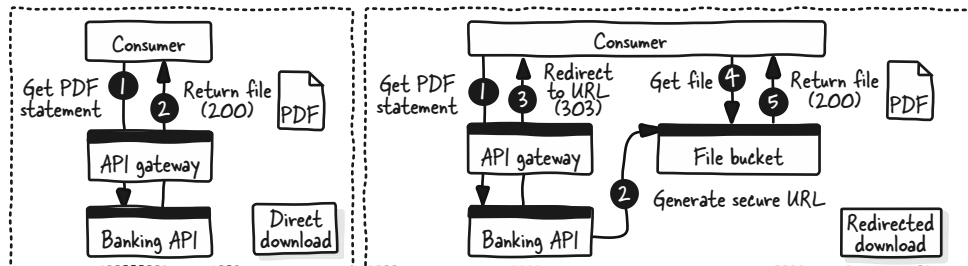


Figure 14.5 When we can't directly return file content to the consumer, we can use a 303 See Other redirection response to redirect consumers to the file.

Consumers should download monthly PDF statements via the Banking API; however, due to the gateway's limitation, we can't return 200 OK with PDF content on a `GET /accounts/123/statements/2024-01` request. Because our files are stored in an AWS S3 bucket (a third-party cloud storage service), the implementation can use the AWS S3 API to generate a secure, single-use, time-limited file access URL known as a *signed* or *presigned* URL. It is returned in the `Location` header of a 303 See Other redirection for download. Consumers don't need to know the URL's destination or structure; they just need to follow the redirection.

14.5.2 Uploading files to another system

As shown in figure 14.6, if a direct upload is impossible, we can use a regular API call to obtain an upload request description, including at least a URL, and then upload the file to another system. Our API gateway limitations prevent consumers from directly uploading documents during account application, such as sending proof of residence via `PUT /account-applications/123/documents/residence`. The API implementation can generate an AWS S3 signed URL for upload, but unlike the case of downloads, HTTP redirection isn't possible. It requires routing files through the gateway on the initial call. Consumers can use a dedicated operation to generate the signed URL, such as `POST /account-applications/123/documents/proof-of-residence/upload-requests`. This operation returns 200 OK with the POST or PUT method, URL, and upload headers because the URL alone may not suffice. A signed URL enforces the HTTP method, which may differ from typical usage or not be POST. It may also require specific headers, such as `application/octet-stream` instead of `application/pdf`. This ready-to-use information (section 8.4) helps consumers upload files without errors from hardcoded configurations.

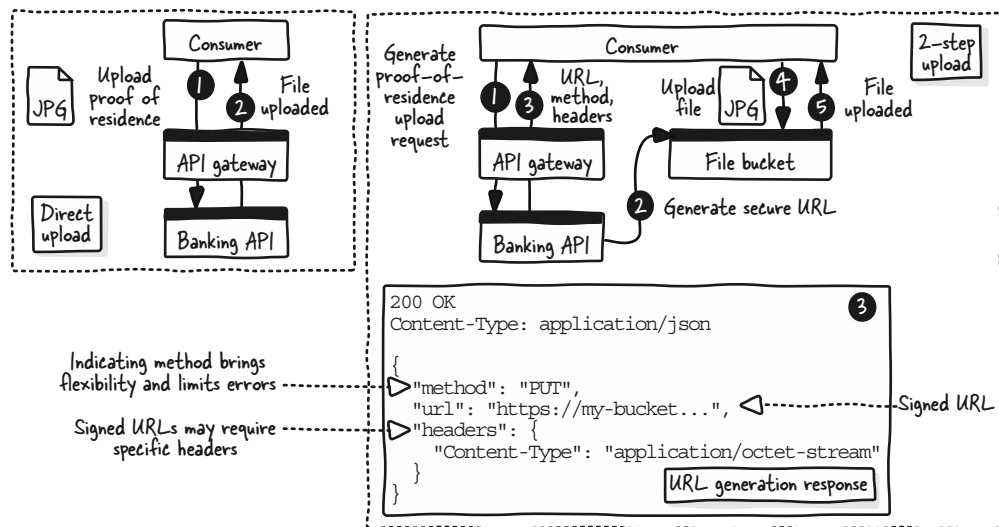


Figure 14.6 When we can't directly receive file content, we can provide an operation that generates a URL targeting a system that can receive it in the place of our API.

This pattern also works with the generic “File” resource from section 14.3.1. Instead of calling `POST /files`, which returns a file ID to be used when adding the proof of residence to the account application, consumers call `POST /file-upload-requests`. In that case, the file ID can be returned on request generation or upload calls. In this situation, the generic File API prevents the business APIs from being polluted with operations created because of our system's architecture.

14.6 Notifying consumers about provider-sourced events with a webhook

We’ve learned to design APIs that allow consumers to call a provider. However, in some cases, API providers need to contact consumers to notify them about events; the typical solution to this problem is a webhook. This section explains what a webhook is, why it should be considered, how to design it, the related features, and how to describe it with OpenAPI.

14.6.1 What is a webhook, and why should we consider using one?

A *webhook* is a “reverse web API” that enables an API provider to notify consumers about events occurring on its end. Unlike regular API calls initiated by consumers, webhooks allow the provider to call the consumer. The API provider defines the webhook interface that consumers implement. Events may occur independently of consumer interactions, and consumers can be notified in real time. Webhooks help avoid inefficient polling (consumers repeatedly calling an operation), reducing unnecessary infrastructure load.

Figure 14.7 shows that third parties using our Banking API need to be updated about new transactions on their users’ bank accounts. Transactions can occur any time (instant payments and debit card use, for example). Although consumers can call the “List transactions” operation every second, it often returns no new data, risking infrastructure overload without rate-limiting (section 13.2.2). Conversely, by calling less frequently, consumers risk getting transaction information long after it happens.

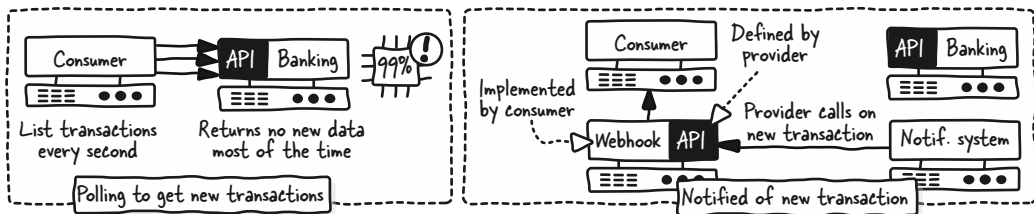


Figure 14.7 The banking notification system calls the Transaction Notification webhook to prevent the consumers from polling transactions.

To prevent these problems, we can define a transaction notification webhook. A third party interested in receiving transactions in real time can implement it according to our specifications and expose it under a consumer-defined address, such as `https://consumer.com/notifications`. Once they have configured the webhook address on our Banking API developer portal, we can send a POST request whose body contains transaction information, as defined in our specifications, as soon as it occurs.



NOTE Section 14.7.2 discusses how long operations can benefit from a similar reverse API mechanism called a *callback*. Section 14.8 discusses API types

other than REST that can be used in consumer-to-provider communication scenarios.

14.6.2 Webhooks should be optional

Not all consumers can implement a webhook. It should be optional if similar actions can be done through regular API calls (at the expense of consumers being notified in real time) and without infrastructure problems. Addressing these concerns may lead to revising user needs, use cases, or operations.

Some Banking API consumers can't implement webhooks. A few have mobile applications with no backend and prefer not to set up a server. Others face challenges exposing APIs because of their infrastructure or security. Nevertheless, they can access user account transactions via `GET /accounts/{accountId}/transactions`, which is inefficient and doesn't meet their needs: retrieving any new transaction. We can introduce a `GET /transactions` operation to fetch all transactions across accessible accounts, allowing them to retrieve any new transactions since the last one. Although this isn't in real time and may lead to excessive calls, it's a more efficient solution.

14.6.3 Designing a webhook operation

Designing a webhook is similar to how we design regular operations. It must meet user needs, be user-friendly, and ensure efficiency and security. We also need to describe it using OpenAPI. However, there are a few differences. Webhooks always use the `POST` method, with consumers choosing the operation path (and implementing the webhook). Furthermore, we can enhance our API ecosystem to simplify webhook use. In the following sections, we discuss the following:

- Using a standard event format to be consistent and interoperable and simplify implementation
- Deciding which data to put in an event, depending on the nature of the data and its usage
- Securing a webhook by ensuring that it only deals with necessary data and can only be called by the API provider
- Defining the webhook behavior on success
- Dealing with webhook failures by enhancing the webhook call implementation, API, or developer portal
- Describing a webhook with OpenAPI

14.6.4 Using a standard event format

Instead of creating an event format, I suggest using the CloudEvents standard (<https://cloudevents.io/>) for interoperability and to streamline work for us and consumers. It provides a protocol-agnostic definition of events compatible with various technologies. For more information, see the specification documentation at <https://github.com/cloudevents/spec>. Figure 14.8 illustrates a webhook call to notify a consumer about a new transaction using this format.

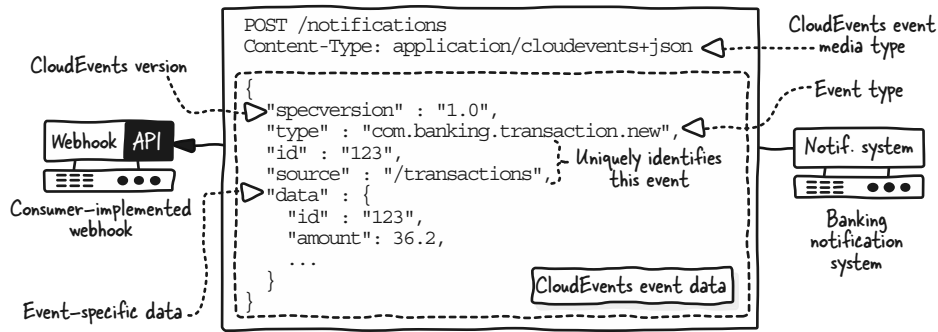


Figure 14.8 The banking notification system notifies a consumer about a new transaction with a POST request whose body uses the CloudEvents format.

Following the CloudEvents specification, we set the request's Content-Type header to `application/cloudevents+json`. The request body is a JSON object describing a unique event. Alternatively, we could have used `application/cloudevents-batch+json` to send a JSON array describing multiple events. But in our case, we want to send each transaction in real time as it occurs.

The event object contains a `specversion` field indicating the CloudEvents format version (1.0). The `type` represents the event type and the originating organization with a reverse domain-like name (`com.banking.transaction.new`). The concatenation of `source` and `id` must uniquely identify an event. The `source` set to the transactions collection path (`/transactions`) is a URI identifying the event's context. The `id` set to the transaction ID (`123456`) uniquely identifies the event within the context. As a comparison, for a `com.banking.account.balance.update` event notifying the consumer about an account's balance update, the `source` could be set to `/accounts/654321` (account resource path) and the `id` to a timestamp such as `1719061768`. The `data` property contains event-specific data related to our `com.banking.transaction.new` event (discussed in the following section).

14.6.5 Choosing event data granularity

As shown in figure 14.9, when designing event data, the `data` property of the CloudEvents format, we must choose between creating a minimal, light, or thin event that contains a pointer to data accessible via a regular API call and a complete, heavy, or thick event that includes more or less usable data. The choice depends on the volume and volatility of the data and its usage. Data sensitivity may affect the decision, too; see section 14.6.6.

We chose the complete option for our transaction notification event, equivalent to the response of `GET /transactions/123456`. This prevents consumers from needing to read each new transaction after receiving the event. This option is valid because transaction data is static and small.

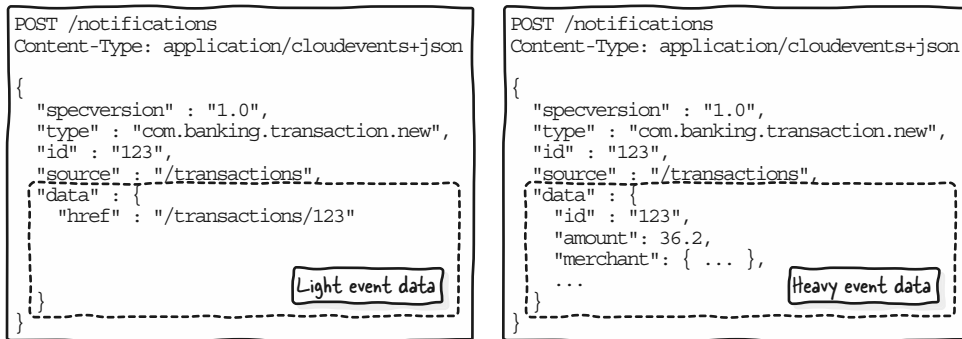


Figure 14.9 A light event only contains a reference that can be used to get data via a regular API call. A heavy event contains usable data and may not necessitate an API call.

For volatile data, a minimal event is better. For instance, when notifying the consumer about an account's balance update, we will send a minimal event with a `href` to the bank account (`/accounts/12345`) instead of the balance, which can change quickly. However, if consumers want to track balance changes, including it may make sense.

To ensure efficiency, events shouldn't carry large volumes of data. If we generate yearly statements for corporate customers and notify them when the statements are available, including statement data in the events would make them too large.



NOTE The CloudEvents format recommends a maximum size of 64 KB for the entire event (data and other properties) to ensure compatibility across different protocols and allow for a broader distribution.

14.6.6 *Designing a secure webhook*

Like a regular operation, a webhook must be designed with security in mind. It should expose only necessary, ideally non-sensitive, data and be accessible only to authorized consumers.

We previously considered transaction `merchant` data sensitive and restricted its access to specific consumers through a behavioral tweak (see section 12.3.3). We must ensure that a similar behavior is implemented when we send transaction notification events. Alternatively, we could use a light event that only references the new transaction; but consumers would need to read the transaction, and some consumers might consider this impractical and complain.

It's up to the consumer to implement the webhook and ensure that only we, the Banking API provider, can access it. But we must decide which security measure must be implemented. As for a regular API, consult security experts to determine necessary security measures, which may include signing messages with a shared secret (which allows the consumer to confirm that the message hasn't been tampered with and legitimately came from our platform), IP server addresses allowlisting, mutual Transport Layer Security (TLS), and OAuth 2.0. Multiple security mechanisms can be combined,

and the API provider may offer various options to facilitate implementation on the consumer's side. Depending on the chosen security mechanisms and how we trust the webhook implementers, we may decide whether including sensitive data in our notifications is acceptable.

14.6.7 Defining the expected webhook behavior

We only need to confirm that our consumers' servers successfully received the webhook calls without further information. A webhook can respond with any 2xx status, ideally in a few milliseconds. There's no need to return data or a `Location` header; although this could be considered a creation, we, the API provider, don't care what happens to the event and won't need to read the possibly created resource later. In our transaction notification webhook case, we'll define a 2xx response with no data in the OpenAPI document (see section 14.6.9). For example, it allows an implementation to return 200 OK or 204 No Content when successfully receiving an event. If extra data is sent, we won't read it.

Implementing a webhook

A webhook implementation must be as simple as possible to respond quickly and avoid failures. It can check security and the data format before putting the event in a database or event queue, but it must not process the event. This must be done afterward by another system.

14.6.8 Dealing with webhook failures

What happens when the API provider can't send events due to failures on the webhook implemented by a consumer? This can have major implications for the consumers and provider sides, depending on how the provider reacts and the features put in place to fix failures. We have the following options:

- Let consumers fill the gap with regular operations.
- Provide specific operations to help consumers fill the gap.
- Resend past events on demand via the developer portal or API.
- Retry automatically on timeout or errors.

Suppose the transaction notification webhook implemented by a consumer takes too long to respond to our call or responds with a status other than 2xx. We can let the consumer get the missed transaction by calling the regular operations of the Banking API (such as the `GET /transactions` operation we defined in section 14.6.2).

However, this can complicate webhook implementation by requiring specific API calls for different events like new transactions ("List transactions") or account balance updates ("Read account"). To mitigate this, we could provide a "List notifications" operation to retrieve past notifications. Consumers could get the undelivered notifications using appropriate search filters

To simplify the implementation further, we could propose resending past events on demand via the API developer portal or an API call to a “Redeliver notifications” operation. That way, the events would go through the same channel on the webhook implementation.

To prevent the need to catch up on missed events, we can automatically retry failed event deliveries due to timeouts or errors. Webhook implementers could include a `Retry` header in the response to optimize our retry. However, we don’t want to use excessive resources; therefore, we’ll limit retries to five over 24 hours. Consumers can use on-demand redelivery after this limit.



NOTE Have a discussion with stakeholders, especially architects or tech leads, to evaluate the need for and complexity of setting up an event history and automated or on-demand retries. For comparison, look at how popular APIs like GitHub and PayPal handle webhooks. Additionally, their documentation offers valuable insights into webhook behavior and implementation.

14.6.9 Describing a webhook with OpenAPI

Figure 14.10 illustrates that describing a webhook within an OpenAPI document is similar to describing a regular operation. Regular operations are defined under `paths`,

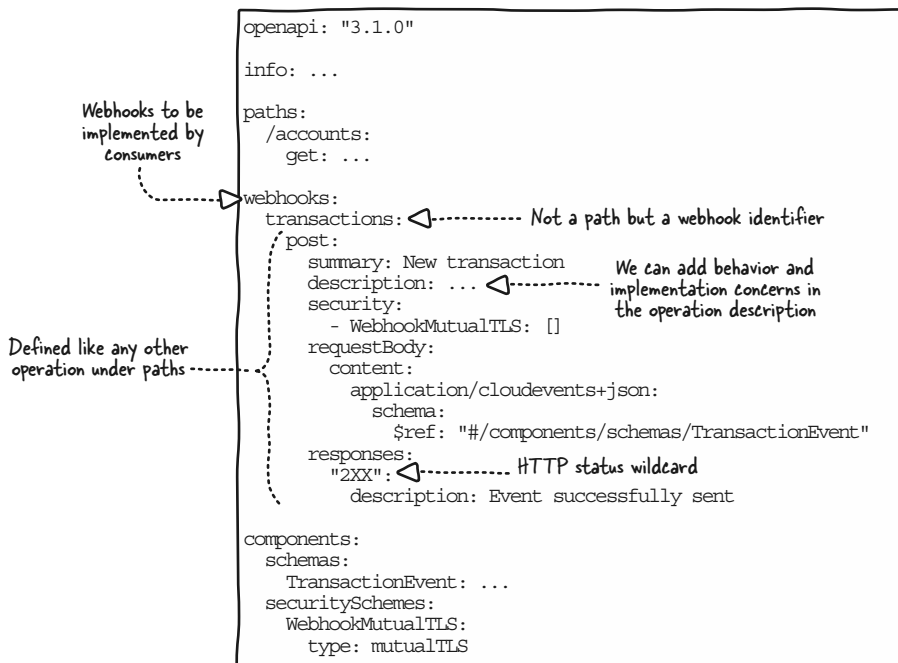


Figure 14.10 Webhooks go under the `webhooks` property of the OpenAPI document. A webhook doesn’t have a path but an identifier. Other than that, a webhook is defined like any other operation under `paths`.

but webhooks are defined under `webhooks`. The keys under the `webhooks` property are not paths but webhook identifiers, such as `TransactionNotification`. Consumers are free to use any path for their implementation. These are the only differences; the rest of the webhook is defined like any other operation. As usual, we added the `POST HTTP` method, `summary`, `requestBody`, and `responses` and referenced a model defined under `components.schemas`. It's worth noting that OpenAPI supports HTTP wildcards; we can define a `2XX` response to signify that we accept any successful response. Because we're using a security mode handled by OpenAPI (mutual TLS), we defined it under `components.securitySchemes` and indicated it in the `security` property of the webhook operation. Other security measures may be specified using a formal (custom header holding the message signature, for example) or textual description.

We can use the webhook `description` to indicate that it has to be implemented by the API consumers and describe how they provide us with its URL and security configuration. It is also recommended to indicate the behavior of the caller ("Events are resent up to 5 times over 24 hours in case of failure," for example) and the expected behavior of the implementation ("must respond in less than 200 ms," for example).

14.7 Handling long operations

Some operations are qualified as "long" because they require more than a few seconds to return a response; they may last minutes or hours. For instance, detecting kittens in a 2-hour video may take more than 1 minute. Although consumers can technically wait a long time, it should be avoided. We may face scalability problems (connection limits, infrastructure strain). The connection may be interrupted because of a timeout or consumer problem, in which case the consumer would need to restart the process, which wastes server resources. In this section, we discuss options we can use when we need to integrate long operations into our API design:

- Starting a long operation and monitoring its status with polling
- Using a provider-initiated callback to prevent polling (similar to a webhook)
- Letting consumers choose an execution mode with the `Prefer` header

14.7.1 Starting a long operation and monitoring its status with polling

When dealing with long operations, it's common to start with a `create` API call to get a reference to the running operation (`POST /resources`) and check its status with subsequent read calls (`GET /resources/{resourceId}`). Although we saw that such a polling strategy can be inefficient with random events (section 14.6), it can be optimized for long operations for which we can determine how they progress and their duration.

Our bank needs to regularly run stock market stress tests to gauge resilience to economic downturns, market volatility, and financial shocks. We want to execute these simulations on demand using a dashboard that makes API calls. A simulation can last up to an hour, so we can't call a `POST /simulations` and wait for the end of the simulation to respond with `201 Created`. Instead, we can return `202 Accepted` along with

the `simulation` status and `Location` header set to `/simulations/12345`. Afterward, the consumer can poll on `GET /simulations/12345` to check whether the status is `IN_PROGRESS` or `DONE`.

We can enhance the response with information about the processing to optimize polling on the `GET` request. We can indicate the progress as a percentage (`"progress-Percentage": 31.6`), the progression rate (`"progressPercentagePerMinute": 1.8`), and the estimated end time (`"estimatedEnd": "2024-06-21T10:34:12"`). With this data, the consumer can adjust the call frequency or wait for the estimated end before making another call. We may also use the `Retry-After` HTTP header discovered with the `503 Service Unavailable` in section 14.2.3. Although using it on successful `GET` requests is uncommon, returning it set with the estimated end date on a `GET /simulations/12345` call can make sense.

14.7.2 *Using a callback API to avoid polling*

Instead of letting consumers poll, we can use a callback to notify them that the long operation they initiated has ended. A callback is like the webhook in section 14.6: a reverse API defined by the API provider and implemented by the API consumer. Sometimes, the terms *webhook* and *callback* are used interchangeably. In this book, we differentiate them by saying that a callback call is the result of a consumer-initiated action (the end of a stress simulation launched by the consumer, for example), and a webhook call is generated by a provider-initiated event (a new transaction caused by a card payment, for example). Technically, it results in the API provider calling the API consumer. It's worth noting that webhook and callback calls can be received by the same implementation or different ones on the consumer side. Designing a callback is no different than designing a webhook, so we can reuse what we learned in section 14.6.6; we won't go into all the details here.

We treat webhooks and callbacks similarly for simplicity of implementation on the consumer and provider sides. Both use the same mutual TLS security configuration and URL defined via the API developer portal. To be consistent with the transaction notification events, we use the `CloudEvents` format for the "Simulation end" callback. The event type is `com.banking.simulation.end`, and its data contains a reference to the ended simulation. We use a lightweight event because the simulation results represent a huge amount of data, which is accessible via various operations on the simulation subresources.

14.7.3 *Describing a callback with OpenAPI*

To describe a callback with OpenAPI, we can describe it as a webhook under webhooks, like the `TransactionNotification` webhook (see section 14.6.9). However, although we could use the `SimulationEnd` webhook and `POST /simulations` description to mention their relationship, it's not formally explicit. Instead, we can define a callback operation under `callbacks` in the `POST /simulations` operation, as shown in figure 14.11.

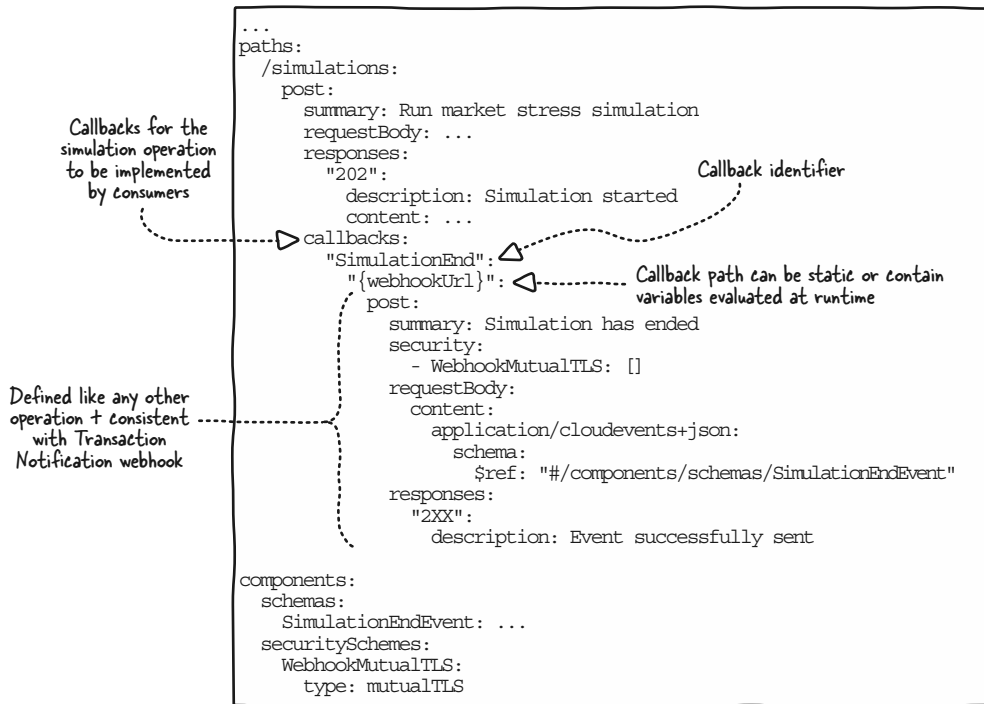


Figure 14.11 An operation callback is defined under `callbacks` within the operation object. Its path can be static or contain runtime variables. We use a design consistent with the previously defined transaction notification webhook.

Multiple callbacks can be defined under `callbacks`. Each callback has an identifier, such as `SimulationEnd`. We could, for example, add a `SimulationFail` callback. Unlike webhooks, OpenAPI callbacks have URLs, which can be static (`/my-callback`) or dynamic (`/my-callback/{withVariable}`). A dynamic callback URL can use the request or response data. If we allowed the consumer to define the callback URL via a query parameter (`POST /simulations?callback=https://consumer.com/callback`), the callback path would be `{request.query.callback}`. However, we prefer to have a shared URL for all our webhooks and callbacks, defined via the API developer portal, so we use a custom variable, `{webhookUrl}`. When the simulation ends, the system responsible for calling the callback will fetch the URL from the consumer configuration.



TIP If your editor reports an OpenAPI syntax error, ensure that you have defined callbacks at the root of the operation (same level as `responses`) and that your callback has an identifier and then a URL (it's common to forget one or the other).

The rest of the callback description is similar to any operation and is especially consistent with the `TransactionNotification` webhook. We use the same `WebhookMutualTLS`

security scheme defined under `components.securitySchemes`. The `application/cloudevents+json` request body references the `SimulationEndEvent` JSON Schema model under `components.schemas`.

14.7.4 *Choosing an execution mode with the Prefer header*

We can propose letting consumers choose whether to execute a long operation in asynchronous or synchronous mode. Even if it's not the best option and should be used with caution, it can be helpful to simplify implementation on the consumer side when they have minimal capabilities.

A `POST /simulations` request with a `Prefer` header set to `response=synchronous` would wait for the end of the simulation before responding with `201 Created` and a `DONE` status. Not providing the `Prefer` header or setting it to `response=asynchronous` would result in an immediate `202 Accepted` response with an `IN_PROGRESS` status as in section 14.7.2.

But this synchronous execution is not super-optimized due to its very long duration. We may reserve it only for selected consumers, and we can use behavioral scope to do so. Only consumers with the `simulation:run:synchronous` scope can benefit from this option if they provide the appropriate `Prefer` header.

14.8 *Considering other API types*

Although we usually rely on a predefined development stack, including an API type (REST in the context of this book), we must stay open to other solutions, because our usual API type may not be the best option. “If you have a hammer, everything looks like a nail” is a common way to summarize the “Law of the Instrument,” a cognitive bias involving overreliance on a familiar tool or methodology. Like any other aspect of the solution our team builds, we must choose an API type according to the user needs and context. As API designers, knowing about alternatives will help us detect possible problems and alert architects and tech leads. This section introduces typical alternatives and discusses when to select an API type.

14.8.1 *Introducing REST API alternatives*

This section briefly discusses a few cases to illustrate alternatives to REST APIs:

- Event-driven architecture (EDA)
- HTTP Server-Sent Events (SSE)
- WebSockets
- GraphQL
- gRPC

In section 14.6, we used webhooks to notify consumers about events happening independently, such as new transactions. Although webhooks make sense when communicating over the internet, setting up an event-driven architecture (EDA) makes sense when communicating internally. In this architecture, an event provider, such as the

transaction system, can publish events in message channels, and event consumers can subscribe to these channels to receive the events, such as notifications for new transactions. Note that our transaction notification webhook could be powered by such an architecture.

We used a callback to notify consumers about the end of a simulation in section 14.7.2. We could use HTTP Server-Sent Events (SSE) to provide real-time information about the simulation's progress. SSE allows textual data to be streamed from the server to the client over HTTP. SSE could also be useful when feeding a web application with real-time stock market data or streaming an AI agent's response. It's common to use SSE in conjunction with a REST API because you just need a regular HTTP call to receive the server-sent events. Note that this means keeping HTTP connections open, which can be resource-intensive. If account owners need to chat with their bank advisors, we need real-time bidirectional communication, which can be achieved with WebSockets. This protocol allows the bidirectional exchange of textual and binary data. Note that as with long-lived HTTP connections, handling large numbers of WebSockets connections can be resource-intensive. Also, firewalls or proxies may block WebSockets connections, requiring additional configuration.

Section 13.8 discussed creating a backend-for-frontend for heavy optimizations; we may consider using a GraphQL API in such a case. GraphQL can aggregate data from various APIs, allows precise selection of elements to retrieve, and features an account application mechanism that uses WebSockets to receive server events, such as data updates. It's important to consider the challenges of using GraphQL, such as caching, preventing resource-intensive complex queries, and ensuring security. If you don't need GraphQL-specific features, a well-designed REST API over HTTP/2 may be enough.

gRPC may be considered in a microservices architecture where many components communicate. It fully uses HTTP/2's possibilities, including bidirectional streaming. Messages use the Protobuf binary format, which is more compact and has more efficient serialization than JSON (even over HTTP/2). Be aware that browsers do not support gRPC.

14.8.2 When to select an API type

In most cases, we use a standard development stack that includes an API type (which may vary depending on the component type and location within the architecture). That means we usually already have a predefined API type when entering the design stage. However, the needs analysis may reveal requirements incompatible with our usual choice. In the context of this book, our standard choice is a REST API, and we may consider other options if

- We need to stream data to consumers. (consider SSE)
- We need bidirectional communication. (consider WebSockets)
- We need to send events to internal systems. (consider EDA)
- We build a backend-for-frontend. (consider GraphQL if you need its specific features that a well-designed REST API exposed over HTTP/2 can't provide)

Note that if your standard choice is not REST, you should consider it if

- You design a public or partner API. (the vast majority use REST, although there are a few public GraphQL APIs)
- You design a private API that may become a partner or public. (or you need to train before providing public or partner APIs)
- Browsers consume the API. (gRPC is not browser-compatible, for example)
- You need provider-to-consumer communication over the internet. (HTTP web-hook or callback)

Summary

- Adapt the design only after challenging constraints to ensure that they're not solvable.
- Use the X-HTTP-Method-Override request header to allow consumers to call an operation using a method they don't support.
- Use content negotiation to propose different formats if they don't constrain the operation flow; otherwise, consider separate APIs or support only one format.
- Return 503 Service Unavailable and a Retry-After header, or store requests to process them later, responding with 202 Accepted in the case of planned unavailability.
- Ensure that data, especially IDs, is URL compatible and that URLs don't grow over 2,000 characters.
- When uploading files in a flow, use generic files to attach to a business resource (POST /files) or files directly attached to a business resource (POST /resources/{resourceId}/documents).
- Mix files and data using Base64 file content encoding or a multipart content type, but be careful about efficiency.
- To describe a file in a body in OpenAPI, indicate the relevant media type(s) under content without any other information (empty object).
- To describe a Base64-encoded file in a JSON Schema, use JSON Schema's `contentMediaType` and `contentEncoding`.
- Use an OpenAPI encoding object to override a part media type with `multipart/form-data`.
- Enable caching, and don't return files on creation or upload for efficiency.
- To efficiently handle large files, enable partial download (`Range` header) and upload (`Content-Range` header) and upload prechecks (`Expect: 100-continue` header and `100 Continue` status). Ideally, rely on a file management system.
- Redirect consumers with 303 See Other and a secured URL to download a file from a third-party system. Return a body with the secured URL, HTTP method, and headers for file uploads.
- Define a consumer-implemented webhook to notify them about provider-generated events and prevent inefficient polling.

- Not all consumers can implement webhooks; ensure that consumers can retrieve data with means other than webhooks if feasible.
- Use the CloudEvents standard to design interoperable events.
- Design lightweight events pointing to a resource for volatile or large data, and reserve heavyweight events for small and static data.
- Optionally propose resending webhook events or retrieving past events in case of failure.
- Define webhooks under the `webhooks` property of the API's OpenAPI document.
- Provide progress information or a `Retry-After` header to optimize polling on long operations.
- Propose a callback to notify the consumer of the end of a long operation.
- Define long operation's callback under `callbacks` in the API's OpenAPI document.
- Consider API types other than REST when there's a need for data streaming, bidirectional communication, sending events to internal systems, and highly specific backend-for-frontend needs.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 14.1

The Wood as a Service company proposes made-to-measure planks in different types of wood. It wants to create an API to automate ordering. However, human validation is required, limiting ordering to business hours. How can you adapt the design to this constraint?

Exercise 14.2

The Very Fast Shipping company offers an API providing information about shipments but faces infrastructure overload due to consumers repeatedly calling `GET /shipments` and `GET /shipments/{shipmentId}` operations. What can they do to fix this?

Exercise 14.3

The Music Analyzer API processes audio and video files to detect tempo, key, scale, chord progressions, instruments, and voices and to generate audio files containing selected instruments and voices with modified tempo or scale to create backing tracks or for karaoke. Users may proceed in one or several passes to extract the needed data from a file. The API comprises a single operation expecting a Base64-encoded file and the list of processing tasks to perform (listing 14.5); each processing task may take

between 1 second and 1 minute per minute of audio content. The response (listing 14.6) returns data for each processing. Processing data varies depending on the type and may include Base64-encoded files. What are the problems with this design, and how could it be improved?

Listing 14.5 Processing request

```
POST /processing-tasks
```

```
{
  "file": "QmFzZTY0RW5jb2RlZFN0cmIuZw==...",
  "processingTasks": [
    { "type": "tempo" },
    ...
    { "type": "karaoke" }
  ]
}
```

Listing 14.6 Processing response

```
200 OK
```

```
{
  "processingTasks": [
    { "type": "tempo", "data": { tempo: 180 } },
    ...
    { "type": "karaoke", "data": { "file": "U2ltcGx..." } }
  ]
}
```


15

Modifying an API

This chapter covers

- Designing backward-compatible modifications
- Balancing the value and effects of breaking changes
- Versioning an API
- Creating an extensible design
- Describing modifications with OpenAPI

The Shopping company would like to enhance its API with support for multiple categories. Toward this end, it considers replacing the product's category (string) with a categories array. However, that would require modifying all consuming applications' code to use the categories array instead of the category property. A non-updated application's code will break because the category property it expects to find isn't there. Although updating internal applications is not a problem, asking all partners to modify their applications is more complicated. Now, imagine that such a design modification is made without anyone being informed; all the consuming applications would suddenly stop functioning. That could cost a lot of money and affect the Shopping company's reputation.

An API rarely stays unmodified once consumed; consumers can provide feedback, and new user needs can arise. When modifying an API, we must still fulfill

user needs; be user-friendly, interoperable, secure, and efficient; and adapt to context. However, modifications may cause problems such as breaking consumer code, user interface errors, or data corruption. Careful design of modifications is crucial to avoid or identify such breaking or non-backward-compatible changes before deploying them in production. Identifying breaking changes allows us to do what is necessary to enable consumers to migrate smoothly to the new API version if those changes can't be avoided. Additionally, the risk of introducing breaking changes on modification may increase due to the initial design. However, careful initial API design can help reduce this risk and facilitate evolution.

This chapter examines the specific concerns related to API modifications and then discusses designing backward-compatible changes, API versioning, and balancing the value and effects of breaking changes. We also explain how to create an extensible design. Finally, we cover documenting modifications and informing consumers about them.

15.1 *An overview of API modification concerns*

As shown in figure 15.1, we are exploring the last concern of the constraints layer of the API design process. Modifying an API design is similar to designing an API from scratch. We must identify capabilities to meet user needs and design the matching programming interface. We must ensure that our design is user-friendly, interoperable, secure, and efficient and integrates contextual constraints. But there are additional concerns specific to modifying an API that we must consider, such as introducing non-backward-compatible changes that break consumers and API versioning. This section examines what can happen when modifying an API and how to design API modifications.

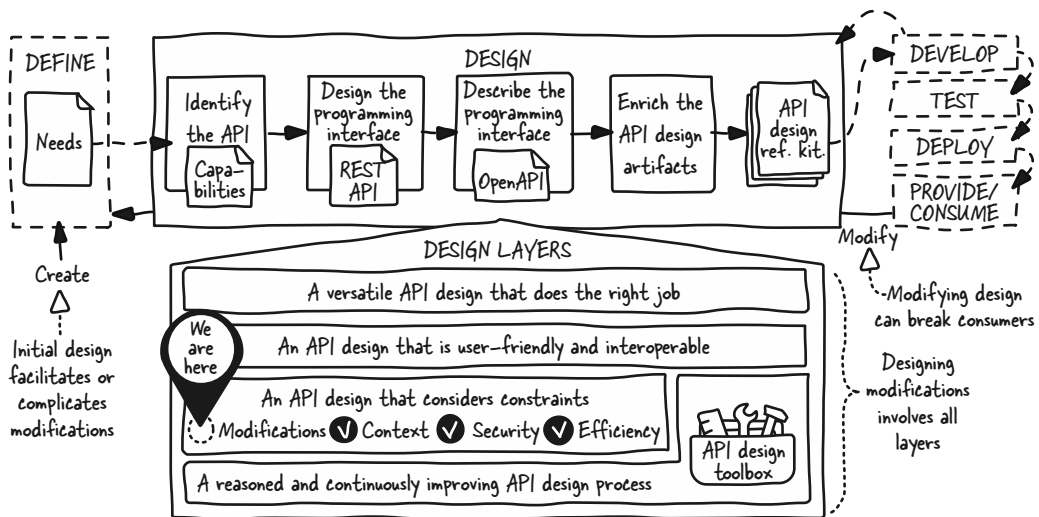


Figure 15.1 Modifying a consumed API is “design as usual,” with the constraint of determining how modifications will affect consumers and making trade-offs if necessary. The initial design can facilitate later modifications.

15.1.1 What can happen when modifying an API?

Suppose we want to add support for multiple currencies to the Shopping API. We could replace the price number property with a value and a currency object. This would require modifying all operations that use prices in input or output data. If we carelessly deployed this change in production, consumers would err when reading products because they expected prices to be numbers. They would also face an error when adding products because the server expects a price object, not a number. Both errors would be visible to end users.

Fortunately, we are well aware of the potential problems this modification could cause and won't deploy such a breaking change without thorough preparation. If only one internal consuming application existed, we could easily synchronize the deployment of its update with the new API. However, synchronized deployment isn't feasible if numerous partners use the API. To address this, we can expose both the previous and new versions of the Shopping API in parallel, allowing consumers to adjust their code at their own pace for a smooth transition.

Another option is to revise our design modification to ensure backward compatibility. Grouping the price's value and currency in an object is a nice design but introduces a breaking change. Instead, we can leave the price property unchanged and add a `priceCurrency` with a default value. It won't affect unmodified consumers, and we won't have to maintain two versions of the Shopping API. A better initial design could have avoided this trade-off. It's frustrating, but we'll learn to live with it; we'll probably discover more problems as more people use our API in various situations.



CAUTION Modifying an API must be done carefully, regardless of its visibility (public, partner, private), “size” (from microservices to huge monoliths), or type (REST or other).

15.1.2 Uncovering API design modification concerns

We need to consider the following when modifying APIs:

- Creating an extensible initial design
- Listing the modifications made to the design
- Determining whether modifications are backward compatible
- Balancing the effects and value of breaking changes
- Versioning our API

Designing API modifications starts before they happen. We must make our initial design extensible to prepare the ground for future modifications. Creating a versatile and flexible design that's usable in various contexts is an excellent start. However, we must avoid specific design patterns to limit the risks of introducing non-backward-compatible changes later.

We must exhaustively list modifications to ensure that they meet user needs and to evaluate their effect. The developers of the implementation and the developers of consuming applications will also need this information to update their code if necessary.

We must determine whether the modifications are backward compatible to ensure that we won't inadvertently break consumers or API security. However, introducing a breaking change is not always a problem. Its effect depends on who consumes the API and how, our versioning capabilities, and the value it adds to the API. If a breaking change is not worth the cost or is impossible, we can make a design trade-off to avoid it.

API versioning involves more than picking a version name or number; it requires defining a versioning policy that outlines when and how modifications occur and how to address breaking changes (which we may choose to avoid) and specifies the number of supported past version and their lifespan. API versioning also considers implementation and architecture: can we run multiple API versions concurrently, and how does this affect the policy? API product owners and architects manage these decisions, but versioning may constrain our design work.

15.1.3 How to design API modifications

Figure 15.2 shows how to integrate API modification concerns into our design process. API product owners and architects define versioning policy and related architecture. Because it affects design, we (or another designer) contribute to defining the versioning scheme (what we version and how we identify a version). Then we design the initial version of the API with extensibility in mind.

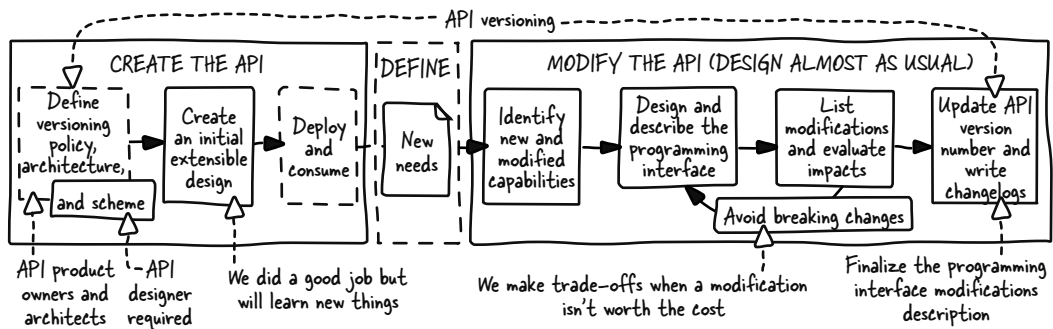


Figure 15.2 Handling design modification starts when we create an API. Modifying an API is “design as usual” with the addition of taking care of breaking changes and modification logs.

Once the API is consumed, modifications may be necessary to meet new user needs or fix problems due to inaccurate needs identification during the Define stage of the API lifecycle or a faulty design (which should not happen due to our excellent API design skills). We identify new and modified capabilities and design and describe the matching programming interface as usual. However, we must exhaustively list modifications to evaluate their effects and make design trade-offs when necessary (similar to when we integrated unsolvable contextual constraints in the design in section 14.1.4). Once

we're all set, we can finalize the API description by updating the version number and adding a change log.

The rest of this chapter dives into these concerns by discussing the following:

- Identifying breaking changes and ensuring backward compatibility
- Identifying security-related breaking changes and preventing breaches
- Assigning a version to an API
- Carefully breaking and versioning an API
- Creating an extensible API design
- Describing the design modifications with OpenAPI

15.2 Identifying breaking changes and ensuring backward compatibility

A breaking or non-backward-compatible change requires consumers to modify their code to continue using the API. Such modifications can lead to blatant errors, especially when end users are involved. They can also cause silent problems because the unmodified code can still process modified data but produce results that are different than expected. This section contrasts breaking and backward-compatible changes when modifying

- Success and error output
- Input data
- Resource paths
- Operations (HTTP methods)
- HTTP statuses
- Operation flows

We also discuss specific breaking changes that may occur because consumers rely on an API's non-explicit behaviors and how to identify and prevent unintended modifications.

15.2.1 Modifying output data

Consumer code may break if it can't find expected data or receives unexpected values in a modified response. To ensure backward-compatible changes, we must only add or adjust data within the original limits. This applies to success and error response headers and bodies. Figure 15.3 outlines breaking and compatible changes, and figure 15.4 shows these changes applied to an output Transaction JSON Schema model.

API consumers may crash or behave unexpectedly if the data they rely on is missing from the operation response. For example, making the always-included `category` property optional could result in crashes or in "null" being displayed in a UI. Renaming `amt` to `amount`, eliminating the required `aboveAverageAmount`, or nesting `merchantName` inside a `merchant` object leads to similar problems because consumers won't find what they expect.

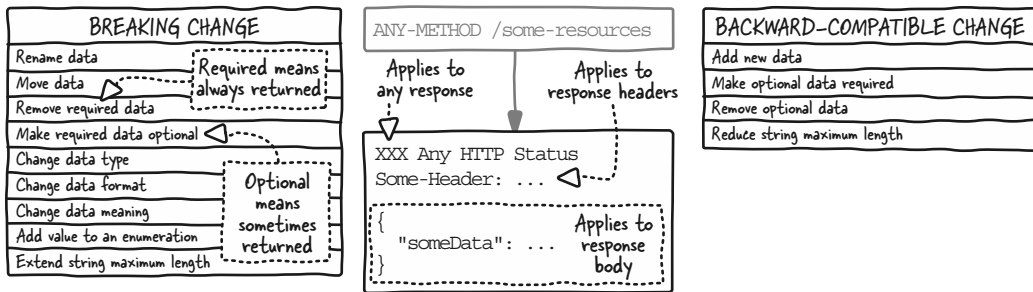


Figure 15.3 Breaking changes can occur in headers and bodies of any success or error response. Only add or modify data within the limits of the original data to be backward compatible.

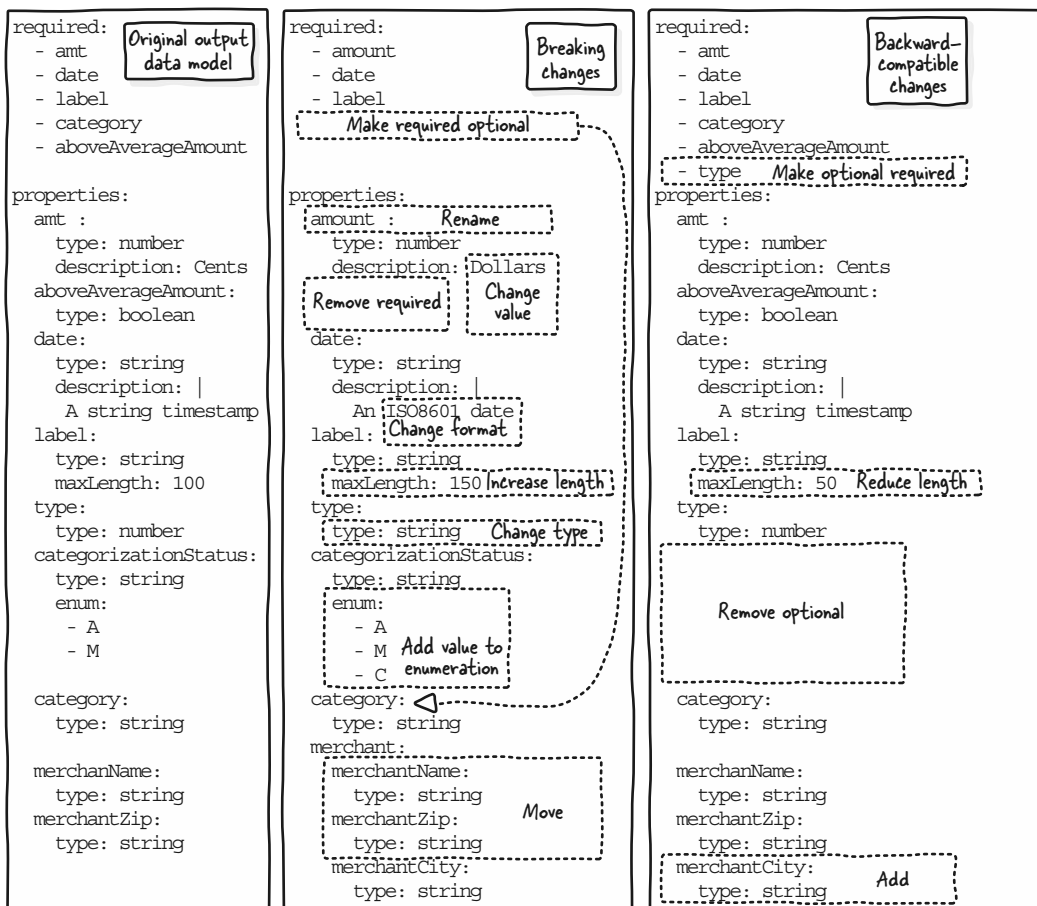


Figure 15.4 Breaking changes in output data require updating the consuming applications' code. Backward-compatible changes have no effect on non-updated consumers, but they won't use the newly added data.

API consumers may face problems with unexpected values or types. Changing the type property from a number to a string can crash JSON parsing. Altering the date property format from a number (timestamp) to a string (ISO8601) will also cause a crash. Adding `M` to the `categorizationStatus` enumeration will cause a problem because it is likely mapped to something more meaningful. Increasing the `label` length from 100 to 150 may lead to errors in a relational database on the consumer side if the column is set to 100.

Adding required data like `merchantCity` won't cause problems; non-updated API consumers won't use it. Always returning previously optional data is fine; adding type to the required list is acceptable. We can remove optional data like `categorizationStatus` because consumers expect that it may not be returned. Shortening string lengths, like `label`, is also fine.

Introducing breaking changes affects all responses, including errors, as shown in figure 15.5. For instance, renaming `items` to `errors` may cause problems, potentially crashing consumers or hiding errors due to the missing `items` property. Retaining `items` while replacing specific error types (`MISSING_SOURCE`, `MISSING_DESTINATION`) with a generic type (`REQUIRED`) can lead to crashes or ignored errors.

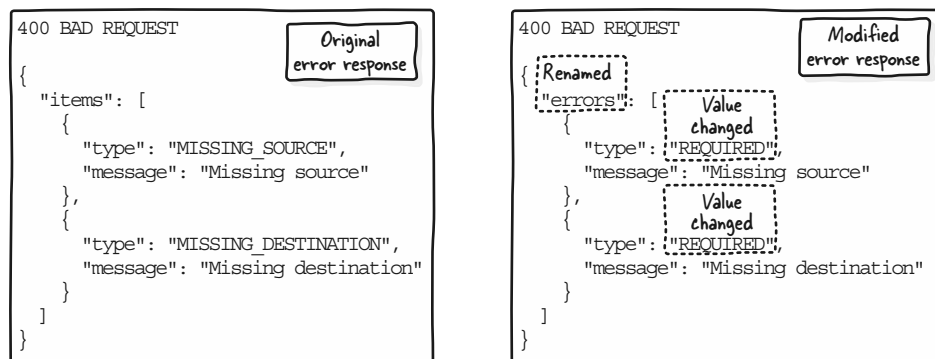


Figure 15.5 Breaking changes are introduced while enhancing an error data model. The error list is renamed, and specific error types are replaced by generic ones.

All these concerns apply to response headers, too. For example, if we always return a `Location` header on `POST` requests, no longer returning it could cause consumers to crash.

15.2.2 Modifying input data

Unmodified consumers will send requests with inputs defined by the previous version. If the server can't find the required data or receives unexpected values, it will return errors or unexpected, more, or less data. More concerning is that incorrect data may be accepted, causing serious side effects. To ensure backward-compatible

input modifications, we can make required data optional, add new optional data, and expand existing data limits. This applies to body, headers, query parameters, and paths (see section 15.2.3). Figure 15.6 lists input data changes, and figure 15.7 illustrates them in the Transfer JSON Schema data model of the POST /transfers operation.

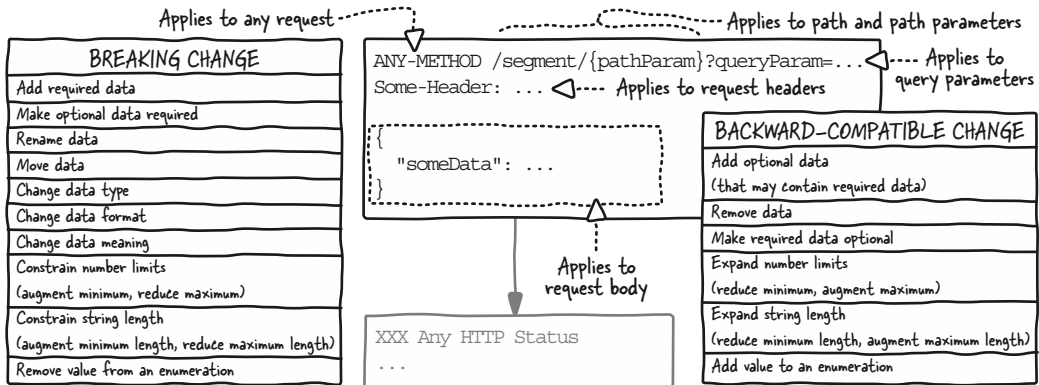


Figure 15.6 Breaking changes can occur in request bodies, headers, query parameters, and paths (including path parameters). To be backward compatible, only make required data optional, add new optional data, and expand the limits of existing data.

The server returns an error if the required data for the new version is missing. Consumers will send `amt` instead of `amount` and `destinationAccount` instead of the `destination` object. They will not send new required elements like `reason`.

The server returns an error if required or optional data has unexpected values or types. Consumers sending money transfers in `EUR` will receive an error because it has been removed from the currency enumeration. They will send the date as a string timestamp instead of an ISO 8601 date and an account internal ID (number) instead of an IBAN (string) for `source`. The description length limits have changed from 10–300 to 20–150. Although the minimum amount value has no effect, its maximum has been reduced from 1,000,000 to 9,000.

Modifying the meaning of an input value can have dire consequences. For example, the server may receive cents from non-updated consumers but interpret them as dollars. Thus, a consumer sending a 1,000-cent transfer will trigger a 1,000-dollar transfer.

Adding optional data containing required data is backward compatible. The new `reason` object is optional but must include a `code` if sent. Updated consumers will receive an error if they send it without a `code`. However, non-updated consumers won't send a `reason`, so they don't risk causing an error.

We can remove data or make required data optional. The `date` was previously required but was only necessary for delayed transfers, so it's now optional. The

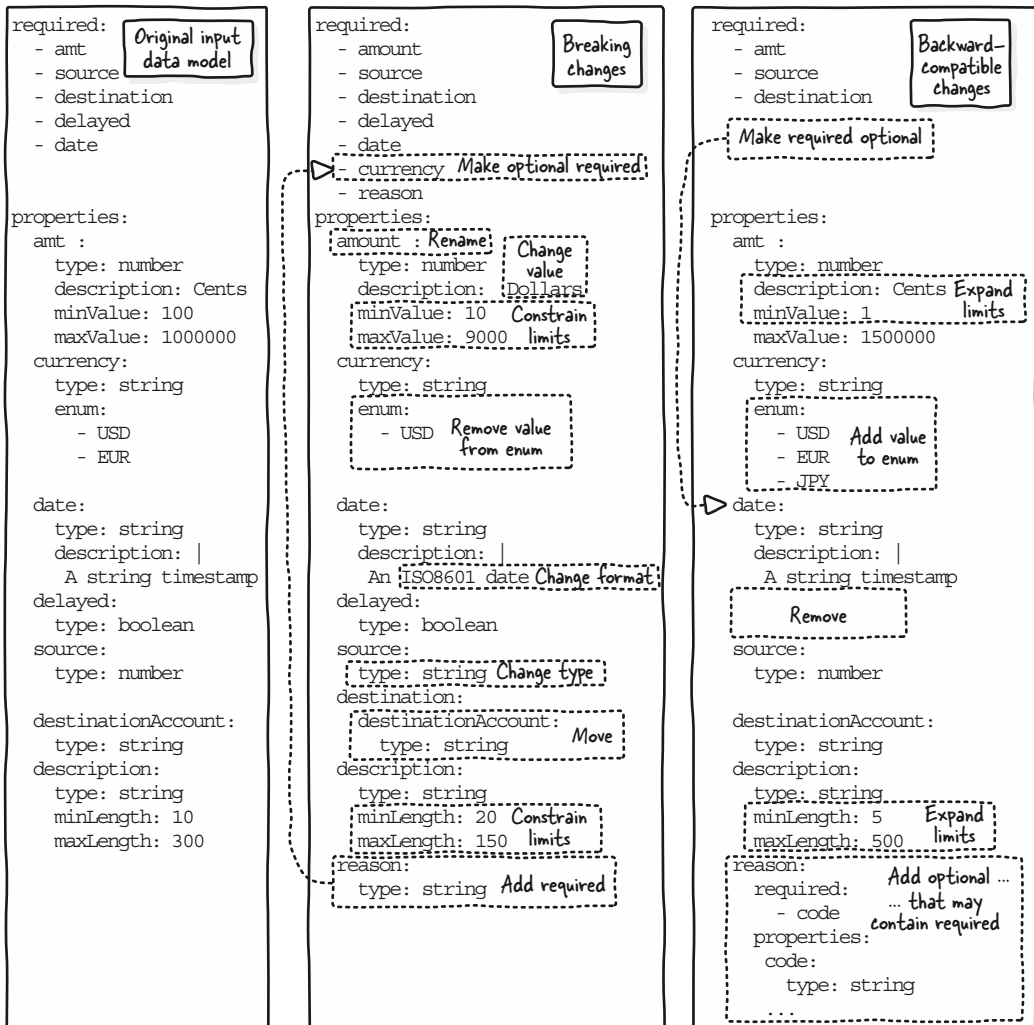


Figure 15.7 The breaking changes in the data model require updating the consumer code. Backward-compatible changes have no effect on consumers. However, non-updated applications will continue to send removed data and will never send newly added data.

delayed flag, which was redundant with the date, has been removed. Consumers can still send these properties; the server will ignore unnecessary data (section 9.8.1).

Expanding ranges and enumerations is backward compatible for inputs. Adding JPY to the currency enumeration will not cause problems; non-updated consumers will not use it. Non-updated consumers will send amounts within the new 1–15,000,000 range because they're used to the smaller 100–10,000,000 range. The same is true for the description string length.

This applies to all input data locations in a request. For instance, renaming the `pageIndex` parameter to `page` means consumers will always receive the first page of transactions instead of the expected page when calling `GET /transactions?page-Index=2`. Adding pagination if it wasn't enabled or reducing the page size will lead to consumers getting fewer transactions than expected, but they may not notice. Additionally, enabling conditional requests and adding a required `If-Match` header on all `PUT` and `PATCH` requests prevents unmodified consumers from making any modifications.



CAUTION Modifying input data often implies modifying output data, so ensure that no breaking changes are introduced there (see section 15.2.1). Input data modification may also affect the implementation's internal behavior; ensure that it doesn't break calls to other operations, hence breaking operation flows (see section 15.2.6).

15.2.3 *Modifying resource paths*

Modifying resource paths follows similar rules as other input data. Non-backward-compatible path modifications will lead to 404 Not Found errors.

Modifying static segments of a resource path or changing its organization will break consumers. For example, changing the `/transaction` resource path to `/transactions` will lead to consumers getting a 404 Not Found error on all of the resource's operations, such as `GET /transaction`. Similarly, changing `/accounts/{accountId}/transactions/{transactionId}` to `/transactions/{transactionId}` also leads to 404 errors. We could implement a 308 Permanent Redirect redirection to mitigate path modification problems, returning a `Location` header targeting the new resource paths (`/transactions` or `/transactions/123`). Unfortunately, consumers are often configured not to follow redirections.

Replacing the value of a path parameter will break consumers, but adding new options is backward compatible. If we replace `/accounts/{accountNumber}` with `/accounts/{iban}`, all consumers calling `GET /accounts/{accountNumber}` will get a 404 Not Found response. But if we accept both options (`/accounts/{accountNumber-OrIban}`), we're backward compatible. Although we modified the path parameter's name, there will be no problem as it's not part of the request; it's replaced by a value (`/accounts/123`).

Suppressing or modifying options for enumerated path parameters (magic identifiers) will break consumers; adding options causes no problems. If we accepted `home` and `office` as address types in `GET /addresses/{type}`, renaming the `office` type to `professional` or removing it will lead to 404 Not Found. Adding a `vacation` address type is OK.

15.2.4 *Modifying operations or their HTTP methods*

Removing an operation or replacing its HTTP method will cause errors for non-updated consumers that will continue to call it. The HTTP status returned when

calling an operation that doesn't exist (anymore) may vary. If we remove `PATCH /addresses/{type}` but still have `GET /addresses/{type}`, the server will respond with a 405 Method Not Allowed status. If we remove all `/addresses` resource operations, the path no longer exists, so the server returns 404 Not Found. Replacing `PATCH /addresses/{type}` with `PUT /addresses/{type}` will cause 405.

Adding an operation under a new or existing path doesn't cause problems. If we already have `PATCH /addresses/{type}`, adding `PUT /addresses/{type}` and adding a new `/customer-addresses/{type}` path and its `PUT` operation are backward compatible. Non-updated applications won't call the newly added operation.

15.2.5 Modifying HTTP statuses

Modifying HTTP statuses can change an operation's behavior and may be accompanied by modified output data. If consumers' HTTP status checks are too strict, or if we modify output data in a non-backward-compatible way, this can break their code. No longer returning a specific HTTP status is backward compatible.

Replacing an HTTP status with another of the same class code or adding one can cause problems. For example, if `POST /transfers` returns 200 OK, we may want to return 201 Created for more precise feedback without affecting the output data. Although the HTTP documentation states that a client should handle any unknown status based on its class, consumers often have strict expectations. Even though 201 is a success, it won't match an `if(response.status !== 200)` statement. If we modified it to return 202 Accepted in case of unavailability, it could also cause problems. If a consumer uses `if(response.status.isSuccess())` but the 202 is accompanied by less or different data than 201, we fall into the problems from section 15.2.1. We modified the expected output data, likely breaking consumers' code when transfers are unavailable.

This also applies to 4XX and 5XX errors. For example, replacing 400 Bad Request with 422 Unprocessable Content may not be handled as expected due to a strict `if`. However, modifying error HTTP statuses may be OK in some cases. For example, adding 429 Too Many Requests errors to our possible 4XX errors can lead to consumers not handling this new case. It will most likely fall under an `else` statement that treats unexpected errors. If lucky, the code can extract information from our generic error format or not use response data. Otherwise, it breaks due to missing or unexpected data.

Stopping to return a specific HTTP status will not cause any problems. If the money transfer operation returned 503 Service Unavailable because of nightly maintenance, but we found a way to avoid this maintenance, consumers would no longer receive this 503.

15.2.6 Modifying operation flows

Modifying an operation flow will break consumers. For example, the "Transfer money" use case consists of "Search sources," "Search destinations for sources," and

“Transfer.” If, for security reasons, we need to add a new “Validate transfer” step that expects the transfer ID and an OTP (one-time password) sent by SMS, unmodified consumers can no longer achieve this use case. Even worse, because there is no error, consumers won’t know the transfer is not finalized; it’s a silent breaking change.

15.2.7 Being aware of the invisible contract

Consumers may rely on an API’s non-explicit behaviors. Hyrum’s law states, “With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.” Sometimes we may blame consumers for complaining about breaking changes related to non-explicit behaviors, but sometimes it’s our fault.

Suppose a consumer relies on an address position in the `addresses` array instead of the `address` type to get the home address of an account owner. If for some reason the `addresses` array order changes, they’ll get the wrong address. Although an array is an ordered set of elements, we never explicitly state how the addresses are ordered. Too bad for them, but that’s their loss. Searching for an address based on its type is the right approach.

The API gateway handles rate limiting, whose 429 Too Many Requests isn’t explicitly documented in our Banking API documentation because “we don’t manage the API gateway.” But breaking changes introduced in such a response’s data will break consumers’ code. This time, it’s our fault: we needed to document this and take care of its evolution correctly. Consumers of partner and public APIs won’t care if another team inside our organization handles this.

Although not directly design-related, it’s worth noting that consumers may be affected by implementation-related modifications that affect response time. Augmenting response time because of new business logic added to an operation may lead to reaching some consumers’ timeout limits.

15.2.8 Preventing unintended modifications

Unwanted or underestimated modifications can break the API at any time, during design and after. Always compare old and new OpenAPI documents to ensure that all modifications are listed and to evaluate their effect. A regular diff will fall short, possibly indicating many structural changes in the OpenAPI document even though the interface hasn’t been modified. Additionally, not everyone knows what kind of modification could break consumers.

Specialized tools exist to compare OpenAPI documents and indicate the safe and breaking changes. They can easily be integrated into continuous integration and continuous delivery (CI/CD) as another layer of testing, comparing the OpenAPI validated for the last deployment with the new version. If unwanted changes are detected, deployment can be blocked. However, such tools won’t detect the changes outside the OpenAPI document, such as response behavior (section 15.2.5) or the invisible contract (section 15.2.7). Thoroughly testing the implementation against an expected

OpenAPI also helps to prevent unwanted changes. This implies that an OpenAPI document acts as a source of truth (section 6.2).



NOTE The “OpenAPI diff” space is constantly evolving; search for “diff” on <https://tools.openapis.org/> or “oas diff” or “openapi diff” with your favorite search engine. Thoroughly check the tools’ documentation to see how they decide whether a modification is a breaking change.

15.3 Identifying security-breaking changes and preventing breaches

We may need to modify API security scopes and schemes when adding or removing operations. Independent of any other modification, we can replace a less secure scheme with a more secure one or adjust scopes for better access control. Depending on the change, modifying security schemes and scopes can introduce breaking changes and security breaches. Although introducing a security breach is not an option, introducing a breaking change to improve security is likely unavoidable.

Replacing or removing a security scheme will break consumers. For instance, our partners can use an API key security scheme. When calling our Banking API, they send an `API-KEY` header with a static value generated via the developer portal. Our security team considers API keys unsecured, and they strongly recommend replacing this security scheme with mutual TLS authentication when communicating with partners. This is a security protocol with which the consumer and the provider authenticate each other’s digital certificates to establish a secure and trusted communication channel. Such a modification requires changing consumer code. Although that’s a breaking change we can’t avoid, we can temporarily propose the two security modes to let consumers switch.

Modifying the security scope configuration can lead to breaking changes. Suppose `account:all` consists of the “List transfers” operation. If we move it under a new `transfer: read` scope, consumers must update their code to call this operation. That’s because consumers indicate which scopes they’ll use on authentication. To mitigate this, we can add the “List transfers” operations to the `transfer: read` scope and later remove it from `account:all` once consumers have been updated.

Modifying and adding scopes may not break consumers but still create security breaches. Apply what we learned about security scope design in section 12.7 when adding new scopes or modifying existing ones. For example, adding “Delete account” to the `user:account:all` scope grants owner-facing applications access to an admin-only operation.

15.4 Assigning a version to an API

Software versioning consists of assigning a unique version number or name to a specific state of the software. The initial version of the Banking API, which only provides access to bank accounts and their transactions, could be “1.0” or “2023-10-03.”

The following version, in which we added money transfers and related operations, could be “1.1” or “2024-03-17.”

However, versioning also involves determining when and how to attribute a version identifier and handle the effects. In the API world, the question of versioning an API typically comes up when a breaking change is introduced and is often reduced to a debate about whether the API version number should be included in the path or elsewhere. However, API versioning is more than that and should be discussed early.

This section contrasts API and implementation versioning. Then we discuss how to choose an API version identifier and represent it in a request and when to make these decisions. Finally, we discuss why we should avoid version sub-elements of the API. Versioning as a process related to introducing breaking changes and the consequences is discussed in section 15.5.

15.4.1 *Differentiating interface and implementation versioning*

When we version the API we’ve designed, we version what is visible from the consumer perspective: the interface contract, not the implementation code. The implementation code can evolve without affecting the API it exposes; this is not the consumer’s business. Figure 15.8 contrasts the evolution of the Banking API and its implementation versions using incremental version numbers.

WHAT CONSUMERS SEE	API IMPL.	MODIFICATION DESCRIPTION
Account Information	v1 v1	Initial version with account information operations
Account Information Unmodified API	v1 v2	Fix account access bug
Account Information, Transfer	v2 v3	Add transfer operations
Account Information, Transfer Unmodified API	v2 v4	Implement transfer subsystem checks and send to queue
Account Information, Transfer, Acc. Application	v3 v5	Add account application operations

Figure 15.8 We version the API from the consumer perspective. Implementation modifications that don’t affect the interface contract are not reflected in API version numbering.

In the initial v1 version, the API (and implementation) provided operations related to account information. However, we discovered a bug that allowed unrestricted access when reading an account, so we fixed this in version v2 of the implementation without affecting the API, which stayed in v1. Later, we added transfer-related operations, resulting in version v2 of the API and v3 of the implementation. We initially had an inefficient implementation of the “Transfer” operation due to a direct connection with the transfer subsystem. We improved it by reimplementing the transfer subsystem

checks and using a message queue. It was a heavy restructuring, but it was transparent for consumers, resulting in version v4 of the implementation and an unmodified v2 for the API. Finally, we added account application-related operations, leading to version v3 of the API and v5 of the implementation.

15.4.2 Choosing an API version identifier

Semantic versioning is the most commonly used versioning scheme in APIs, but you may also come across date-based versioning. Figure 15.9 contrasts both options when modifying the Banking API.

BANKING API VERSION DATE SEMANTIC	CHANGE TYPE	CHANGE DESCRIPTION
2023-10-02 1.0		Initial version with List accounts and Read account operations
2023-11-06 1.1	Backward-compatible	Added List transactions operation
2024-01-08 1.2	Backward-compatible	Added merchant address and name to List transactions output
2024-02-05 2.0	Breaking	Replaced account number (/accounts/{accountNumber}) by IBAN (/accounts/{IBAN}) for Read account and List transactions
2024-03-04 3.0	Breaking	Removed cards list from Read account

Figure 15.9 Semantic versioning indicates the type of change and date-based versioning doesn't.

Semantic versioning (<https://semver.org/>) is widely used in software. It consists of three MAJOR.MINOR.PATCH digits or versions, such as 1.0.0 or 1.2.1. The MAJOR version is incremented for breaking API changes and MINOR for backward-compatible modifications. We don't use the PATCH version for API versioning, because it's for bug fixes that can only happen in the implementation. Therefore, a semantic API version must be 1.0 or 1.2, for example.

The initial version of the Banking API (1.0) comprises the “List accounts” and “Read account” operations. Adding the “List transactions” operation is backward compatible, so we increment the API MINOR version (1.1). Adding the merchant name and address to the “List transactions” response has the same effect (1.2). Replacing the internal account number path parameter with an IBAN when reading an account or its transactions (/accounts/{accountNumber} to /accounts/{IBAN}) is a breaking change that requires incrementing the MAJOR version (2.0). Similarly, removing the list of cards always returned when reading an account requires a MAJOR version increment (3.0).

Date-based versioning identifies a version with a date, typically in ISO 8601 format, corresponding to the deployment date with day (YYYY-MM-DD) or month (YYYY-MM) precision; version 1.0 could be 2023-10-02 or 2023-10, and version 3.0 could be

2024-03-04 or 2024-04. Consumers can tell the most recent version but must look at the documentation to learn how they differ. Although they know there are breaking changes between versions 1.0 and 3.0, they still need to look at the documentation to determine which part of their code needs an update.

By default, I recommend using semantic versioning because it indicates the type of change and is the most widely adopted option. However, you may consider using date-based versioning when releasing frequently or regularly (every quarter, for example).

15.4.3 How the API version can be represented in a request

If different versions of an API are available, consumers need to indicate which version they want to use in their request. Alternatively, it's possible not to include the version in the request and use an out-of-the-band consumer configuration outside the API's standard request-response mechanism. Figure 15.10 shows options for semantic versioning, which apply similarly to date-based versioning. This section describes the possibilities, and section 15.4.4 discusses which should be used.

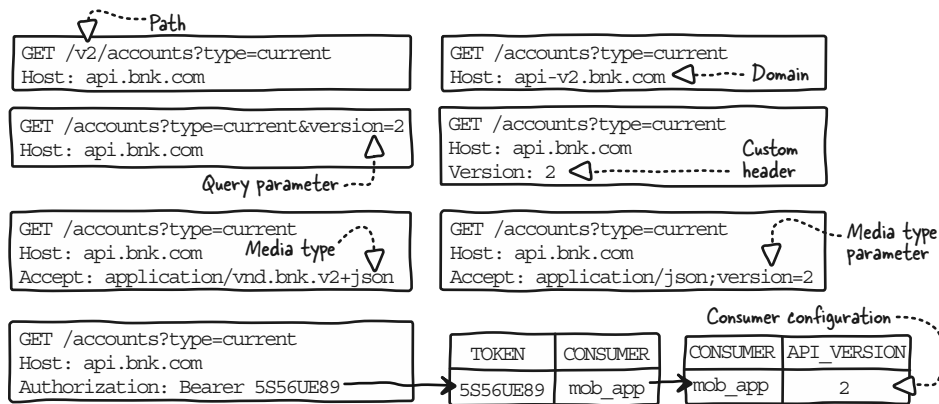


Figure 15.10 The API version can be indicated in the path, domain name, query parameter, header, or media type or an out-of-the-band configuration.

We can add the major version number to the path before the resource, like `/v2/accounts`. If the path includes the API name, it leads to `/banking/v2/accounts`. We use the major version because there's no need to differentiate minor versions at the exposition level; all operations in version 2.0 are backward compatible in version 2.1. A similar but rare option is to indicate the version in the domain name, such as `api-v2.bnk.com`. The version number can also be passed in request parameters, such as a query parameter (`GET /accounts?version=2`) or a custom request header (`Version: 2`).

We can use content negotiation and define a custom `application/vnd.bnk.v2+json` media type to indicate it in the `Accept` request header. However, this approach

limits content negotiation; for example, it's impossible to differentiate versions when reading a bank account as a PDF instead of JSON. To use any media type, we could use a media type parameter to indicate the version (`application/json;version=v2`). But although indicating that a version with a custom media type is common, using a media type version parameter is rare.

We can use the consumer configuration to specify the preferred API version. This should be used in conjunction with the ability to state the desired API version explicitly. Some providers set the version configuration automatically to the latest available version on the first API call (which doesn't indicate a version). They allow consumers to update their configuration later or indicate the version to use in their call, typically using a custom request header like `Version`.

15.4.4 Choosing how to represent the API version in a request

Choosing how to represent an API version was and may still be a source of heated debate. I recommend using path-level versioning by default; it's super simple to implement and use, which is likely why it's the most commonly used option.

Alternatively, a custom header (like `Version`) can be considered. A custom media type is also an option if content-negotiation limitation is not a problem. However, that can be trickier to implement. Also, developers using APIs (especially me) often forget to set version-related headers (`Version` or `Accept`). They can lose precious time figuring out why their call isn't working or returning the expected data. Out-of-the-band configuration can be interesting for public APIs when you promise to support many, if not all, past versions. I do not recommend using a query parameter, because doing so would lead to mixing request data (like search filters) and metadata (the version number): for example, `GET /accounts?version=2&type=current`.

15.4.5 When to choose an API version scheme and representation

There are two options regarding the choice of the API version scheme and its representation: waiting for the first breaking change or deciding from the start. I recommend deciding right from the beginning; that way, consumers know what awaits them, and transitioning to a new version will be easier. However, this doesn't mean you have to change the API version; you may, for example, add a `/v1` segment to your API path and never change it (see section 15.5.6). Furthermore, that's not a decision you'll make per API; your team or organization's APIs will likely share the same versioning policy.

15.4.6 Avoiding sub-API-level versioning

So far, we have discussed versioning the entire API, but it's technically possible to version sub-elements of an API, such as resources or operations. However, I do not recommend sub-API-level versioning because it's highly uncommon and will puzzle consumers (and you). The API will become an inextricable bag of knots, and consumers (and you) won't know which operations can be used together.

Based on the private APIs using it I have encountered, resource-level versioning can result from an overly resource-centric approach or the absence of separation between concept models and API; someone may have taken the REST resource concept too strictly and confounded the resource and API, forgetting that an API can deal with different resources and concepts to allow its users achieve something. The idea of resource versioning can also result from creating a do-it-all API that would benefit from being split (see section 11.2).

Suppose `/v2/transfers` and `/v3/transfers` represent two versions of the transfer resource in the Banking API. Similarly, we have `/v1/sources`, `/v2/sources`, `/v3/sources`, `/v1/sources/{sourceId}/destinations`, and `/v2/sources/{sourceId}/destinations`. How can consumers know which version of each resource to use to achieve a money transfer that requires reading sources (GET `/sources`), getting a matching destination (GET `/sources/{sourceId}/destinations`), and finally executing a transfer (POST `/transfers`)? Maybe all versions can be used together; maybe not.

That doesn't mean you can't version the data put in the bodies in addition to versioning the API. If you create standard data (such as ISO 20022 models) that can be used in other contexts, versioning data models and using different versions in the same operation can make sense. We can use content negotiation to handle model versions unless the model version is embedded in the data. But don't bother consumers unnecessarily with such two-level versioning.

I've only seen operation-level versioning on a few non-REST remote procedure call (RPC) APIs that use HTTP just for transport and don't care about its semantics. They can expose operations like POST `/deleteTransferV1`, for example (v1 being the operation version). The problems are the same as for resource-level versioning: how do you know which version of each operation can be used together?

15.5 *Carefully breaking and versioning an API*

API versioning involves more than bumping the API version number. Although incrementing the API version from 1.1 to 1.2 has no consequences because it involves backward-compatible changes, version 2.0 introduces breaking changes, which must be carefully considered. Evaluating the effects of breaking changes on the consumers and the provider is essential to ensure that they're possible and the benefits balance the cost. We can use the following questions to evaluate this:

- How many consumers are potentially affected, and who are they?
- Do consumers use what we break?
- Can we expose multiple API versions simultaneously?
- Does a versioning policy constrain us?
- What are the purposes of the breaking changes?

If breaking the API is not worth the cost, we can make design trade-offs, but accumulating trade-offs can have long-term consequences. Once we decide to break the API, providing runtime information about the deprecation of older versions can be helpful.

15.5.1 Listing consumers and their types

To evaluate the effects, we must know who consumes the API. The more consumers are affected, and the more distant they are, the trickier it is to introduce a breaking change. Modifying our Banking API to replace the amounts with a currency and a value object will likely affect most operations and, therefore, all consumers. If our team develops the only application using the API, we can easily fix it. If there are dozens of internal applications, the change will cost our organization a lot of money and time. If we have only two partners using the API, that doesn't seem like many, but these are customers; we may not want to bother them.



TIP We can sync the deployment of an API and mobile applications. The typical approach is to perform an API call or retrieve a static JSON file that indicates whether an update is needed when the application starts (Google Play has a force-update feature; the Apple App Store doesn't).

15.5.2 Checking whether consumers use what we break

Some or all consumers may not be affected by the modifications, possibly making it simpler to introduce a breaking change. Analyzing scopes, logs, and code to check how consumers use the API will help refine the evaluation of the effects.

Scopes limit consumers to specific operations. Replacing the “Read current year” dashboard with a more flexible “Read yearly” dashboard affects only those currently using it. Consumers lacking the `account` scope covering the replaced operation won't be affected.

Consumers may not use all API operations granted by their scopes. For instance, a consumer with the `account` scope may not call the “Read current year” dashboard operation. We can scan logs, typically at the API gateway level, to track operation usage, but some calls may not appear due to log retention. If end users focus on yearly statistics in December and logs are retained for six months, the calls to the “Read current year” dashboard operation may not be visible when checked in November.

Analyzing code is sometimes the only option to identify effects. Changing the `owner` string to an object in the “Read account” operation response is a breaking change. For security and efficiency, API logs typically exclude request bodies. If included, request bodies may show data sent by consumers, but logs do not tell how consumers use response body data. We must ask consumer developers to analyze their code to see whether the property is used, which is likely impossible with partner or public APIs.

15.5.3 Determining whether it's possible to expose multiple API versions

It's common to expose different API versions simultaneously to let consumers update their code when possible. However, we may be technically constrained. Architects and tech leads oversee API infrastructure and implementation. As an API designer, it's worth knowing the two options to handle API versioning: multiple instances of the

API or a single instance supporting different versions. Multiple instances can lead to excessive infrastructure costs and may not be technically feasible. For instance, maintaining an unmodified v1 instance is likely impossible with database schema changes introduced by v2. A single instance is the recommended option.

A common implementation technique involves request and response adapters; they transform a v1 request into a v2 request, allow v2 processing, return a v2 result, and convert it back to a v1 result. However, this can complicate implementation based on modifications and the number of versions. Some transformations may not be feasible due to missing data in the newer version or operation behavior modification.

15.5.4 *Complying with the API versioning policy*

Any public or partner API has a versioning policy that specifies how many past versions are supported and for how long (deprecation policy). The API versioning policy is managed by the API product owner, along with architects and tech leads, as it affects infrastructure and implementation. It can also outline when new versions are published and what changes they may include. This may limit how we can alter the API design. For instance, our Banking API policy may indicate that we support the last two major versions, with breaking changes annually and quarterly releases only with backward-compatible changes. Private APIs may have versioning policies, but enforcing updates on consuming applications is often challenging due to differing priorities and budget constraints among teams.

15.5.5 *Balancing effects and benefits of breaking changes*

We must balance the effects of the breaking changes with the benefits to our consumers and ourselves. Suppose our partners have requested that we modify the amounts to support multicurrency operations across all use cases. In that case, it's an acceptable breaking change that pleases our consumers, may attract new ones, and generates revenue. If we replace the "Read current year" dashboard with a "Read yearly" dashboard to fix a sub-optimal design, it brings nothing while requiring us to update various applications. We can decide not to do it or keep the old operation and add the better-designed one to be backward compatible.



TIP We must not hesitate to be pragmatic, especially with private APIs. If we're sure the breaking changes affect nobody, we can change only the minor version to avoid updating the base URL (`/banking/v1` to `/banking/v2`), which will affect all consumers. This may be risky with a partner or public API. Alternatively, we may keep elements we'd like to remove but mark them deprecated in the OpenAPI document (section 15.7.2).

15.5.6 *Accumulating trade-offs or breaking regularly*

Accumulating trade-offs in the long term may or may not be a problem. In some conditions, it may be better to break the API regularly. I usually recommend staying with version 1 of an API as long as possible to cover the domain, get feedback, and learn

what works. Only transition to version 2, which breaks everything, if it's worth the cost; awaited features may justify it (see section 15.5.5). Version 2 often doesn't occur, and we may have to live with a sub-optimal API forever, but it may suffice if it meets user needs.

However, long-term consequences must be considered. As consumer numbers grow, fixing sub-optimal APIs becomes more complicated. Accumulating trade-offs may make our APIs complex, lengthening the integration of private APIs. This may also make the API less engaging for new consumers, which is a problem for partner and public APIs. A strong API versioning policy can enforce transitions to newer versions, allowing for breaking changes. But partner or public API users may become frustrated with frequent updates, especially if they don't use the new features. We run the risk of consumers switching to competitors with more stable APIs. Although private API consumers are captive, it may be complicated to enforce updates as priorities and budgets differ between departments or teams.



NOTE Fortunately, we can limit the need for breaking changes and trade-offs with an extensible design; see section 15.6.

15.6 Creating extensible API designs

Introducing breaking changes and making design trade-offs is not a fatality. We can design our API and its modifications to minimize the effects of future evolutions and even limit the need for future evolutions. We already worked hard on that topic because we've learned to create versatile and flexible APIs, and with a few additions, we can make our APIs totally extensible. This section revisits our past learnings in the light of extensibility, illustrates how we can learn extensible design patterns from past decisions, and shows extensible design patterns.

15.6.1 Designing a user-friendly, interoperable REST API that does the job

By learning to design REST APIs that do the job and are user-friendly and interoperable, we implicitly saw how to limit the need for changes or introduce breaking changes at the data, operation, flow, and API levels, thanks to the following practices:

- Use REST uniform interface
- Use ready-to-use data
- Make operations accept extra input data
- Design flexible flows
- Choose the right size for APIs

The uniform interface of REST (see section 4.8.1) makes our API flexible. By looking for resources to represent right-sized and sometimes not obvious business concepts like money transfer destinations, our API is built on elements that are less prone to breaking changes. Thanks to content negotiation, we can also seamlessly add new data formats like CSV or PDF.

Data that doesn't need to be interpreted is flexible and less prone to breaking changes. In section 8.4.3, we learned to provide ready-to-use data; in particular, we decided to add a currency to our amounts. That way, consumers don't rely on hard-coded logic.

Operations that don't return an error when receiving extra data are good candidates for removing input data seamlessly. In section 9.8.1, we decided that updating a money transfer would accept all data, even unmodifiable data. The implementation ignores extra elements. This must apply to any operation; that way, input data can be removed seamlessly as long as the effects of the operation executed with ignored extra data are backward-compatible.

Flexible flows can avoid many later modifications. During needs analysis, we learned not to map flows to UI and expanded our skills to build user-friendly flows that give consumers total freedom. For example, in section 10.4, we designed an account application flow to collect applicant data in any order.

A big, do-it-all API is complicated for users to grasp, but it will also irremediably lead to many breaking changes. Fortunately, section 11.2.2 taught us to define right-sized APIs that focus on smaller pieces of subject matter. We split the do-it-all Banking API into three different APIs dealing with account information, transfers, and account applications. It's easier to make each one evolve individually as they have a smaller perimeter and possibly fewer consumers.

15.6.2 *Learning from past decisions*

Capitalizing on trade-offs, including unfixed erroneous past decisions, is essential for the next design decisions. For example, suppose we designed the Banking API's "Transfer" operation to manage only immediate transfers and return an `executed` Boolean value. It is `true` if the transfer is executed and `false` if the transfer is postponed due to the transfer subsystem's planned unavailability. We realized the `executed` flag was limited when we added the delayed transfer feature (transfer executed on a specific date). We need to indicate whether (1) the transfer is executed, (2) the transfer is delayed by the user, or (3) the transfer subsystem is unavailable. As we learned in section 15.2, we can't replace `executed` with a `status` property, but we can keep it and add `status`. It could be better, but it works. We can remove the `executed` flag when switching to version 2 (if that version ever comes). This incident teaches us that the next time we want to use Booleans, we should think about it carefully first.



TIP Establish a list of known design problems for your APIs, and add elements as soon as they are detected (post-deployment). This helps you avoid reproducing them, detect the need for clearer guidelines (section 16.3), and, if v2 happens, quickly determine what needs to be fixed.

15.6.3 Using extensible design patterns

We discovered that some design patterns are not extensible in section 15.6.2. The corollary is that others are extensible:

- Always use objects in bodies.
- Consider using arrays of objects over arrays of atomics.
- Use interpretable formats.
- Consider grouping similar data in arrays.
- Consider using enumerations over Booleans.
- Consider using operations over enumerations.

Always use an object at the request and response body level; array and atomic types can't be extended. If `GET /accounts` returns an array of accounts and we decide to add pagination metadata, we'll have to replace the array with an object and break the interface contract. A `POST /transfers` that returns a string will have similar consequences if we want to return more data.

Consider using arrays of objects instead of arrays of atomics unless you're sure you'll never need to add more data to each element. For example, if we have an owner's array of owner IDs in the `Account` data model, we can't add more owner data without replacing the strings with objects, introducing a breaking change. A `tags` array of strings on a transaction is acceptable as a tag string is self-sufficient.

Use interpretable and extensible formats whenever possible. For instance, instead of using static values like `MONTHLY` or `QUARTERLY` to represent a money transfer recurring period, it's better to use ISO 8601 durations like `P1M` and `P3M`, which are easily understandable and can be seamlessly extended: we can add a weekly option (`P1W`), for example.

Grouping similar data in arrays allows us to add elements that consumers will integrate seamlessly if they don't interpret the data. For example, a money transfer may have multiple dates, such as `creationDate`, `executionDate`, and `validationDate`, which can be grouped in an `events` or `statuses` array where each element has a date and status (`created`, for example). If consumers loop on the array without interpreting data, adding a new status, like `canceled`, is fine. However, it may cause a breaking change if consumers interpret it.

Booleans must be used carefully because they support only two values (`true` and `false`). As seen in section 15.6.2, our `Transfer` data model can have an extensible enumerated status string (`executed` and later `delayed`) instead of an `executed` Boolean flag. However, remember that adding new values to an enumeration used in output can be a breaking change.

If an enumeration changes regularly, consider adding an operation to return values (and extra information). For instance, transaction categories frequently evolve, and using a static enumeration for a search filter can be problematic. To address this, we add a `GET /transaction-categories` operation to return category names (accompanied by icon URLs).



CAUTION Extensibility must not lead to an ultrageneric but ultracomplex design that covers all imagined futures that will never happen. Be careful not to neglect other design aspects, especially meeting user needs and being user-friendly.

15.6.4 *Providing deprecation runtime information*

When exposing multiple versions of an API, each supported for a limited time, it can be helpful to indicate future deprecation programmatically. When we know an API version will be deprecated, we can add to all responses a `Sunset: Wed, 31 Aug 2024 23:59:59 GMT` response header defined by RFC 8594. The HTTP date it contains indicates when the API will be deprecated. However, it's only helpful if consumers or the infrastructure actually checks the presence of this header to do something about it. An API SDK created by an API provider typically uses this to output warnings in the consuming application logs.

15.7 *Describing the design modifications with OpenAPI*

Clear and exhaustive information about the API design modifications is essential so that design stakeholders, including us, can better assess whether the modifications meet user needs and make an informed decision regarding breaking changes (section 15.1.3). Implementation developers need this information to update the API properly, and consumers will use it to update their code if necessary. This section discusses using OpenAPI to do so by

- Indicating the API version
- Deprecating elements
- Adding a changelog



NOTE Section 19.1 discusses our role in providing information to the developers of the API implementation (Develop stage) and consuming application developers (Consume stage).

15.7.1 *Indicating the API version*

As shown in figure 15.11, the API version identifier is indicated in the `info.version` field in an OpenAPI document; we can also use it to indicate the document version. Add quotes if the API version is a number; use `version: "1.2"` instead of `version: 1.2` to prevent parsers from interpreting the version as a number instead of a string. A semantic version number conveniently indicates whether the modifications are backward compatible, but you can use other schemes (section 15.4.2).



CAUTION An OpenAPI document generated from the implementation may have the implementation version in `info.version` instead of the API version (section 15.4.1). Check the generator documentation to see how to override this behavior.

Modifications can be made to the OpenAPI document without affecting the interface contract; the `info.version` value can track document versions by adding a

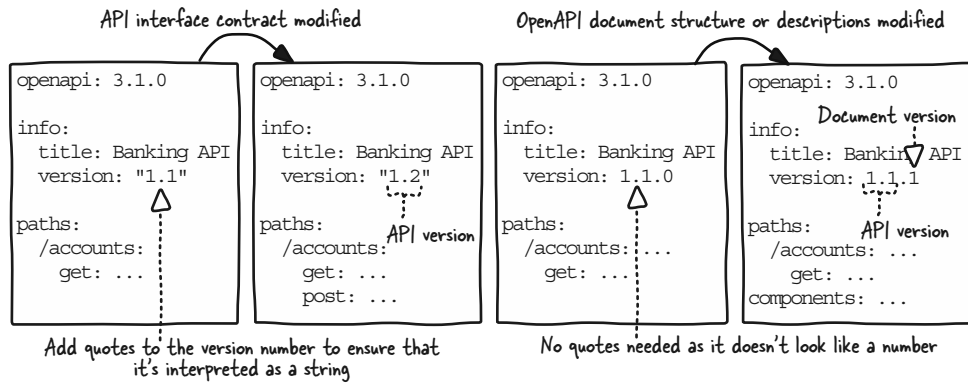


Figure 15.11 We can indicate the API version or API plus document version in `info.version`.

patch version. For example, a document describing version 1.2 of an API may show `version: 1.2.0` (no quotes, as it's a string). Clarifying descriptions or reorganizing data models (like using components instead of inline definitions) can result in `version: 1.2.1`. This detail is often unnecessary for private APIs.



NOTE Update the base URL in `servers` if the version appears in the domain (`https://api-v2.bnk.com`) or base path (`/v2` or `/banking/v2`); see section 11.3.3. For other version representations (section 15.4.3), update the necessary header, query parameter, or specific media type on all operations.

15.7.2 Deprecating elements

As shown in figure 15.12, in an OpenAPI document, we can indicate that an operation, parameter, response header, or property is deprecated. The `GET /current-year-dashboard` operation and `cards` property of the `Account` schema are deprecated. The deprecated flag indicates that an element shouldn't be used anymore but is kept for backward compatibility. This only requires updating the minor version of the API, as it's not a breaking change.



TIP When sharing the OpenAPI document with consumers, you can remove deprecated elements to ensure that only existing consumers use them. New consumers won't use them because they are not visible in the documentation. However, output data remains visible when calling the API.

We could consider using this flag in an intermediate version of the OpenAPI document to indicate the elements to remove from the new version. However, it's less precise than the output of an OpenAPI diff tool, which can also provide that information, among many other things, such as indicating that a value is removed from an enumeration or an optional parameter becomes required.

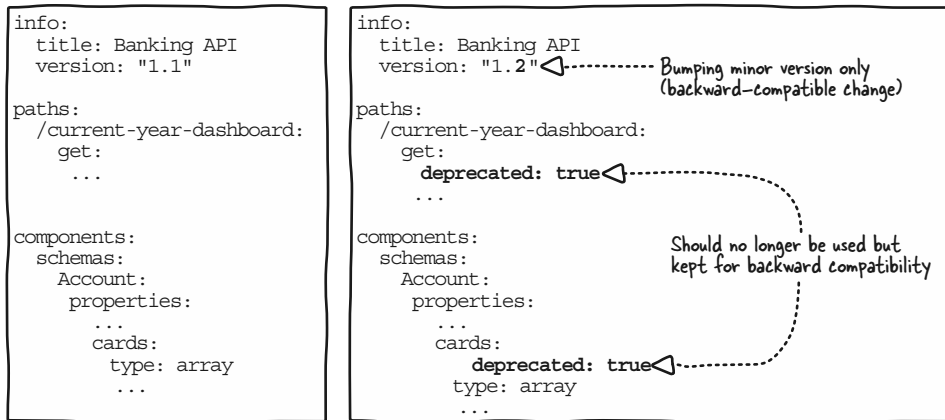


Figure 15.12 Use the deprecated flag to mark elements that should no longer be used but must be kept to maintain backward compatibility.

15.7.3 Adding a changelog

Figure 15.13 shows that we can use descriptions and tags to add changelog information to an OpenAPI document. Because all description fields are Markdown compatible,

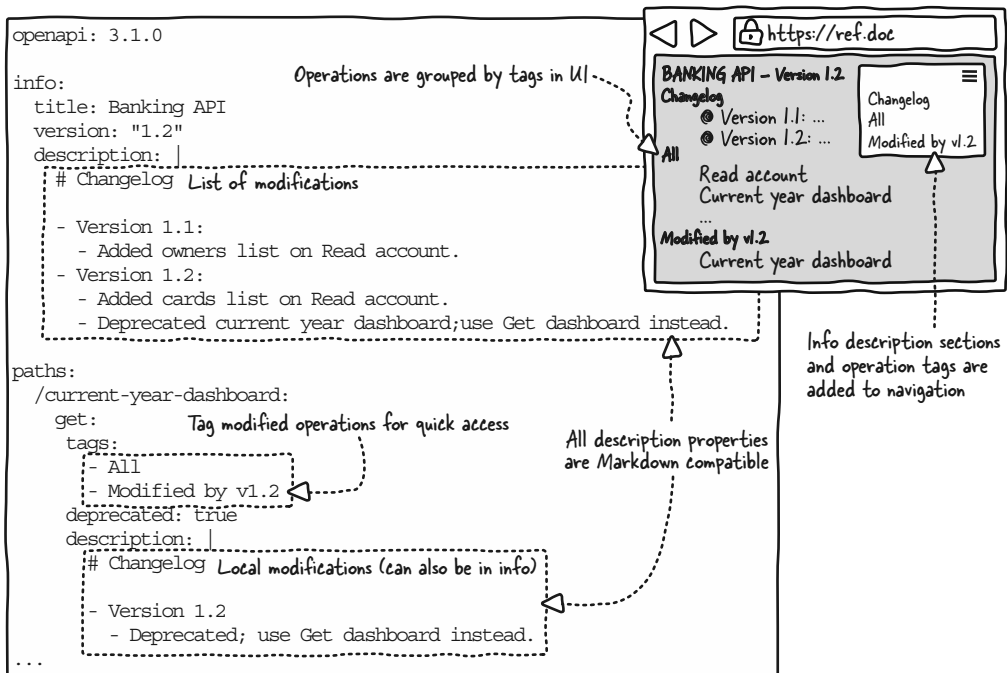


Figure 15.13 Use Markdown to add changelogs to the API and any element that supports description. Consider adding tags to highlight modified operations.

we can add a `# Changelog` section with a bulleted list to the `info.description` field. We can do the same in the `description` of an operation, parameter, header, response, or schema to provide more detailed information.



TIP Use the OpenAPI diff tools discussed in section 15.2.8 to list changes and feed AI with them to describe the modification in plain English (or French). Some tools may handle all this for you seamlessly. However, you'll need to manually add modifications that aren't visible in the OpenAPI document, such as response behavior (section 15.2.5) and the invisible contract (section 15.2.7).

Operations have a `tags` list that OpenAPI UI renderers can use to group operations. We may want to use this feature to highlight new and modified operations, which can be helpful for design stakeholders, implementers, and consumers. All operations have the "All" tag, and the operations affected by version 1.2 also have a "Modified by v1.2" tag.



NOTE OpenAPI tags can be used to provide an overview of concepts and use cases; see section 19.2.

Summary

- Modify an API as you design it. Fulfill user needs; be user-friendly, interoperable, secure, and efficient; and adapt to the context.
- To design backward-compatible output data modifications, only add or modify data within the original data's limits.
- To design backward-compatible input data modifications, only make required data optional, add new optional data, or expand the limits of existing data.
- Be careful when removing operations, replacing HTTP status codes, modifying flow steps, or modifying security schemes, as these are breaking changes.
- Be careful when modifying security scopes, which may lead to breaking changes and security breaches.
- Version the interface contract separately from the implementation.
- Prefer the most common semantic versioning, which indicates the type of change. Consider using date-based versioning when releasing frequently or regularly.
- Prefer the most common path-level versioning. Alternatively, consider using a custom header. Media type versioning is also an option, but it constrains content-negotiation possibilities.
- Decide on the versioning scheme and representation from the beginning so that consumers know what awaits them and transitioning to a new version is easier.
- Check how many consumers are affected, who they are, and any technical or contractual constraints to evaluate the effects of breaking changes.

- Balance the effect of non-negligible breaking changes with benefits for your consumers and your organization.
- Accept design trade-offs, stick to version N as long as possible, and move to version $N + 1$ if new features justify it.
- Use ready-to-use data, make operations accept extra input data, design flexible flows, and avoid do-it-all APIs to create an extensible design.
- Use objects in bodies, group similar data in arrays, use interpretable formats, and consider using arrays of objects over arrays of atomics, enumerations over Booleans, and operations over enumerations to create an extensible design.
- Update the version, add a changelog to the description, and indicate deprecated elements in the OpenAPI document so the implementation's developers and consumers are aware of the modifications.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 15.1

An online course platform proposes an API to selected third parties. Its initial version allowed users to search for courses. A new version of the API allows the addition of new courses. Listing 15.1 shows an excerpt of the OpenAPI document describing the API's new version. Can you spot any problems? If so, how could you fix them?

Listing 15.1 Online course platform OpenAPI document

```
openapi: "3.1.0"

info:
  title: Online Course Platform
  version: athena-build.20250117.1830

paths:
  /v1/courses:
    get:
      summary: Search courses
      ...
  /v2/courses:
    post:
      summary: Add a course
      ...
```

Exercise 15.2

Modifications were made to the request body of a Fitness API's "Create workout" (POST /workouts) operation. Listing 15.2 shows the request body JSON schema

before the modifications, and listing 15.3 shows the modified version. List all changes made to the schema, and evaluate whether they are backward compatible.

Listing 15.2 Request body data before modifications

```
properties:
  difficulty:
    type: string
    enum:
      - easy
      - challenging
  duration:
    type: integer
    description: Minutes.
  activities:
    type: array
    minItems: 1
    items:
      required:
        - name
      properties:
        name:
          type: string
          description: Plank, burpees, etc.
        dur:
          type: number
          description: Seconds.
        repetitions:
          type: integer
          description: Number of repetitions
required:
  - difficulty
  - duration
  - activities
```

Listing 15.3 Request body data after modifications

```
required:
  - duration
  - activities
properties:
  duration:
    type: string
    description: ISO8601 duration.
  difficulty:
    type: string
    enum:
      - easy
      - challenging
      - insane
  activities:
    type: array
    minItems: 1
    maxItems: 20
```

```

items:
  required:
    - name
    - repetitions
  properties:
    name:
      type: string
      description: Plank, burpees, etc.
    duration:
      type: string
      description: ISO8601 duration.
    repetitions:
      type: integer
      description: Number of repetitions

```

Exercise 15.3

Modifications were made to the response body of an Industrial Machine Monitoring API's GET /machine-reports/{reportId} operation. Listing 15.4 shows the response body JSON schema before the modifications, and listing 15.5 shows the modified version. List all changes made to the schema, and evaluate whether they are backward compatible.

Listing 15.4 Response body data before modification

```

required:
  - id
  - machineIdentifier
  - conditionComment
  - conditionStatus
properties:
  id:
    type: string
    pattern: "[0-9]{10}"
  date:
    type: integer
  conditionComment:
    type: string
  conditionStatus:
    type: string
    enum:
      - green
      - orange
      - red
  machineIdentifier:
    type: integer
  status:
    type: string
    enum:
      - running
      - stopped
      - maintenance
  temperature:

```

```
type: number
description: Celsius.
```

Listing 15.5 Response body data after modifications

```
required:
- id
- date
- machineId
- condition
properties:
  id:
    type: string
    pattern: "[a-z]{3}-[0-9]{10}"
  date:
    type: string
    format: date-time
  condition:
    required:
    - conditionComment
    - conditionStatus
    properties:
      conditionComment:
        type: string
      conditionStatus:
        type: string
        enum:
        - green
        - orange
        - red
  machineId:
    type: integer
  status:
    type: string
    enum:
    - running
    - stopped
    - maintenance
  temperature:
    type: number
    description: Fahrenheit.
  pressure:
    type: number
    description: Kilopascals.
```

Exercise 15.4

An Event Management Platform API has a `GET /events` operation that returns an object with a data array of events. It doesn't support pagination and returns all available data. Could you decide that the server will now return a maximum of 50 events and add an optional `page` query parameter allowing consumers to get other pages of events? If that's a problem, how could you safely enable pagination?

Exercise 15.5

A Document Management API uses the generic file design pattern to handle document content. Do you see any problems with the design of the `POST /files` operation in listing 15.6? If so, how could you fix them?

Listing 15.6 Add file operation

```
paths:
  /files:
    post:
      requestBody:
        content:
          application/octet-stream: {}
      responses:
        "201":
          description: File stored.
          content:
            application/json:
              schema:
                type: integer
                description: The ID of the created document.
```


Part 4

Scaled and simplified API design

We have designed an API that does the right job, is user-friendly and interoperable, and integrates necessary constraints like security and efficiency. But doing so has involved many design decisions and tasks, some with huge effects. We must be equipped to make our work sustainable in the long term and at scale because we, as well as our colleagues, will continue to enhance the API we designed and also design new APIs. For example, integrating pagination into an API design is challenging because of the many design options; we must choose the “right” one without taking ages, but what is “right”? Once we decide, pagination must remain consistent across APIs, regardless of who designs them, enhancing interoperability and user-friendliness. Additionally, not re-discussing pagination for each new search operation speeds up the design and allows us to focus on more essential concerns like meeting user needs.

Design is also just one stage in the API lifecycle, and our work supports the next steps, especially implementation. How can we better equip developers with essential information for accurate implementation? For example, how can we clarify that account balances should come from the `BALX` table instead of `BALY`?

This part of the book discusses the final design layer: using a reasoned and continuously improving process (section 1.7.4). We also address the last gaps in the design process (section 1.6), linking our efforts with the next stages of the API lifecycle. Chapter 16 covers simplifying design decisions with user-friendly API design guidelines. Chapter 17 focuses on optimizing OpenAPI documents

for consistency and simplified authoring. Chapter 18 describes automating guidelines to ensure consistency and free our minds of details. Chapter 19 concludes the book by discussing the final design step: enhancing the API design artifacts we created to build a design reference kit that streamlines our work, ensures accurate implementation, and supports the entire API lifecycle.

16

Facilitating API design decision-making

This chapter covers

- Making design decisions confidently and consistently
- Researching solutions to API design questions
- Creating and evolving API design guidelines

Should we use an IBAN or an account number to identify a bank account? “Owner” or “user”? `/accounts` or `/account`? 403 or 404? Boolean or string? How do we handle pagination? Designing an API involves countless decisions to fulfill user needs; be user-friendly, secure, efficient, and extensible; integrate contextual constraints; and not break consumers. Any decision can have significant consequences or introduce inconsistency. We may struggle to choose one option among many, have no clue how to solve a problem, or endlessly repeat the same discussions (sometimes with different conclusions). All this can make API design decision-making inefficient, inconsistent, and daunting.

The design process, patterns, tips, and tricks we have learned so far contribute to mitigating this. However, this book can hold only some of the answers to design questions, and its answers may not be adaptable to all contexts. It’s essential to establish a clear decision-making process and learn how to research solutions to design questions to make confident decisions efficiently. We must also record why

we made certain decisions and turn them into actionable API design guidelines to make our decisions consistent. API design guidelines also streamline the design process by removing the need to make many decisions.

API design guidelines are often seen as essential when multiple people work on APIs within an organization. However, they are invaluable assets even when working alone, because we don't always remember what we did last summer. This chapter discusses making design decisions confidently and consistently, researching solutions to API design questions, and creating and evolving API design guidelines.

16.1 Making design decisions confidently and consistently

As shown in figure 16.1, this chapter focuses on the last design layers introduced in section 1.7.4. We must fill our API design toolbox with the thinking process, tools, and guidelines that will simplify our work and help us be confident and consistent, whether we're facing old or new problems and questions.

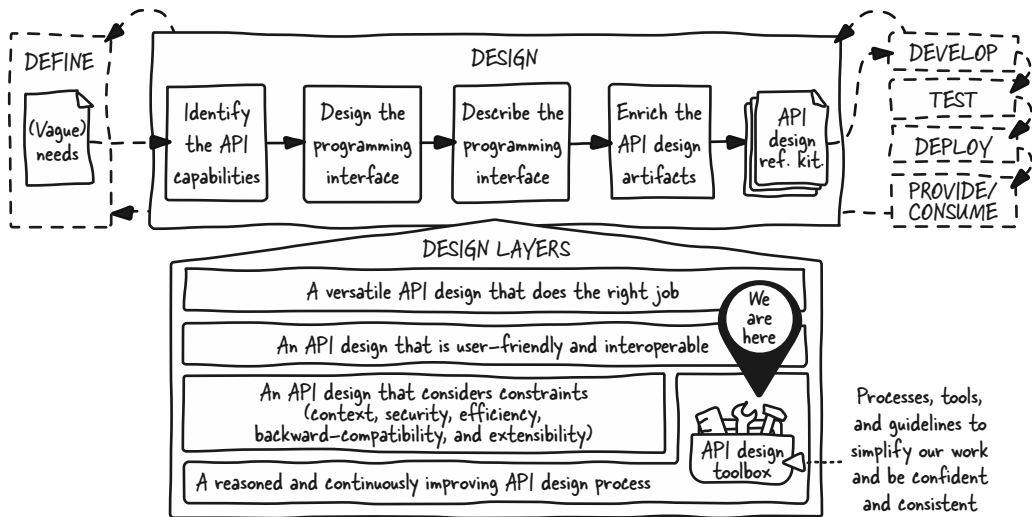


Figure 16.1 Designing APIs also involves thinking about simplifying our work so we can be seamlessly confident and consistent.

The needs analysis, resulting API Capabilities Canvas, design methodology, principles, tips, and tricks we have learned provide solid foundations for designing APIs. However, even with experience, we may not have the answer to a design question or may feel unsure about a design decision. We may also do things differently over time and inadvertently introduce inconsistency. To overcome doubts and make decisions confidently and consistently, we can do the following

- Ensure that it's the right time to make a decision.
- Evaluate the scope of the decision.

- Decide based on trusted past decisions.
- Decide based on trusted external sources.
- Back our decision with reasoning and sourced information.
- Explain out loud.

16.1.1 Ensuring that it's the right time to make a decision

Not every question needs immediate answers. We've learned to separate concerns during the design process to ensure efficiency and keep debates and thinking focused. For example, we won't spend an hour debating whether a money transfer should have an `executed` Boolean or an enumerated `status` string during the first pass through data modeling. Instead, we will include both or a single `executedOrStatus` in our model, with notes in the property description. After the initial data modeling, we can reconsider these details based on user needs, user-friendliness, and extensibility. This may seem like procrastination, but it's not. We focus on the right problems and gather more information, making decision-making easier.

16.1.2 Evaluating the scope of the decision

Not all decisions will have significant effects. Understanding the problem's scope reduces decision-making pressure and ensures that we invest time only when needed. A decision such as choosing a pagination type (section 13.6), whether to use the "Problem Details for HTTP APIs" format for errors (section 9.8.6), or naming the account resource identifier `accountId` or `id` (section 8.9.3) affects all our APIs and must be carefully considered. However, we won't make such decisions daily. Often, our work involves local concerns within our API or its operations. For instance, whether to have a single resource for all accounts or separate ones for checking and savings accounts is a matter specific to our Banking API.

16.1.3 Deciding based on trusted past decisions

The most straightforward decisions are those already made by us or our colleagues, but we must ensure that these decisions are trustworthy. We can take inspiration from existing APIs, but API design guidelines are a more reliable source for past choices.

We stressed the importance of consistency and interoperability when discussing data (section 8.9), operations (section 9.10), flows (section 10.1), and API usability (section 11.4). Aligning our design with our organization's other APIs relieves us of the burden of decision-making. Problems like partial updates, pagination, file uploads, and choosing an interoperable account identifier may already be resolved. However, finding a trustworthy solution in our existing APIs can be challenging. Different teams may have found various valid solutions. A popular choice may be outdated or unsuitable for our context. We may miss some APIs in our research. Having many APIs and opinions can make decision-making uncertain, inefficient, and error-prone.

This is why many small teams and large organizations create API design guidelines: to standardize solutions and simplify decision-making (usually after creating divergent

APIs and being frustrated about reinventing the wrong wheel). We'll learn more in section 16.3.

16.1.4 Deciding based on trusted external sources

We won't always find design solutions in our API design guidelines, but answers most likely exist in the outside world. Remember how we emphasized copying generic and domain-specific practices and standards to foster consistency and interoperability? This also helps in decision-making.

Common API problems like pagination, searching, and file uploads have established solutions we can adopt instead of reinventing the wheel; some are already implemented in our development frameworks. Beyond common practices, we can use generic standards, like the CloudEvents format for webhooks (section 14.6.4). Industry-specific practices and standards can significantly help us; for example, Banking APIs created by different companies can share similar data and patterns or use standards like ISO 20022 (section 14.2.2).

However, the solutions we find may be wrong or not adapted to our context, or different valid solutions can exist. It's essential that we carefully benchmark what we find. Check out section 16.2 to learn more about researching solutions to API design questions.

16.1.5 Backing decisions with reasoning and sourced information

Whatever the scope of a design decision, explaining why we make it with reasoning and sourced information avoids doubt and ensures its validity. For example, in section 8.9.3, we used a generic name, like `id`, for the resource identifier in resource data models. The reasoning was that this property could be immediately identified and is the same whether it's an "Account," "Transaction," "Product," or other resource in any of our APIs. This design pattern makes our APIs user-friendly and interoperable.

However, our reasoning may be biased; trusted information must back it up. We can add to our reasoning that this pattern is adopted by many APIs in the outside world (section 16.1.4) and can ask for confirmation from developers of API consumers. Alternatively, if it's not our first API, we'll likely say that this is what is defined in our guidelines (16.1.3).

But not all decisions require extensive reasoning and research. For instance, there's no need to waste time considering why we use the term "account" in our Banking API, as there's no ambiguity in the banking world and our use case.



NOTE Remember that we have the API Capabilities Canvas to help us with needs-related decisions, especially flow design.

16.1.6 Explaining out loud

When working alone and struggling to solve a problem, it's helpful to express the problem and reasoning out loud instead of thinking in our heads. Hearing the words can lift all doubts or help us figure out what's wrong and make adjustments. Developers often use a similar technique called *rubber ducking* when debugging code; they talk to an inanimate object, traditionally a rubber duck (but you can use whatever you like).

Explaining the reasoning behind a decision out loud when working with others is also a good idea. It fosters a deeper understanding of the decision. This approach can also help reduce the risk of heated arguments, especially about “controversial” topics (such as pagination and hypermedia APIs).

16.2 *Researching solutions to API design questions*

Generic design questions that apply to all API operations and across APIs, such as pagination, error formats, naming patterns, and complex search filters, require careful consideration. These questions are relevant across various APIs you and others will design, not just the one being designed at the moment. Questions can also pertain to the API’s subject matter, such as which identifier to use for an account, affecting many operations and interoperability across APIs. We may create solutions or adapt existing ones, but we must justify our decisions with reasoning based on reliable sources. This section outlines how to research solutions and document our findings, thinking, and decisions.

16.2.1 *Where to research solutions to design questions*

For general-purpose or domain-specific design questions, we can check the following internal and external sources to find ready-to-use solutions or inspiration to help us build our solutions:

- Our API design guidelines
- Our organization’s APIs
- Our usual API development stack
- API or data standards
- Others’ API design guidelines
- APIs of other organizations
- Books, articles, or videos

We may already have the solution in-house. Our first option must be to search for solutions in our API design guidelines (if they exist); we’ll discuss those design guidelines in section 16.3. They likely explain how to structure a path or handle pagination. We can also check our organization’s APIs, although their general-purpose design patterns should already be in our guidelines. Our development frameworks often come with out-of-the-box generic features useful for our API. For example, Spring Boot (Java), Django (Python), or Express.js (with `express-paginate` middleware) support pagination; we may use their default pagination parameters in our design. However, we must be careful not to introduce inconsistency in a multi-stack environment.

We can use API or data standards and look at what others do. In this book, we relied heavily on the HTTP specification. We also discovered generic standards like Problem Details for HTTP APIs for errors and CloudEvents for webhooks that apply to any domain. We may also use domain-specific standards, such as banking’s ISO 20022 or telecom’s TM Forum Open APIs or CAMARA project. Many organizations share API design guidelines that likely include solutions to our problems, such as

Microsoft (<https://github.com/microsoft/api-guidelines>). Additionally, public APIs can be helpful references. For instance, we can look at e-commerce APIs to model a person's address. Valuable insights also come from tech websites, blogs, forums, videos, and books, offering benchmarks, pros and cons, and context for solutions.



NOTE We used several RFCs to guide our decisions, including those describing the HTTP protocol. You may find solutions in one of the many other RFCs at www.rfc-editor.org. I recommend checking the Best Current Practice category.

16.2.2 *Searching and considering*

A good-old search engine can help us find information. For example, we can search for “API guidelines,” “API stylebook,” and “API style guide” when looking for API design guidelines. We can also let AI do the searches and summarize them for us. Try a prompt like “How do you represent search filters greater than or equal to in a REST API?”



CAUTION AI can make mistakes; I suggest using AI-driven search engines that show their sources. This allows us to verify the sources and assess the quality of the information (particularly if it wasn't fabricated) and its relevance to our context.

Whatever means we use, we must objectively consider our findings. Always benchmark the solutions to evaluate how common they are. For example, an AI-powered search engine may list JSON-encoded filters when responding to our question about greater-than-or-equal-to filters (`GET /transactions?filter={"amount":{"$gte":100}}`). Asking, “Are JSON-encoded filters very common?” will show us that it's not a common pattern. Therefore, we shouldn't use it.

It's also essential to objectively and factually ensure that found solutions fit our context, especially concerning fulfilling user needs, efficiency, security, and design consistency. We can use what we've learned in this book and our guidelines to help us. Additionally, as is the case for software architecture, “famous company A does it” is likely not a valid argument; it's even OK not to agree with the design of other organizations' APIs. Similar problems in different contexts may lead to different (good and bad) solutions. Understanding why we disagree or why a solution we like was chosen is essential, especially for structuring decisions like choosing the type of API to create (REST or other).

16.2.3 *Using an architectural decision record format*

It's essential to record structuring decisions to remember them and be able to understand and explain them later, but also to force us to justify them. I recommend using an architectural decision record (ADR) as a guide. ADRs were initially intended to record justified design choices that address significant functional or nonfunctional requirements for software architecture. An ADR can be a Markdown file or a wiki page. A project may need multiple ADRs; they constitute a decision log. The ADR concept can be extended to other domains (any decision record), including API design.

In that case, the “project” can be the API we design or all our APIs. Most API design-related ADRs will deal with cross-API concerns. They will serve as foundations for our API design guidelines (see section 16.3).

This section provides a quick overview of using an ADR in the context of API design; I recommend exploring the homepage of the ADR GitHub organization (<https://adr.github.io/>) to learn more about the various ADR formats, practices, and tools. I like the MADR (Markdown any decision records) format, which has short and long versions (<https://adr.github.io/madr/>). I usually use the long version of the MADR format for my API design decision record because it allows me to fully describe the options I considered, my reasoning, and sources of information. Figure 16.2 shows an example involving resource ID property names.

```
# Resource ID Property Name ◀----- The question or problem to solve

## Context and Problem Statement ◀-----

Each resource data model must contain the unique identifier of the
resource; how to name the property holding it?

## Considered Options ◀----- The possible solutions

* Generic name (`id`)
* Name that includes the resource (`accountId` or `ownerId`)

## Decision Outcomes ◀----- The chosen solutions with good and bad consequences

We chose the Generic name (`id`) option because it's interoperable and
easily identifiable. A minor downside is that its content can't be
determined when seen alone. It's used in many APIs, including GitHub
or Stripe.

## Pros and Cons of the Options ◀----- Info about each solution (links, description)
and detailed analysis (pros and cons)

### Generic Name

* Pro, immediately identifiable among other IDs (`id` vs. `ownerId`).
* Pro, interoperable; all resource IDs are all named `id`.
* Con, not identifiable if seen alone (`id` could be the ID of an
account or an owner). However this element will always be used in the
whole resource data context.

### Name That Includes the Resource

* Pro, what kind of ID is in the property is immediately identifiable
* Con, not interoperable; each resource ID has a specific name.
* Con, the resource identifier is not differentiable from other
resource identifiers within the data (`accountId` vs. `ownerId`)

## More Information ◀----- Comments and sources

- `id` used in many APIs such as GitHub or Stripe (Twilio uses `sid`)
- Some Salesforce APIs use `resourceId` and others `id`
- The Design of Web APIs, Second Edition recommends `id`
```

Figure 16.2 This example of ADR uses the MADR template. It’s a Markdown file, but you can also use a wiki page or any other format. It describes the problem, solution, reasoning, and sources of information.

This example of ADR is a final result. I started framing the problem in the Context and Problem Statement section. After some research, I listed the options and described and analyzed them in the Pros and Cons of the Options section (I could also have added links to option-specific sources). In the process, I added comments and sources of information in the More Information section. Finally, I wrote the decision outcomes and consequences (both good and bad).

16.3 What are API design guidelines?

ADRs are valuable decision logs but are not the most effective day-to-day guides for API design: this is the role of API design guidelines. API design guidelines can help us across all steps and layers of API design, rather than being a constraint that only enforces a certain level of consistency across APIs. They are a user-friendly guide that simplifies decision-making and helps create user-friendly, secure, efficient, consistent APIs. Although often linked with API governance, API design guidelines can stand alone. They are invaluable assets, whether we work alone creating or evolving a single API or are working on many APIs with colleagues.

This section discusses how API design guidelines help us, how they relate to API governance, and when we need them. Section 16.4 describes what we can put in user-friendly API design guidelines, and section 16.5 shows how to build them iteratively.

16.3.1 How design guidelines can help us

Relying solely on ADRs for API design can make our work complex. For instance, designing a “Search transactions” operation may require reviewing several ADRs to decide on resource paths, HTTP status, response formats, pagination, filtering, and error handling. This process is cumbersome and prone to oversights. Moreover, our ADRs may not cover every aspect of API design.

With API design guidelines that provide actionable knowledge based on the solid foundations of our ADRs, we can quickly and confidently design this operation by consulting the “How to search or list elements” guide that explains everything in one place (without all the details about the why of everything, but including links to relevant ADRs) and offers us a complete example. The risk of missing something is minimal, especially when we complement our textual guidelines with tools that analyze our OpenAPI documents and identify errors or possible enhancements (such a tool is called a *linter*). The guide can also offer suggestions on crafting user-friendly search filters and advice on the pagination type, which should be customized to our specific context.

API design guidelines help us focus on fulfilling user needs and delivering value without burdening ourselves with endless discussions about details. If the “How to search or list elements” guide provides all the details on error handling, we won’t need to debate for the hundredth time whether the “Search transactions” operation should return `200 OK` with an empty list or `404 Not Found` if no transaction is found. Instead, we can concentrate on ensuring that the `TransactionSummary` data model provides the necessary data.

16.3.2 How API design guidelines relate to API governance

API design guidelines are often associated with *API governance*, which aims to guide how an organization handles APIs. To describe API governance, I usually say that it's like "the API police yelling at people who don't follow (complex and impractical) API design rules" or "the resource and support that facilitate creating APIs at scale." The "API police" approach focuses too much on ensuring design consistency across the entire API landscape. It isn't concerned with facilitating the work of the people creating APIs and often neglects user-friendly design aspects. I obviously recommend the other approach and see governance as an enabler that uses guidelines, tools, training, advocacy, and support to help people create APIs. Such API governance helps people with all design layers and covers the entire API lifecycle, not just design. To learn more about API governance, I recommend watching my Human-Centered API Governance presentation (<https://apihandyman.io/human-centered-api-governance/>) and reading the many related posts on my blog (search for "governance" on <https://apihandyman.io>).

16.3.3 When do we need design guidelines?

Although API design guidelines are associated with API governance, they can exist without it. API guidelines are helpful when we're working alone, even for personal projects. We will likely create and evolve many APIs, and the same guidelines can serve us over time. If we're in a team or organization and work with several people on the same API or multiple APIs belonging to a whole, it's essential to share a common guide, as we may not have learned to design APIs in the same way.

API design guidelines may not always include all the details and additional artifacts described in this book. Adapt the level of detail to your context; but remember that people come and go in small teams or larger organizations, and having ready-to-use guidelines can speed up the integration of newcomers who may not be as experienced as you.

16.4 What to put in user-friendly API design guidelines

Before learning how to build API design guidelines (section 16.5), this section discusses what we can include in them. Our guidelines can comprise any information and artifacts that will facilitate our API designer's job:

- Principles and rules
- Actionable recipes
- OpenAPI templates and libraries
- Tool to automatically check a design (linter)
- Meta-information about the design process
- Implementation or architecture considerations

16.4.1 *Listing principles and rules*

The most basic API design guidelines contain a list of generic or domain-specific principles and rules. These can include authorized HTTP methods (`PUT` versus `PATCH`, or both) or HTTP statuses (400 versus 422), resource path naming conventions (plural versus singular), the type of pagination to use by default (cursor versus index), and associated parameters and metadata. A rule can indicate that any error must use the Problem Details for HTTP APIs, and another can state that when a resource is not found, the type of the problem must be `resource-not-found`.

The principles and rules are not limited to general REST (or other API types) design concerns; they can also cover domain-specific concerns. For example, a rule can indicate that an account must always be identified with an IBAN. Ideally, all these rules are connected to the ADRs that explain them.



TIP Follow the **MUST**, **SHOULD**, and **MAY** definitions in RFC 2119 (www.rfc-editor.org/rfc/rfc2119.html) to describe requirement levels: what is mandatory, recommended (mandatory in a specific context), or optional.

16.4.2 *Providing actionable recipes*

Limiting API design guidelines to a (vast) set of fine-grained rules is unfortunately common, and it is the surest path to being unable to follow them. How can we know all the rules to use together in a specific case? We must group rules in actionable recipes (a rule can appear in more than one recipe) that we can use to achieve a specific design task, making our guidelines user-friendly.

Recipes don't just list rules; they guide us in detail. For instance, the “How to search for list elements” recipe should explain the operation's purpose (“listing or searching for elements in a collection”), extra features (filters, pagination, sorting), and behavior (success, errors, handling no results). It should detail the resource path, parameters (name and location of filter, pagination, and sorting parameters), and all possible responses (context, HTTP status, body data, and headers).

Recipes (and rules) may not be applicable in all contexts; it's essential to indicate when to use them or whether there are limitations. For example, the “Search operation” recipe could contain a callout about operations dealing with sensitive search filters and a link to “How to search for elements with sensitive filters.”

Recipes can cover typical flow questions, such as “How to integrate file upload into an operation flow” or “How to collect data in a multistep flow.” They can also provide an overview of specific aspects, helping us understand the global design, create new recipes, or implement the API. For example, the “How to handle errors” recipe may explain which HTTP status code to use and how to use the Problem Details for HTTP API for errors based on the situation.

16.4.3 Providing ready-to-use artifacts and tools

API design guidelines are composed of more than just descriptive text. To facilitate our designer's work, we can also expand them with

- Examples and OpenAPI templates
- Libraries of OpenAPI components
- Tool to automatically check a design (linter)

We can illustrate recipes with examples in the form of partial OpenAPI or JSON Schema snippets or links to complete OpenAPI templates. However, that may sometimes lead to duplicating elements across operations or APIs. To mitigate this, we can provide ready-to-use shared components that can be referenced from many OpenAPI documents, such as the standard `Error` data model or `UnexpectedError` response (see section 17.6). We can use a tool called a linter to check that our OpenAPI document describes a design that conforms to our guidelines automatically, so we don't always need to check our guidelines; check chapter 18 to see how to do this.

16.4.4 Helping with the API design process

To facilitate the design process, we can add meta-elements about designing APIs, such as

- API Capabilities Canvas template
- Design process description
- List of trusted sources of information
- ADR template

We can add a link to an API Capabilities Canvas template (or any other artifacts that help analyze user needs or that are related to other design methodologies). We can document the design process by adding a checklist of expected input and output artifacts. We may also describe who is responsible for what, when, and why, and how people should meet during the design of an API. To simplify research, we can have a list of trusted references, tips for research, and an ADR template with an instruction manual.

16.4.5 Adding implementation or architecture considerations

We can extend the content of our API design guidelines with considerations beyond design. For example, we can add implementation or architecture details to our recipes when relevant, such as the implications of cursor-based pagination, which may help us decide about or facilitate the work of implementation developers. However, we shouldn't add too many details unrelated to API design; we can have complete but independent API implementation guidelines and reference them from the design guidelines (and the reverse).

16.5 How to build API design guidelines

We won't build complete API design guidelines from scratch before designing our first API. It's essential to iteratively build guidelines that match our needs and carefully expand or modify them. Building API design guidelines implies that we do the following:

- Start small.
- Consider existing APIs.
- Expand the guidelines when new questions arise.
- Ensure that each rule brings value.
- Modify the guidelines with care.

16.5.1 Starting with basic API design guidelines

We won't build exhaustive guidelines covering all possible questions and edge cases from the beginning. As illustrated in figure 16.3, we can start with minimal recipes covering typical create, read, search (list), update, and delete operations and their underlying principles and rules; that will cover most of our needs. Our knowledge of API design will be helpful.

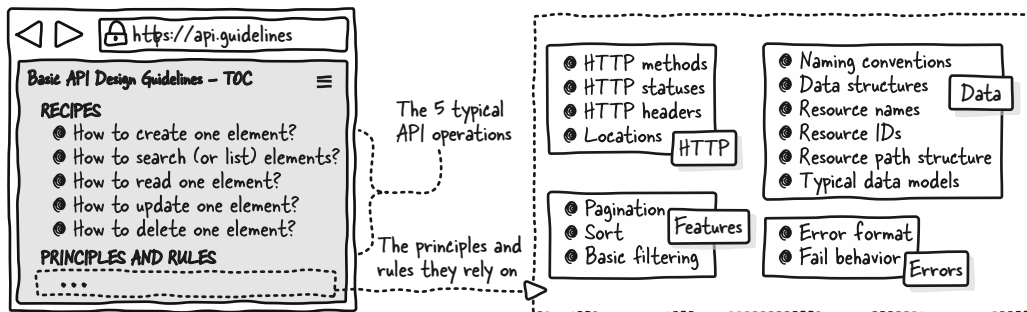


Figure 16.3 Basic guidelines cover the typical operations (create, search, read, update, and delete) and the principles and rules they rely on (covering HTTP usage, data, errors, and additional features).

We focus on basic operations like “Update one element” and “Search elements” and save less common topics such as complex filtering, bulk operations, and file uploads for later. To write guides to design typical operations, we need to consider the perspectives we’ve covered when learning to design them, such as HTTP, data, errors, and additional features, to identify rules and principles.

We need to clarify our use of HTTP. Will we allow `PATCH` or favor `PUT`? Which HTTP statuses will we use and how? Will custom HTTP headers be authorized? And what locations will we use in HTTP requests and responses?

We must decide on the data envelope for single items and lists: should we use a `data` property? Establish naming conventions such as casing for all HTTP locations or

how to name resource IDs in body data. Define typical data models (complete, summary, etc.) and their usage. Decide how to structure paths and name resources (singular or plural nouns).

We must choose our error data format, perhaps using the Problem Details for HTTP APIs standard. It's crucial to define how operations must fail (including all possible errors, for example). And specifically for the search operation, we'll need to define how we handle features such as pagination, sort, and basic filtering, including how we design them and whether we make them mandatory.

To streamline this work, we can create example designs and define principles and rules from there. Although writing guides based on these designs is possible, please don't skip the principles and rules (backed with ADRs when relevant); they are the foundational elements for all current and future guidelines and justify our design choices (see section 16.5.4).

16.5.2 Considering existing APIs

We should create guidelines for our first API, but that rarely happens; guidelines almost always come once a few or many APIs exist (which may also help us decide what we like and what works or doesn't work). When creating our guidelines, we can look at our existing APIs and start from there unless they contradict too many principles described in this book or significantly differ from what exists in the outside world (see section 16.2 to do research). Ideally, we should begin with the "cleanest" guidelines, which may imply a new API look and feel.

The consequences of modifying our API look and feel need to be carefully evaluated, which doesn't mean we can't introduce changes. If our new API design guidelines differ from what exists in our systems, our team or organization must decide how to handle existing APIs that don't comply. We can't choose to modify them all without assessing the effects (section 15.5). Architects, tech leads, managers, and the API governance team (if it exists) should collaborate to decide on a strategy. For example, it's common to have new APIs follow new guidelines; existing ones may leverage them for additions or can be entirely redesigned if they are heavily modified to introduce new, awaited features, justifying updating consumers. Having a consistent API surface will take time, and it may never happen. However, this strategy enables the introduction of a standardized API look and feel to stop the API design technical debt from increasing without breaking existing consumers.

16.5.3 Expanding the guidelines when new questions arise

Starting small implies that we won't cover every possible design question. This is not a problem; it's even a good strategy because that way, we'll only define solutions for actual problems we encounter, and we can field-test our solution. Once we go beyond the basic features listed in section 16.5.1, the context may influence the choice of solution. For instance, it is best to add file upload considerations to our guidelines only when needed, if we've never dealt with such a use case before in our architecture.

Uploading files implies much more than just designing an HTTP request; infrastructure and security are involved. We'd better wait to see the whole picture before deciding on the API design.

16.5.4 Ensuring that each rule brings value

Either in the initial version or with later expansion of the API design guidelines, it's essential to add only rules that make sense and bring value regarding consistency, efficiency, security, or interoperability. One pitfall we can fall into is defining rules for the sake of defining rules (which is typical of the "API police" mentioned in section 16.3). That's why filling out an ADR explaining each rule's good and bad consequences is essential. If we cannot explain the reason for a rule, it must not exist. For example, we should not add a rule that enforces using numbers for resource IDs. If we analyze the question of typing resource IDs as described in section 16.2, listing options with their pros and cons, we'll realize it's not a good idea (I'll let you figure out why).

16.5.5 Carefully modifying API design guidelines

Because our API design guidelines define how we design APIs, we must be mindful of how we modify them. If a modification means that all existing APIs will become invalid, we may have a problem. We most likely won't decide to drastically change how we handle pagination or replace generic 400 Bad Request with 422 Unprocessable Content just to make our design "better." We can use what we learned in chapter 15 about API modification to evolve our guidelines smoothly. We can version our guidelines to handle the introduction of breaking changes and say that some APIs conform to guidelines v1 and newer ones must conform to v2, but that would be highly complex to handle in the long run. That doesn't mean we can't version our guidelines to track their evolution; we just need to not introduce breaking changes (unless we want to).

Summary

- Ensure that it's the right time to make a design decision so you focus on the right problem. Wait for a broader vision to make an easier decision.
- Evaluate the scope of a decision to reduce decision-making pressure and spend only the necessary time on it.
- Logically explain why and how you reached a specific conclusion.
- Copy trusted decisions and solutions available in your API design guidelines, other operations of the API you design, or other internal APIs.
- Check how your question has been answered in the outside world to ensure global consistency.
- Explain your reasoning out loud to aid your thinking and help others understand the decision.
- Use sources such as your API guidelines or APIs, your usual development stack, API or data standards, and others' API design guidelines and APIs to find solutions.

- Check the source of responses when using AI to find solutions.
- Objectively consider found solutions and ensure that they are commonly used.
- Record and justify your decisions with an ADR that contains the problem to solve, the decision, options you considered, your reasoning, pros and cons, and sources of information.
- Always create API design guidelines, even when you're working alone and without fully fledged API governance.
- List generic or domain-specific principles and rules (backed by ADRs) in your API design guidelines.
- Group design rules in actionable recipes that explain how to achieve a specific design task; a rule can appear in multiple recipes. This also helps ensure that the rules work well together.
- Consider extending guideline content with OpenAPI templates, shared OpenAPI components, tools, and considerations about the design process or implementation.
- To build guidelines, start with minimal recipes that cover typical cases, such as designing create, read, search (excluding complex filtering), update, and delete operations.
- Expand the guidelines only when new questions arise so you can define solutions for actual problems and field-test your solutions.
- Only add design rules that make sense and bring value regarding consistency, efficiency, security, or interoperability.
- Carefully modify your guidelines to avoid introducing a breaking change inadvertently.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 16.1

It's your first time designing an API for your company. Is it OK to look at another of the company's APIs to decide between `GET /contracts` and `GET /contract` to search for contracts? Explain why.

Exercise 16.2

You're initiating API design guidelines. Because file upload and download can be tricky, you are considering adding recipes related to this topic. Explain why you should or shouldn't do this.

Exercise 16.3

You are working for a delivery and logistics company and designing an API to manage a fleet of vehicles. Should you add a rule to your company's API design guidelines stating that all vehicles are represented by the `/vehicles` resource?

Exercise 16.4

Your API design guidelines start with “Rule 1: Base URL contains the name and version of the API” and end with “Rule 789: A `Location` header is always returned on 201 Created.” What's the problem with such guidelines?

Exercise 16.5

How can you convince your API governance team that enforcing the use of UUIDs for all resource IDs is a terrible idea and that you should decide on a per-resource basis?

17

Optimizing an OpenAPI document

This chapter covers

- Defining consistent elements with JSON Schema and OpenAPI
- Sharing components across OpenAPI documents
- Defining OpenAPI guidelines

We may have hesitated between an integer and a string when modeling a product ID and used one or the other in various path and query parameters or properties in body data models in our OpenAPI document. We will need to update each instance of the many 500 errors if we intend to add an `Error-Id` response header after describing numerous operations.

Authoring OpenAPI documents can lead to cumbersome information duplication and, more worryingly, API design inconsistency. Additionally, unoptimized OpenAPI documents that don't use all of the format's possibilities are more complex to author and maintain, leading to more inconsistencies. By using OpenAPI and JSON Schema extensively, we can optimize our OpenAPI documents to reduce the risk of inconsistencies and facilitate our work. That must be done separately from the first pass through the design so we don't mix design- and OpenAPI-related discussions. However, with experience, we'll be able to introduce optimization, speed up authoring, and ensure consistency from the start.

This chapter begins with an overview of OpenAPI document optimization. We then explain how to use OpenAPI and JSON Schema to describe consistent schemas, parameters, request bodies, response bodies, and headers. Finally, we discuss creating OpenAPI libraries and OpenAPI authoring guidelines.

17.1 An overview of OpenAPI document optimization

As illustrated in figure 17.1, we’re back to the “Describe the programming interface” stage of the design process to discuss OpenAPI optimizations. While representing operations with HTTP (section 4.1) and modeling data (section 5.1), we used OpenAPI (section 6.1) and JSON Schema (section 7.1) to describe our design. We set aside the final optimization step to learn about the importance of consistency when working on user-friendly design and how API design guidelines (section 16.3) can contribute to it. We can now learn how to optimize an OpenAPI document to do the following:

- Ensure API design consistency
- Simplify authoring and maintaining the OpenAPI document

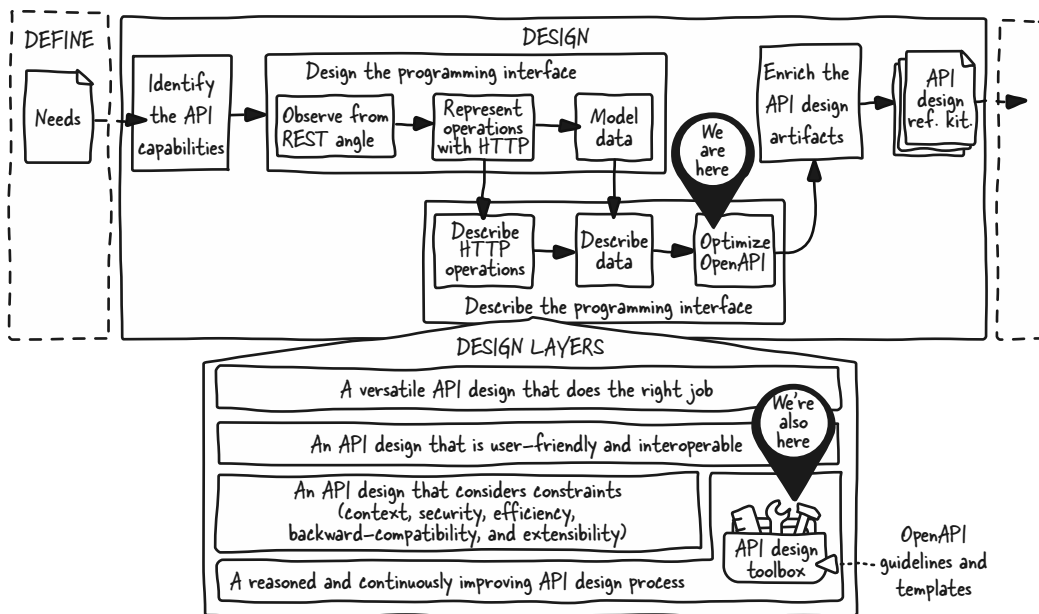


Figure 17.1 We’re back to the “Describe the programming interface” stage of the design process. We must optimize our OpenAPI document to ensure consistency and simplify its evolution. To avoid mixing design and OpenAPI concerns, we should optimize the OpenAPI document after the “Design the programming interface” step.

We’ll use OpenAPI features to define consistent schemas, parameters, bodies, and responses. For instance, the 404 Not Found response will be defined once for all `/products/{productId}` operations. Additionally, we’ll ensure cross-API consistency

by creating an OpenAPI library. This will allow us to share elements, such as standard pagination parameters, across OpenAPI documents and, thus, APIs.

We have already slightly optimized our OpenAPI document with reusable schemas when describing data, simplifying our work. But we have barely scratched the optimization surface; we'll learn many new techniques using the online shopping example. Once you gain experience, you can seamlessly use some of these techniques during actual API design, speeding up authoring. However, in the beginning, or for complex techniques, it's better to use a separate optimization step: put essential information in the OpenAPI document, and then see how it can be optimized. This approach will ensure that the HTTP and data modeling steps are not polluted with OpenAPI-specific concerns. Additionally, to facilitate authoring and ensure consistent use of OpenAPI and these techniques, you can consider expanding the meta-information of your API design guidelines with OpenAPI-related concerns.

17.2 Defining consistent data models

When we described data with JSON Schema in our OpenAPI document for the Shopping API in section 7.7.1, we learned to share data models between operations or create complex models using reusable models and references. That helps to easily create consistent data models and facilitates maintaining our OpenAPI document. We can use other techniques to help with this; these techniques apply to all data, regardless of its final use (parameter, request body, response body, or response headers). This section starts by reminding us of what we've learned about reusable schema. Then we show how to use deep references, override a reusable schema description, and create unique read-and-write models. Finally, we look at how to aggregate data models and carefully consider this practice.

17.2.1 Reusing schemas

We learned to define reusable schemas under `components.schemas` in section 7.4 and use them with a `$ref` JSON pointer reference in section 7.7.1.

Listing 17.1 Defining and referencing a schema

```
paths:
  /products:
    get:
      responses:
        "200":
          content:
            application/json:
              schema:
                type: object
                properties:
                  data:
                    type: array
                    items:
                      $ref: "#/components/schemas/ProductSummary"
```

Reference to the
ProductSummary
schema



```

components:
  schemas:
    SupplierSummary:
      ...
    Product:
      ...
      properties:
        supplier:
          $ref: "#/components/schemas/SupplierSummary"
    ProductSummary:
      ...
      properties:
        supplier:
          $ref: "#/components/schemas/SupplierSummary"

```

← Identifier of a reusable schema

← References to the SupplierSummary schema

I recommend defining all resource data models (in all their versions, such as complete and summary) as reusable schemas. This keeps all of our data models organized in one place for readability and reuse and allows us to model them before using them. The “Product” resource’s complete and summary data models are defined under `components.schemas` as `Product` and `ProductSummary` JSON schemas.

We can reference any reusable schema with a `$ref` JSON pointer anywhere in any schema, including parameters, request bodies, response bodies, response headers, and all schemas under `components.schemas`. The `ProductSummary` model is referenced with a `$ref` JSON pointer set to `#/components/schemas/ProductSummary` under the 200 response of the GET `/products` operation. It’s the schema of the data array items.

17.2.2 Defining subschemas

In listing 17.1, the `SupplierSummary` schema is referenced in both the `Product` and `ProductSummary` schemas. Defining a subdata model shared by resource models under `components.schemas` is recommended to foster consistency and facilitate further modification. But we can wait for the OpenAPI optimization step to perform this optimization if we’re unsure. The drawback of this technique is that the reusable-schemas section may be bloated with many utility schemas. Sorting them by resource schemas first and then utility schemas may help limit the annoyance; we can place schemas in any order in the document. We can also consider using the deep reference described in section 17.2.4.

17.2.3 Targeting part of a schema with a deep reference

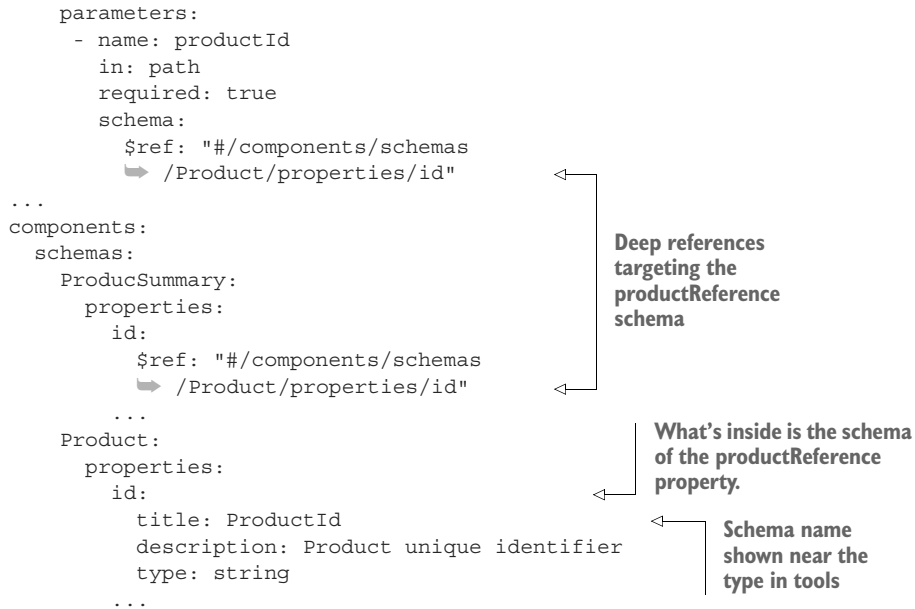
When an element is a copy of another sub-element of a schema, we can use a *deep reference* that targets a subpart of a schema instead of creating a small reusable schema.

Listing 17.2 Referencing an inner schema

```

...
paths:
  /products/{productId}:

```



The `Product` and `ProductSummary` models' `id` property and `productId` path parameter are all product IDs. We could define a `ProductId` reusable schema under `components.schemas` and reference it in these three cases with `$ref: "#/components/schemas/ProductId"`. However, in that case, we would also have an `OrderId` and many other `ResourceId` micro schemas.

Instead, we can define a complete schema under the `id` property of the `Product` schema, the source of truth for all product-related data; all other product-related schemas, including the path parameter, are derived from it (see section 5.4). Then we can reference the `Product`'s `id` property schema when defining the `id` property of the `ProductSummary` schema and the `productId` path parameter schema with the `#/components/Product/properties/id` JSON Pointer. The `Product`'s `id` schema's optional `title` property will be shown in OpenAPI tools, usually next to the type: `string (ProductId)`, for example.

This deep-reference technique works for any schema, not only atomic ones. If it doesn't make sense for the `SupplierSummary` object schema from section 17.2.1 to exist independently, we can define it under the `supplier` property of the `Product` schema (with `title` set to `SupplierSummary`) and use it with the `#/components/Product/properties/supplier` reference.

17.2.4 Overriding descriptions when using a \$ref

When we reference schemas with a `$ref`, especially generic ones, it can be helpful to override, and hence replace, their original description to contextualize their use. We just need to add a `description` near `$ref`, as illustrated in listing 17.3.



TIP OpenAPI allows the override of the summary (if it exists) and description of any element targeted with a `$ref`.

Listing 17.3 Overriding a referenced schema description

```
components:
  Amount:
    description: An amount of money
    type: number
  Product:
    properties:
      price:
        description: The product's price in USD
        $ref: "#/components/schemas/Amount"
  Order:
    properties:
      total:
        description: The total amount
        $ref: "#/components/schemas/Amount"
```

Overrides the description of the following referenced component

Reference to a reusable component

For example, a generic `Amount` schema is useful in our Shopping API; we can use it any time we need to represent an amount of money, such as a product price or an order total. However, the referenced schema description (*An amount of money*) will be shown when rendering the OpenAPI document. To override the referenced schema description, we set a `description` property with the desired value next to the `$ref` reference (*The total amount*, for example).



CAUTION Overriding the `$ref` description is unavailable in OpenAPI 3.0; tools will ignore a `description` property near a `$ref` or show a parsing error. It's possible to emulate a similar behavior using the `allOf` JSON Schema keyword (see section 17.2.6) to merge a schema containing a `$ref` and another with the `description`. But I do not recommend doing this as it makes the documentation and resulting API documentation unnecessarily complex to read. It's better to switch to a more recent version of OpenAPI.

17.2.5 Creating unique read-and-write models

The input and output schemas of create and update operations are usually similar, although their fields may differ slightly. Initially, we learned to duplicate complete schemas to define “create” or “update” schemas, which may lead to inconsistency if the design is modified later. Instead, we can create a single read-and-write schema by using the `readOnly` and `writeOnly` JSON Schema flags. We can use such a schema as the output for create, read, and replace operations and as the input for create and replace operations.

In section 7.7.2, we duplicated the complete `Product` resource schema to create the `ProductCreationOrReplacement` schema by removing server-defined properties. The deep-reference trick from section 17.2.3 could limit duplication, but using the `readOnly` and `writeOnly` JSON Schema flags prevents duplication, creating a unique read-and-write `Product` schema.

Listing 17.4 Defining a unique read-and-write model

```

...
components:
  schemas:
    Product:
      properties:
        id:
          readOnly: true
          type: string
        price:
          type: number
        type:
          properties:
            code:
              writeOnly: true
              type: string
            name:
              readOnly: true
              type: string
      ...

```

Will be used only in requests

Will be used only in responses

The `id` property is handled by the server, so we set its `readOnly` flag to `true`. That means it exists only in a response (of a read, create, or update product operation), not in a request (of a create or update product operation). For demonstration purposes only, we've set the `writeOnly` flag of the `type.code` property to `true`. This means the property exists only when the schema is used in a request; the effect is the opposite of the `readOnly` flag. Consumers provide it when creating or modifying a product. The `writeOnly` flag is usually necessary when there's a structural difference between input and output during creation or update. However, as advised in section 9.10.1, it's best to avoid introducing structural variations between output and input so consumers can seamlessly take the output from the read operation, modify it, and then use it as input for the update operation.

The `Product` schema can be used as input for create or update product operations and as output for read, create, or update product operations with the `#/components/schemas/Product` reference. The following listing illustrates its usage in the 200 response of `GET /products/{productId}` and the request body of `PUT /products/{productId}`.

Listing 17.5 Using the same model in the request and response

```

paths:
  /products/{ProductId}:
    ...
    get:
      ...
      responses:
        "200":
          content:
            application/json:

```

```

        schema:
          $ref: "#/components/schemas/Product"
    put:
      ...
      requestBody:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Product"
      ...

```

Same schema
used in the
request and
response

17.2.6 Defining a complete schema from its summary

A summary schema is a subset of a complete schema; we create them by copying and pasting elements, risking inconsistency with future changes. To avoid duplication and inconsistency, we can define a complete schema by combining a summary and other data using JSON Schema's `allOf` keyword.

In section 5.4.1, we created the `ProductSummary` by copying and pasting elements from the `Product` schema; listing 17.6 shows how we can clean this up. We keep the `ProductSummary` schema as we originally defined it by picking elements from the complete schema. We strip from the `Product` schema all properties defined in its summarized version. Then we use the remaining elements as the second item in an `allOf` list. The first item is a reference (`$ref`) to the `ProductSummary` schema. That means the `Product` schema is the sum of the summarized schema (first item) and the inline schema defined in the second item.

Listing 17.6 Merging schemas to create a complete schema

```

components:
  schemas:
    ProductSummary:      ← Summary schema
      required: ...
      properties:
        productReference: ...
      ...
    Product:             ← Complete schema

    allOf:               ← Merges a list of schemas
      - $ref: "#/components/schemas/ProductSummary"
      - required: ...
        properties: ...

```

Reference to
the summary

Schema defining all
other properties

This feature is fully compatible with the read-and-write model technique. We can set the `readOnly` and `writeOnly` flags where required on the summarized model (even if we use it only on read operations) or in the second inline model in the `allOf` list.

17.2.7 Considering schema optimizations

We've learned about several features and techniques to help us avoid duplicating information in an OpenAPI document. However, in some cases you may feel the resulting optimized document is too complex, especially if it uses the JSON Schema `allOf` keyword heavily. It's essential to know that this can happen and not to worry too much if it does; a document with a few duplications may be easier to maintain.



NOTE We focus on JSON Schema's most common possibilities; refer to the documentation for additional keywords at www.learnjsonschema.com. Besides `allOf`, you may be interested in polymorphism-related keywords such as `anyOf` and `oneOf`. However, a schema may become complex by combining these keywords. We'll also see in section 18.9.4 that some JSON Schema keywords may not be compatible with code generation.

17.3 Defining consistent parameters

We learned to define path parameters (`/resources/{resourceId}`) using the path-level `parameters` list in section 6.4.3. Doing so ensures that all operations under a path share the exact same definition. It's worth noting that we can use the path-level `parameters` list to define other types of parameters consistently across a path's operations. However there are more ways to ensure request parameter consistency in our OpenAPI document. This section reminds us how to use the path-level parameter list, how to define reusable parameters, and how to create reusable groups of parameters.

17.3.1 Using path-level parameters

We must always define path parameters in the path-level `parameters` list to avoid duplicating information at the operation level. That's a no-brainer optimization we can use from the start when describing the resource paths (section 6.4.2). However, the path-level `parameters` list is not reserved for path parameters. We can use it for any parameter that all operations share under a path regardless of its location (`in`), such as query or header. However, given that we decided to use query parameters as resource modifiers, such as search filters (section 4.4.1), we likely won't define query parameters at this level because they can't apply to all operations under a path. The following listing defines a `Version` request header that indicates the API version (although I usually recommend using path versioning; see section 15.4.4).

Listing 17.7 Defining path-level parameters

```
...
paths:
  ...
  /products/{productId}:
    parameters:
      - name: productId
        in: path
        required: true
```

Always define path parameters at the path level.



```

    schema: ...
  - name: Version
    in: header
    required: true
    schema:
      type: string
      const: "2"
get:
  ...
put:
  ...
delete:
  ...

```

← Request header

← All path operations need the productId and Version parameters.

The Version request header is defined as usual; we set its location (`in: header`), indicate whether it's required, and define its schema (note the use of the JSON Schema `const` keyword, which indicates the only possible value, similar to an `enum` with a single value). Using a Version header implies defining it on all paths. Unfortunately, there's no way to define parameters for all operations in OpenAPI 3. We could define a reusable schema so that all these Version parameters share the same schema. But the next section will show us how to define a parameter once and for all, ensuring consistency across all paths that need it.

17.3.2 Reusing parameters

We defined reusable JSON Schemas under `components.schemas`, but the `components` block can hold other reusable elements, such as parameters. They are defined and used similarly to reusable schemas; we define reusable parameters under `components.parameters` (listing 17.8) and use them with a `$ref` whose value is a JSON pointer targeting the parameter (listing 17.9). As is the case with schemas, we can override the targeted parameter description.

Listing 17.8 Defining a reusable parameter

```

...
components:
  parameters:
    ProductId:
      name: productId
      in: path
      required: true
      schema: ...
...

```

Where to define reusable parameters

Reusable parameter identifier

Same elements as for an item in the parameters list

Listing 17.9 Referencing a reusable parameter

```

...
paths:
  ...
  /products/{productId}:

```

```

parameters:
  - $ref: "#/components/parameters
    ➡ /ProductId"
...
/products/{productId}/suppliers:
  parameters:
    - description: Same as in Read Product
      $ref: "#/components/parameters
        ➡ /ProductId"
...

```

Reference (JSON pointer) targeting a reusable parameter

Overrides the targeted parameter's description

We define the `productId` path parameter from section 17.2.3 under `components.parameters` under the `ProductId` key. Then we can reference it with the `#/components/parameters/ProductId` JSON pointer on different paths; remember the dash (-) before the references. For demonstration purposes, we override the description for the second reference (this specific description isn't necessary in that case). This is not reserved for path parameters, and we can use this feature for any other parameter type at the path or operation level. For instance, we can use this technique to define the `Version` request header from section 17.3.1 or generic pagination query parameters shared across all search operations (see section 17.3.3).

17.3.3 Defining reusable groups of query parameters

The reusable parameters we saw in section 17.3.2 may fall short when we'd like to reuse a group of query parameters: for example, for pagination. Doing so requires defining and referencing parameters one by one; it's cumbersome, and we risk introducing inconsistency in our operations by forgetting some or redefining them differently. Fortunately, the *OpenAPI Parameter object* proposes a default serialization mechanism that turns each property of an object query parameter into multiple query parameters. We can use this feature to create a group of query parameters (listing 17.10) and use it with a single reference (listing 17.11).

Listing 17.10 Creating a group of query parameters

```

components:
  parameters:
    Pagination:
      name: pagination
      description: |
        Pagination parameters (`.../?next=s00999&limit=10`)
      in: query
      schema:
        type: object
        properties:
          next:
            description: Next page cursor
            type: string
          limit:
            description: Number of elements per page
            type: number

```

Just shown in the documentation, not used in the request

Clarifies how this parameter works

Sets type to object to create a group

Each property is serialized as a query parameter.

Listing 17.11 Using a query parameter group.

```

paths:
  /products:
    get:
      parameters:
        - $ref: "#/components/parameters/Pagination"

```

Tools will show the next and limit query parameters.

When using cursor-based pagination (section 13.6.2), we typically need query parameters like `next` (next page cursor) and `limit` (page size). We could define separate `PaginationNext` and `PaginationLimit` query parameters under `components.parameters` and then reference them in all search operations. However, a simpler and more efficient solution is to define a single reusable `Pagination` parameter and describe its schema as an object with `next` and `limit` properties.

Once the parameter is defined, we can reference it like any other, using the `#/components/parameters/Pagination` JSON pointer. Using the query parameter default serialization, tools will interpret this single parameter as two `next` and `limit` query parameters. Although powerful, this technique is rarely seen, and the rendering in tools may be unclear because of their design or because they don't support all OpenAPI features. I recommend adding a description to the `Pagination` parameter clarifying how to use it and explicitly showing that `name: pagination` is not used in the request (`.../?next=s00999&limit=10`).



NOTE Parameter serialization doesn't allow the definition of multiple headers because it's the content of the parameter that is serialized. For a query parameter, the serialization mechanism turns the `{ "a": 1, "b": 2 }` object into the `a=1&b=2` string. In the case of a header, it gives the `a,1,b,2` string by default and does not lead to two `a` and `b` headers. Parameter serialization can be tuned with `style` and `explode` parameter fields; check out the OpenAPI documentation at <https://spec.openapis.org/oas/v3.1.0#parameter-object>.

17.4 Defining consistent request bodies

Different operations—typically create and update (replace) operations—may share the same request body. We could reference the same schema to ensure consistency, but a better approach is to define a reusable request body under `components.requestBodies` (listing 17.12) and reference it with a `$ref` (listing 17.13). Additionally, we can use the description override trick from section 17.2.4.

Listing 17.12 Defining a reusable request body

```

...
components:
  requestBodies:
    ProductCreateOrReplace:
      description: Product info
      content:
        application/json:
          schema: ...

```

Where to define reusable request bodies

Reusable request body identifier

Same elements as in an operation's requestBody property

Listing 17.13 Referencing a reusable request body

```

...
paths:
  /products:
    post:
      requestBody:
        $ref: "#/components/requestBodies/ProductCreateOrReplace"
  /products/{productId}:
    ...
    put:
      requestBody:
        description: |
          Extra properties are ignored. The complete
          Product data returned by Read product is
          accepted.
        $ref: "#/components/requestBodies/ProductCreateOrReplace"

```

Reference (JSON pointer) targeting a reusable request body

Overrides the referenced request body description

The “Create product” and “Update product” operations must share the same `Product` read-and-write schema from section 17.2.5 in their request bodies. However, instead of targeting it in the schema of each operation’s `requestBody`, we define a unique and reusable `ProductCreateOrReplace` request body under `components.requestBodies`; it contains the same information we would have put in each operation.

Then we can reference it with the `#/components/requestBodies/ProductCreateOrReplace` JSON pointer under the `requestBody` of each operation. We can override the reusable request body’s description by adding a `description` next to the `$ref` to state, for example, that the “Update” operation accepts extra properties; a consumer can send a request body containing the complete `Product` data that the “Read” operation returns (see section 9.8.1).

17.5 Defining consistent responses

Responses of different operations may share similar schemas, response headers, or descriptions. We can use reusable schemas, response headers, and responses to ensure consistency. We already covered reusable schemas in section 17.2.1; this section focuses on reusable responses and response headers.

17.5.1 Reusing response headers

Responses can share identical headers. Defining reusable response headers is similar (how surprisingly consistent!) to how we define schemas, parameters, and responses. We define headers under `components.headers` (listing 17.14) and use them with the appropriate `$ref` (listing 17.15). Unlike parameters, we can’t define reusable response header names. A response’s headers are defined in a `headers` object, and its keys are the header names (unfortunate inconsistency!).

Listing 17.14 Defining a reusable response header

```

components:
  headers:
    ResourceLocation:
      description: Created resource URL
      schema: ...

```

Where to define reusable response headers

Reusable header identifier (not its name)

Listing 17.15 Referencing a reusable response header.

```

...
/products:
  post:
    ...
    responses:
      "201":
        headers:
          Location:
            description: The created product's URL
            $ref: "#/components/headers/ResourceLocation"
    ...

```

The response header name must be indicated.

Overrides the reusable response header's description

Reference to the reusable response header

When creating a resource with `POST /resources`, we return a `Location` header along with the 201 Created response (section 4.6.2); we can define it under `components.headers` with the `ResourceLocation` identifier. For teaching purposes, I do not use `Location` so we can differentiate the actual header name from the reusable header identifier.

Then we can reference the `ResourceLocation` reusable header under `Location` with the `#/components/headers/ResourceLocation`. As we've seen for schemas, parameters, and responses, we override the targeted reusable header's description by adding a description field near `$ref`.



NOTE Only response headers are defined under `components.headers`; for request headers, define a reusable parameter with `in` set to `header` under `components.parameters`. Unlike reusable response headers, reusable request headers have their names defined; it is a known problem that a future version of OpenAPI will fix. As you can see, like APIs, the design of a format such as OpenAPI can be inconsistent and have room for improvement; what you learn in this book can help you create or contribute to such a format.

17.5.2 Reusing responses

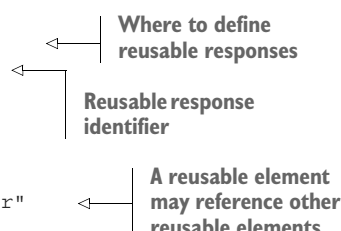
Operations can share identical responses (the same content schemas and headers); for example, all 500 errors and the responses to reading and updating a resource are identical. As is the case with other reusable components, we can define responses under `components.responses` (listing 17.16) and use them with the appropriate `$ref` (listing 17.17). We can't define the HTTP status code of a reusable response because it is a key under an operation's `responses` object.

Listing 17.16 Defining a reusable response

```

components:
  responses:
    ResourceNotFound:
      description: No resource was found
      content:
        'application/json':
          schema:
            $ref: "#/components/schemas/Error"

```



Where to define reusable responses

Reusable response identifier

A reusable element may reference other reusable elements.

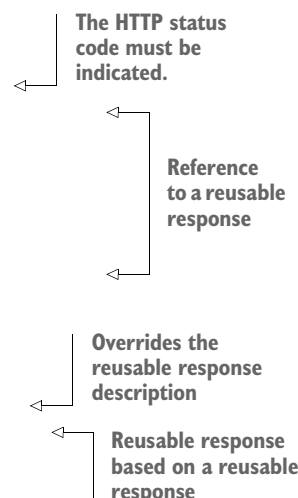
We can define a `ResourceNotFound` reusable response whose content is similar to what we would have put in the `404` key of the responses of any operations under a path containing a path parameter. The `schema` points to our generic `Error` schema shared by all of our errors.

Listing 17.17 Referencing a reusable response

```

...
/products/{productReference}:
  ...
  get:
    ...
    responses:
      ...
      "404":
        $ref: "#/components/responses/ProductNotFound"
  put:
    ...
    responses:
      ...
      "404":
        $ref: "#/components/responses/ProductNotFound"
components:
  responses:
    ...
    ProductNotFound:
      description: No product was found
      $ref: "#/components/responses/ResourceNotFound"

```



The HTTP status code must be indicated.

Reference to a reusable response

Overrides the reusable response description

Reusable response based on a reusable response

Then we can reference this reusable response with the `#/components/responses/ResourceNotFound` JSON pointer. We could use it directly under the `404` response of “Read product” and “Update product” operations. But because we would like to override the description in both, we define a reusable `ProductNotFound` response containing a `description` field near `$ref` with the value `No product was found` and then reference this product-specific response in these two operations: `responses.404`. Similarly, we can create a `ProductSuccess` response referencing the `Product` schema to be returned by the “Read product” and “Update product” operations.

17.6 Ensuring cross-API consistency with external shared components

Some elements, such as generic schemas and responses, will be the same across APIs, which makes them look similar and favors interoperability. To ensure consistency and simplify our OpenAPI document, we can put them in one or multiple *libraries*. As discussed in section 16.4.3, such artifacts can be part of our API design guidelines (also contributing to API governance, if that exists in our organization; see section 16.3.2). This section shows how to define OpenAPI libraries and use the components they contain. Finally, we stress the necessity of ensuring that library files are independent.

17.6.1 Defining a library of reusable components

Listing 17.18 shows an OpenAPI library defining components we can reuse across APIs. We can use all the elements under `components` except `securitySchemes`: security schemes are used not via `$ref` but rather by name (a design limitation that should be fixed in a future major version of OpenAPI).

Listing 17.18 Creating an OpenAPI library

```
openapi: 3.1.0
info:
  title: Shared components
  version: "1.0"

components:
  schemas:
    Errors: ...
  parameters:
    Pagination: ...
  headers:
    ResourceLocation: ...
  responses:
    ResourceNotFound: ...
```

Version of this OpenAPI file

No paths or webhooks are defined.

Components that can be used across APIs

The `shared-components.yaml` file is an OpenAPI file that contains `info` and `components` blocks but doesn't define an actual API with `paths` or `webhooks`. The `info.version` field holds the version of this library, which is independent of the version of any API using it. Under `components`, the library defines the `Errors` schema (in `schemas`), `Pagination` parameter (in `parameters`), `ResourceLocation` response headers (in `headers`), and `ResourceNotFound` response (in `responses`).



CAUTION Be mindful of not inadvertently introducing breaking changes in the components of OpenAPI libraries; all APIs that use these components will be affected.

17.6.2 Using a shared component in an API

Using a component defined in an OpenAPI library is similar to using a component defined in an OpenAPI file; the difference is that the `$ref` value is the concatenation of the absolute or relative file path or URL and the JSON pointer of the element within the file. Relative URLs and paths are calculated from the location of the OpenAPI file targeting the components. As shown in listing 17.19, the `Pagination` parameter defined under `components.parameters` of the `shared-components.yaml` file located in a `/path/to` folder can be referenced with `$ref: "/path/to/shared-components.yaml#/components/parameters/Pagination"`. A `"shared-components.yaml#/components/parameters/Pagination"` relative reference would mean both files are located in the same place.

Listing 17.19 Referencing components from the library

```
openapi: 3.1.0
info:
  title: Shopping
  version: "2.3"

paths:
  /products:
    get:
      parameters:
        - $ref: "/path/to/shared-components.yaml
          ➔ #/components/parameters/Pagination"
  ...
```

Relative or absolute path or URL

JSON pointer in the file



TIP Some tools, such as code generators, may not handle split OpenAPI files. Search for tools that bundle, combine, or merge OpenAPI files. My favorite is Redocly CLI (<https://github.com/Redocly/redocly-cli>).

17.6.3 Ensure that library files are editable independently

We can organize our shared components in multiple files. However, each library file must be a valid OpenAPI or JSON Schema file that can be edited independently.

Don't define shared components in partial OpenAPI files, as in listing 17.20. This `pagination-parameter.yaml` file contains the definition of the `Pagination` parameter from section 17.3.3. Although it's possible to reference the parameter it defines with `$ref: "/path/to/pagination-parameter.yaml"` (no JSON pointer), this file isn't editable independently; an OpenAPI parser can't validate it in a standalone way.

Listing 17.20 A partial OpenAPI file

```
name: pagination
description: |
  Pagination parameters (`.../?next=s00999&limit=10`)
in: query
schema: ...
```

Defining shared schemas in standalone JSON Schema files is OK; they can be edited and validated independently. If we define an `errors-schema.json` file containing the JSON schema of our `Errors` model, we can reference it with `$ref: "/path/to/errors-schema.json"` (no JSON pointer). However, be careful about tooling; although there are UI OpenAPI editors that facilitate editing JSON schemas, they may not work with pure JSON Schema files, and I haven't seen dedicated JSON Schema editors that provide a UI matching that available in good OpenAPI UI editors. Additionally, a JSON Schema file should normally be a JSON file, not YAML. Although it's not a problem to have an OpenAPI YAML file referencing a JSON Schema file in JSON, remember why we chose YAML instead of JSON: it's easier to edit (see section 6.2.5).

17.7 *Enhancing API design guidelines*

As you can see, we can use many features, patterns, and tricks to foster consistency and facilitate authoring our OpenAPI document, and it can be complicated to remember everything. As discussed in section 16.4, we can enhance our guidelines with OpenAPI-related concerns. We must indicate the location of the OpenAPI libraries and when and how to use them. We can also add general guidance about OpenAPI authoring so we can edit our OpenAPI documents consistently. For example, we can always use a deep reference when targeting a resource ID schema and not define a micro-schema. Section 18.6.4 will show us how to automatically check that we properly define and use our OpenAPI libraries and consistently author our OpenAPI documents. This OpenAPI-related information and artifacts significantly contribute to better API governance (if that exists in your organization).

Summary

- Do not pollute HTTP and data modeling with complex OpenAPI-specific concerns. Put essential information in the OpenAPI document, and see if it can be better optimized afterward.
- Define reusable schemas under `components.schemas` to ensure consistency, and use them with a `$ref` JSON pointer targeting `#/components/schemas/SchemaId`.
- Override any reusable component's description by placing a `description` near `$ref`.
- Target inner schemas with deep references to avoid creating many small schemas.
- Create unique read-and-write schemas by using `readOnly` flags. Using `writeOnly` flags may introduce risky inconsistency between the request and response.
- Use `allOf` to define a complete schema by merging a summary schema with an inline schema containing other properties.
- Find a balance between optimization and the complexity it can create.
- Always define parameters that apply to all operations of a path at the path level to prevent duplication.

- Define reusable parameters under `components.parameters` to ensure consistency, and use them with a `$ref` JSON pointer targeting `#/components/parameters/ParameterId`.
- Define a reusable object query parameter to create a reusable group of query parameters.
- Define reusable request bodies under `components.requestBodies` to ensure consistency, and use them with a `$ref` JSON pointer targeting `#/components/requestBodies/RequestBodyId`.
- Define reusable response headers under `components.headers` to ensure consistency, and use them with a `$ref` JSON pointer targeting `#/components/headers/HeaderId`.
- Define reusable responses under `components.responses` to ensure consistency, and use them with a `$ref` JSON pointer targeting `#/components/responses/ResponseId`.
- Define libraries of components in valid and editable OpenAPI or JSON Schema files.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix. I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 17.1

How can you optimize the OpenAPI document in listing 17.21?

Listing 17.21 OpenAPI document

```
openapi: 3.1.0
info: ...
paths:
  /authors:
    get:
      responses:
        "200":
          description: Found authors
          content: ...
        "401":
          description: Unauthorized
          content: ...
        "500":
          description: Server error
          content: ...
  /books:
    get:
      responses:
```

```

"200":
  description: Found books
  content: ...
"401":
  description: Invalid token
  content: ...
"500":
  description: Unexpected error
  content: ...

```

Exercise 17.2

How can you optimize the OpenAPI document in listing 17.22?

Listing 17.22 OpenAPI document

```

openapi: 3.1.0
info: ...
paths:
  /authors/{authorId}:
    get:
      parameters:
        - name: authorId
          in: path
          required: true
          schema:
            type: string
      responses:
        "200":
          description: An author
          content:
            application/json:
              schema:
                properties:
                  id:
                    type: string
    delete:
      parameters:
        - name: authorId
          in: path
          required: true
          schema:
            type: string
      responses:
        "204":
          description: Author deleted.

```

Exercise 17.3

How can you optimize the OpenAPI document in listing 17.23?

Listing 17.23 OpenAPI document

```

openapi: 3.1.0
info: ...

```

```

paths:
  /authors:
    post:
      requestBody:
        content:
          application/json:
            schema:
              properties:
                name:
                  type: string
      responses:
        "201":
          description: Author created.
          content:
            application/json:
              schema:
                properties:
                  id:
                    type: string
                  name:
                    type: string

```

Exercise 17.4

How can you optimize the OpenAPI document in listing 17.24?

Listing 17.24 OpenAPI document

```

openapi: 3.1.0
...
components:
  schemas:
    AuthorSummary:
      properties:
        id:
          type: string
        name:
          type: string
    Author:
      properties:
        id:
          type: string
          readOnly: true
        name:
          type: string
        genres:
          type: array
          items:
            type: string

```

18

Automating API design guidelines

This chapter covers

- Detecting API design problems or improvements with a program
- Deciding what to check in API designs
- Typical elements to target and checks to perform
- Returning helpful feedback
- Using and tweaking shared automated guidelines

Our API design guidelines indicate that all property names should be in camel case. However, it's common to make typos (using `Account` instead of `account`) or be unsure about camel-case acronyms (`sourceIBAN` or `sourceIban`). We also may forget to add pagination to a search operation or be uncertain about the exact names of the pagination query parameters (`cursor` versus `next` and `pageSize` versus `limit`) and whether they are required or optional.

Keeping track of all our API design guidelines' rules can be challenging, even with helpful recipes. Consistently checking the guidelines may become frustrating and slow us down. To mitigate this, we can use a common coding practice: *linting*. This consists of analyzing source code for errors or style problems with a program called a *linter*. Here, we analyze or lint an OpenAPI document describing our API,

which we refer to as *API linting*. This practice can seamlessly guide us and help ensure that our API incorporates the design patterns chosen to foster consistency, user-friendliness, security, efficiency, and extensibility. API linting is a must-have companion to our guidelines; it takes care of details and allows us to focus on whether our design effectively meets consumers' needs.

This chapter discusses the purpose of API linting and the benefits of automating API guidelines with an API linter. It gives an overview of guidelines automation to help us choose a linter. The chapter introduces the Spectral API linter and uses it to illustrate the implementation of guidelines and the use of the result when designing APIs.

18.1 What API linting is and how it can help us

As shown in figure 18.1, we're adding API linting to our toolbox. A linter is a tool that identifies programming errors, bugs, stylistic problems, and suspicious constructs in code. This analysis can help prevent efficiency problems or make code easier to maintain. API linting applies this to an API definition, such as an OpenAPI document. Using a linter when designing APIs frees our minds of details and seamlessly fosters consistency, user-friendliness, security, efficiency, and extensibility. With an API linter, we can

- Detect API design problems.
- Detect OpenAPI authoring problems.
- Apply our API design guidelines seamlessly.
- Concentrate on designing an API that meets user needs.

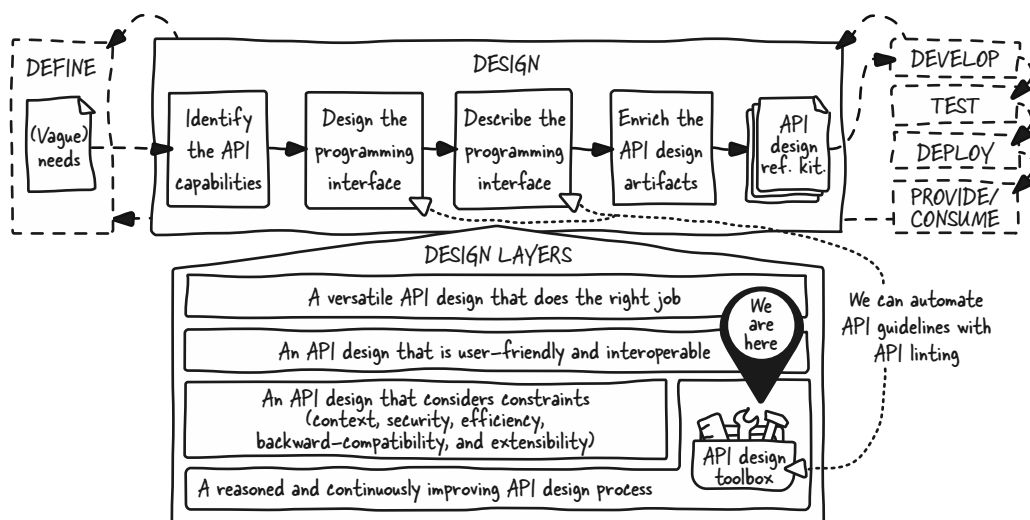


Figure 18.1 We can use an API linter to apply our guidelines seamlessly. Linting can help us with API design and OpenAPI authoring.

18.1.1 Detecting API design and OpenAPI authoring problems

An API linter can detect API design problems, ensuring consistent, user-friendly, efficient, secure, and future-proof API designs. It can also detect OpenAPI authoring problems, guaranteeing clear and maintainable OpenAPI documents. Figure 18.2 shows how an IDE can lint an OpenAPI document describing the API we design. This is similar to when an IDE detects Java or Python code problems, for example.

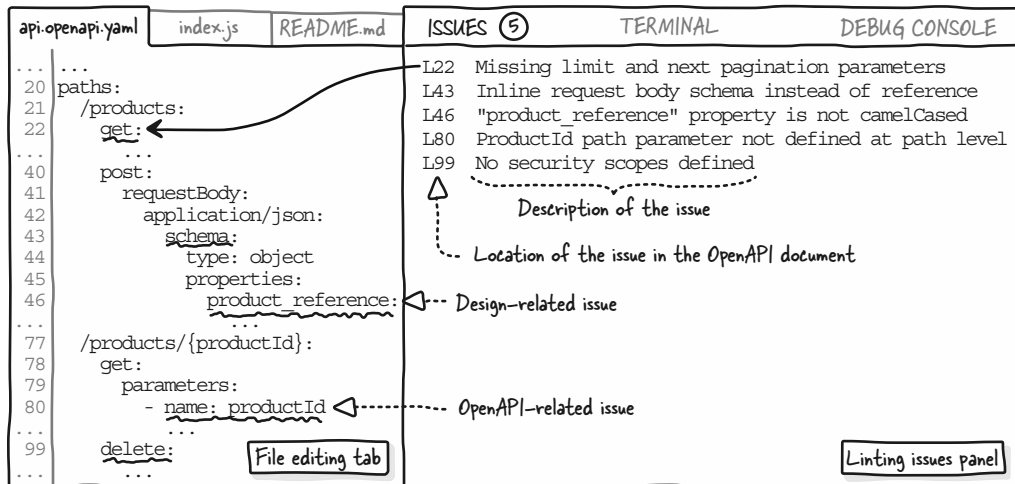


Figure 18.2 An API linter can detect API design and OpenAPI-related problems and help fix them with location indication and description.

The IDE's API linter detected several problems in the OpenAPI file `api.openapi.yaml`. The IDE highlights problems and provides descriptions to help address them quickly. These include design-related problems such as inconsistent property naming (affecting consistency and user-friendliness), missing pagination parameters for a search operation (affecting user-friendliness and efficiency), and undefined scopes (affecting security). Additionally, OpenAPI-related problems were found, such as unnecessary duplication of path parameters and inline definition of request body schema, affecting document clarity.



NOTE We can lint an OpenAPI document in an IDE, OpenAPI UI editor, or terminal with a command-line interface (CLI). An API linter can be integrated into continuous integration and continuous delivery (CI/CD) pipelines and doesn't care how the OpenAPI file was created (see section 6.2). API linting is useful during design and development to prevent introducing avoidable errors that require breaking changes in order to be fixed (renaming a property to fix a casing problem, for example).

18.1.2 Applying guidelines seamlessly and concentrating on user needs

Although API linting could exist without API design guidelines and guidelines could exist without linting, it's best to view linting as an implementation or automation of our guidelines that helps us follow them seamlessly. Using API linting without guidelines is like describing `GET /products` with OpenAPI without analyzing user needs. Our guidelines provide reasonable and reasoned design needs (section 16.5.4). Without them, API linting can lead to irrelevant, cumbersome, and incorrect controls, making our API design needlessly complex and resulting in terrible API design.

API guidelines must be accompanied by API linting. Although API linting won't address all design concerns, it frees our minds from many details. An API linter can't determine whether our designs meet consumer needs, but it helps us avoid mistakes (casing, security), guides us in applying patterns (designing a search operation), and reduces the need to constantly refer to guidelines. This makes our job easier and allows us to focus on creating APIs that meet user needs.



NOTE Like guidelines, API linting is important for API governance but can also be useful on its own (section 16.3.2). It's essential for large organizations and helpful for small teams and individuals.

18.2 Using an API linter to automate API design guidelines

Automating or implementing API design guidelines (section 16.3) with an API linter requires considering the development and usage of API linting rules to identify what we need and choosing an API linter that meets these requirements.

18.2.1 Developing linting rules to automate guidelines

To develop API linting rules that automate our API design guidelines, we need to do the following:

- Decide what the rules verify.
- Easily and efficiently create all the rules we need.
- Create rules that return problem-solving feedback.
- Organize the rules.

Regardless of the API linter we use, we must analyze our needs and create only the necessary linting rules based on our guidelines and OpenAPI libraries.

The API linter must help us easily automate most of our guidelines with out-of-the-box features or via customization, minimizing configuration duplication. It should easily target elements in the OpenAPI documents for necessary checks, such as ensuring camel casing for properties, proper pagination parameters and response model structure for search operations, and consistent error-handling using the shared component from our OpenAPI library. We must ensure that the linting rules' feedback is easily interpretable to help us and others using our automated guidelines understand the nature of the problem, where it is, and how to fix it if needed.

To automate our guidelines, we'll create various rules covering different topics. Organizing them into different groups simplifies creation and allows for easy combination or independent use. For example, we can separate design and OpenAPI authoring rules. This can also allow the secure rollout of new rules as guidelines evolve.



NOTE Create API linting rules as you build your API design guidelines (section 16.5).

18.2.2 Using our automated guidelines while designing APIs

To use our automated guidelines when designing APIs, we'll need to

- Share the linting rules that automate our guidelines.
- Customize the linting rules that apply to an API.
- Ignore specific problems detected by the linting rules.

We'll need to use the linting rules that automate our guidelines to design different APIs and share these rules with others. To avoid risky duplication, we must pull them from a centralized source.

We may need to adjust our linting rules for certain APIs or create specific rules for different types or generations of APIs. Some APIs may also require locally specific rules. Additionally, as our guidelines evolve, we'll introduce new linting rules in a controlled manner.

Our linting rules don't just detect errors; they also offer optional recommendations based on context. For instance, although pagination may be required, search filters are optional in search operations. We should be able to ignore the "missing search filters" problem when necessary.

18.2.3 Choosing an API linter

In this chapter, we'll use the Spectral API linter (detailed in section 18.3) to automate our API design guidelines. I chose Spectral for this book because it's open source, powerful, flexible, widely adopted, and supported by different API tools. But most importantly, it matches our needs and allows us to do the following:

- Automate a significant part of the guidelines.
- Customize checks and reuse elements.
- Return helpful problem-solving feedback.
- Organize rules in different groups.
- Share the rules that implement the guidelines.
- Tweak the use of shared rules when linting.
- Add API-specific rules when linting.
- Ignore specific problems when linting.

The API tool world is always evolving, and although Spectral is a great API linter, there are and will be other options to consider. Alternatives must cover these needs, and I suggest exploring new possibilities, because I expect API linters to become more intelligent

with AI. They should simplify coding, automatically fix problems, and provide guidance based on a broader context than just the API description. For instance, linters like Spectral can't determine whether an operation should be a long operation; that requires knowing the API's context (subject matter and architecture).



NOTE API linting is not specific to OpenAPI and REST APIs. Spectral can also lint asynchronous APIs when using the AsyncAPI format. There are also linters for gRPC and GraphQL APIs. Ask your favorite search engine.

18.3 Introducing Spectral

We'll use Spectral to illustrate the automation or implementation of our API design guidelines and use the result when designing APIs to make our job easier. Spectral is an open source linter. It supports OpenAPI, AsyncAPI, and JSON Schema formats and can be used with any JSON or YAML document. This section examines running the Spectral CLI, how Spectral lints an OpenAPI document, and how to edit Spectral rules.



NOTE Check the Spectral CLI installation details at <https://github.com/stoplightio/spectral>. This chapter showcases many Spectral features, tips, and tricks, but it doesn't cover all the possibilities and challenges. Check my website at <https://apihandyman.io/the-design-of-web-apis> to get all the code examples and more details. For more, refer to the Spectral documentation or search "spectral" on my website.

18.3.1 Linting an OpenAPI document with Spectral CLI

Figure 18.3 shows that we can run the Spectral CLI using a command such as `spectral lint api.openapi.yaml -r rule.spectral.yaml` command. The `api.openapi.yaml` file

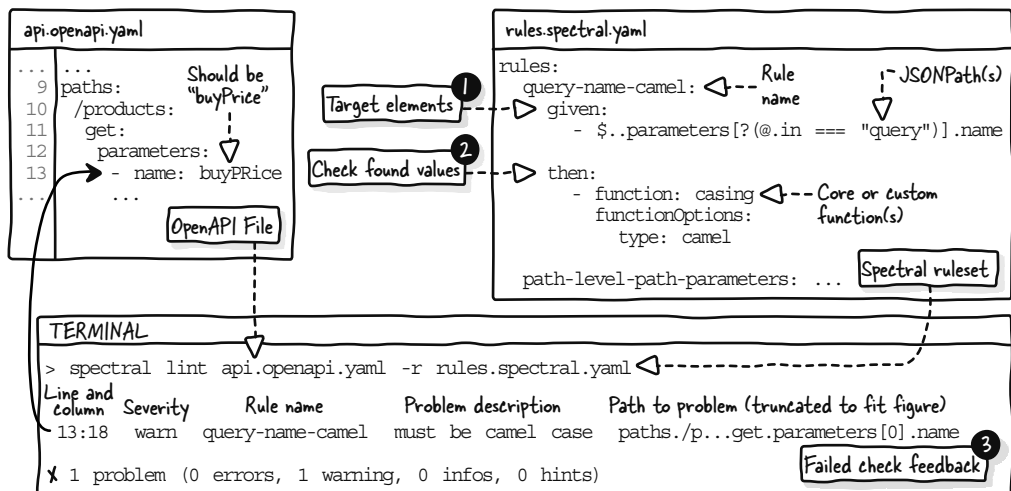


Figure 18.3 Spectral rules target elements with `given` key (JSONPath) and check found values with `then`. Spectral outputs failed checks as “problems.”

is the OpenAPI document we lint. The `rules.spectral.yaml` file is a Spectral ruleset: a YAML file containing the definition of the rules we want our OpenAPI document to follow. The CLI output lists the detected problems.



TIP Running Spectral in your IDE's terminal makes problems clickable, opening the OpenAPI file on the problem's location. Use `spectral lint --help` to see all Spectral CLI options. Check the `-f` or `--format` parameter to tweak the output format (`junit` or `json`, for example); this can be convenient for integration in CI/CD chains.

18.3.2 How Spectral lints an OpenAPI document

As illustrated in figure 18.3, Spectral analyzes the OpenAPI document with each rule defined under the `rules` key of the Spectral ruleset in three steps:

- 1 Target the elements to check (`given` list).
- 2 Check the found elements' values (`then` list).
- 3 Output a problem for each value that doesn't pass the check.

The `rules.spectral.yaml` Spectral ruleset contains two rules: `query-name-camel` and `path-level-path-parameters`. The `query-name-camel` rule checks whether the names of all query parameters defined in the OpenAPI document are camel-cased.

This rule gets all query parameter names with `$.parameters[?(@.in == "query")].name`. This is a JSONPath: a standard for selecting values within a JSON or YAML document (given is detailed in 18.5). Then the rule applies the `casing` function with the `camel` option to ensure that each found name is camel-cased (then is discussed in section 18.6).

The CLI output shows the location and severity (importance) of each problem. The `query-name-camel` rule with default severity `warn` detects that the `buyPrice` query parameter of `GET /products` is not camel-cased; it should be `buyPrice` (section 18.7 discusses rule feedback).



NOTE After deciding which linting rule to create to implement our guidelines (section 18.4), we'll use the same three steps to develop Spectral rules: target, check, and feedback.

18.3.3 Editing Spectral rulesets

We will use YAML Spectral rulesets in this book, JSON, JavaScript, or TypeScript formats are also possible; however, JavaScript and TypeScript can lead to overly complicated code, making rulesets challenging to maintain and modify and less interoperable with Spectral-compatible tools. Additionally, JSON is more complex to edit and does not support comments as YAML does.

You can create a Spectral ruleset with any code editor. Check your IDE marketplace for Spectral extensions. For example, Microsoft Visual Studio Code has a Spectral extension that provides YAML and JSON Spectral file validation and hints while

editing, along with Spectral linting for while-editing linting of OpenAPI files based on Spectral rules.

18.4 Deciding what API linting rules verify

Before creating linting rules, we must decide what to verify. We must create rules that are needed and have appropriate granularity. If we're not careful, we can create rules that enforce unnecessary or incorrect patterns, complicating our work or leading to bad APIs. Rules should not check too many aspects simultaneously or be numerous for no reason, as this can make the feedback overwhelming and challenging to address. To decide what our API linting rules verify, we can do the following:

- Use API design guidelines to create only needed rules.
- Split our needs into small chunks that we'll combine to cover broad topics.
- Use shared OpenAPI components to simplify what we verify.
- Ensure appropriate rule granularity with a concise name and description.



NOTE Creating only needed rules and ensuring that they have an appropriate granularity is a concern with any (API) linter. When creating rules, use your learning about error-handling (section 9.8).

18.4.1 Using our guidelines to create only needed rules

Each API linting rule must have a sourced and valid reason for existing. Creating rules based on guidelines that contain only sourced and valid recommendations, principles, and recipes (section 16.5.4) ensures that our linting rules have a valid purpose and are genuinely necessary.

We can add a link to our guidelines in a rule to indicate its origin. Suppose we create a `resource-name-plural` Spectral rule because a section of our guidelines at <https://guidelines.intra/principles#resource-names> mentions “resource names must be plural.” We can add `documentationUrl: https://guidelines.intra/principles#resource-names` to the rule.

Suppose we create a `resource-id-number` rule to ensure that all resource IDs are numbers, but we cannot back it up with our guidelines. This could mean we need to expand our API design guidelines to cover this topic (section 16.5.3) without forgetting to explain the value of such a rule (section 16.5.4). Our research may make us realize we don't need this rule.



NOTE Without guidelines, you can explain a rule using a link to an architecture decision record (ADR) or other source (section 16.2). However, guidelines are strongly recommended; they offer a clearer vision than many linting rules and coverage for aspects that linting may not cover or that apply to API types other than REST.

18.4.2 Finding small problems to solve

As illustrated in figure 18.4, we must pick our guidelines’ principles or rules and smaller statements inside recipes to create meaningful linting rules. We won’t create a single `api-guidelines` rule that implements our guidelines; it would be a nightmare to implement. Recipes are also too coarse. A `search-operation` rule implementing the “How to search elements” recipe must check whether a `GET /.../resources` operation has the proper pagination, sort, and search filter query parameters and success and error responses with the appropriate statuses and data models integrating pagination metadata. These many aspects may not all need the same severity; pagination may be mandatory, and search filters may be optional.

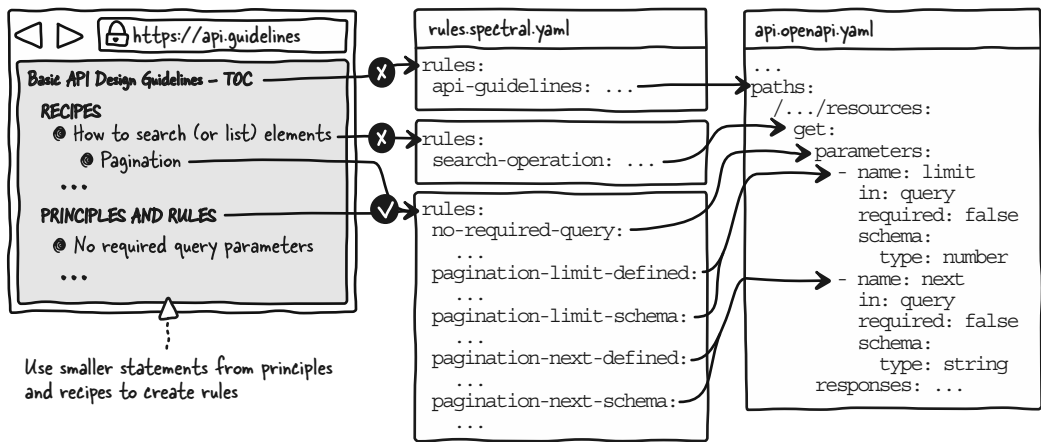


Figure 18.4 Create Spectral rules based on smaller statements from principles and recipes to help detect and fix precise problems.

Recipes rely on finer-grained principles and statements that we should use as a basis for our rules. We can combine rules to cover a topic comprehensively, such as “How to search elements”:

- Some rules can focus on ensuring that search operations have `next` and `limit` pagination as optional query parameters.
- The `pagination-next-defined` and `pagination-limit-defined` rules check that the pagination parameters are defined for search operations.
- The `pagination-next-schema` and `pagination-size-schema` rules ensure that each parameter has the correct schema.
- The generic `parameter-query-optional` rule ensures that no query parameter is required.
- Other rules address other aspects of the “How to search elements” recipe.

18.4.3 Simplifying rules with shared OpenAPI components

Consider adding new shared OpenAPI components to your OpenAPI library before dividing guidelines into smaller chunks. This simplifies OpenAPI authoring and fosters consistency (section 17.6), reduces the number of problems that linting detects, and makes it easier to fix them. For example, instead of creating various complex rules to verify that a 500 error response has an appropriate media type, description, and schema, define a reusable response in your OpenAPI library and create a Spectral rule to ensure that each 500 response references this reusable response.



NOTE Refer to section 18.6.4 to learn how to code rules that verify the use of shared OpenAPI components with Spectral.

18.4.4 Ensuring appropriate granularity with a concise name and description

An effective way to create sensible rules that solve right-sized problems is to provide concise names and descriptions that clarify what the rules verify. We can add a description to Spectral rules, as shown in figure 18.5.

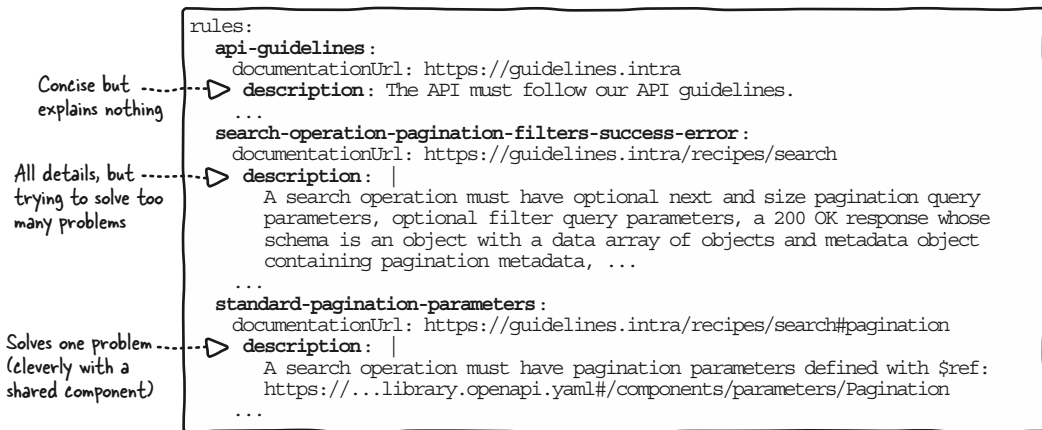


Figure 18.5 If the name and description don't clearly convey what the rule checks, or if they're long and describe many checks, the rule doesn't have the appropriate granularity.

A name and description, such as `api-guidelines` and “The API must follow our API guidelines,” are concise but not meaningful. The `search-operation-pagination-filters-success-error` rule is described as “A search operation must have optional next and size pagination query parameters, optional filter query parameters, a 200 OK response whose schema is an object with a data array of objects and metadata object containing pagination metadata, ...” We know exactly what's expected, but we're blatantly trying to perform too many checks with one rule.

The `standard-pagination-parameters` rule has description set to “A search operation must have pagination parameters defined with `$ref: ...`.” The description is a bit lengthy, but that’s because of the JSON pointer to a shared OpenAPI component. It’s clear that we’re addressing only one small problem.



NOTE The description may also explain why the rule exists if there are no guidelines to reference in `documentationUrl`. However, ADRs and guidelines are strongly recommended (section 18.4.1); linting rules should focus on fixing problems rather than diving into the “why.” We may need to split the rules identified here due to Spectral implementation or feedback concerns; see section 18.7.3.

18.5 Targeting elements to check in the OpenAPI documents

Once we have decided on the purpose of a linting rule, we can develop it. Like many other linters, Spectral rules target elements in OpenAPI documents (`given`) to check whether their values match expectations (`then`). We start developing linting rules by finding the elements to check; these can be anything, such as property names, operations, search operations, reusable components, or OpenAPI metadata. This section discusses the following API linting concerns and in the process further explains Spectral and JSONPath, introduced in section 18.3:

- Targeting the proper elements
- Targeting anything in inline or referenced OpenAPI elements
- Targeting references to local or shared OpenAPI components
- Creating a library to target typical OpenAPI elements



NOTE Spectral uses JSONPath to target elements, a standard for selecting values within a JSON (or YAML) document. It is defined by RFC 9335 (<https://datatracker.ietf.org/doc/html/rfc9335>). However, Spectral relies on the JSONPath Plus implementation, which adds some features, like many implementations created before JSONPath standardization. I recommend referring to its documentation (<https://github.com/JSONPath-Plus/JSONPath>) when creating JSONPaths for a Spectral rule’s `given`.

18.5.1 Starting rule development by targeting the proper elements

When creating a linting rule, it’s crucial to ensure that it targets the intended elements for proper checking. With Spectral, this means ensuring the accuracy of JSONPaths defined under a rule’s `given`. To do so, create new rules with a temporary `then` that uses the `undefined` function, as shown in figure 18.6, and run it on a test document. This allows Spectral to display what `given` finds.

We decided to ensure that our APIs use semantic versioning. We add the `semantic-version` rule name under the `rules` key with a `documentationUrl` and `description` to indicate its origin and purpose (section 18.4). To target the API version, we add the

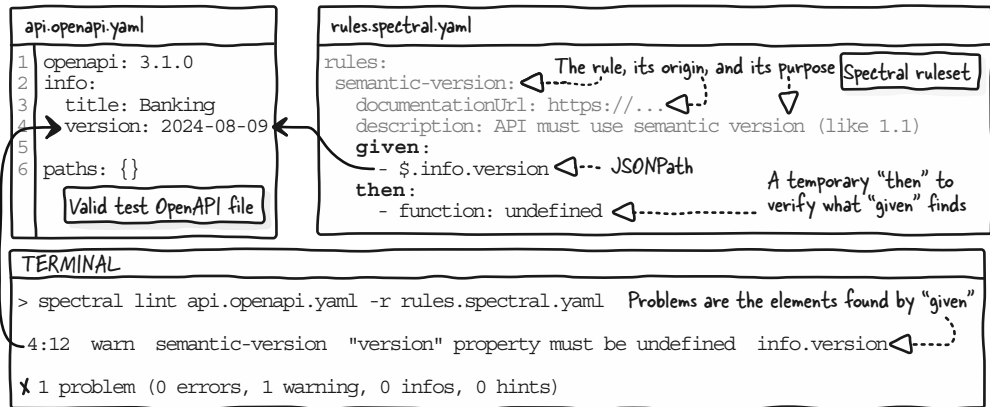


Figure 18.6 Use the undefined function to see what the rule's given finds. Test the rule on a valid OpenAPI document.

`$.info.version` JSONPath to the given list. This path targets the `version` property of the `info` object located at the OpenAPI document's root (`$`). We add a temporary `then` item calling the `undefined` function (we'll replace it in section 18.6).

To verify that our JSONPath doesn't miss or return unexpected elements, we run our Spectral ruleset on a realistic and syntactically valid OpenAPI document that has the targeted elements (and others we'd like to exclude when using complex JSONPaths from section 18.5.2). Such a test document can be based on a template from our guidelines or an API we've designed. Running the Spectral CLI on our test document shows that the rule's JSONPath is correct: the `semantic-version` rule detected a "problem" at `info.version`, the location we want to target.

18.5.2 Targeting any element in the OpenAPI document

We must be able to target any element within the OpenAPI documents we lint, such as

- All put and post operation response objects (to check whether they have a 400 defined)
- All query and path parameters (to check their name case or schema)

To target such elements with JSONPath in our Spectral rule, we can't rely only on `$` (root) and `a.b` ("b" of "a"), seen in section 18.5.1. JSONPath proposes wildcards and filters:

- `a.*` returns "a" object's keys' values or "a" array items.
- `..b` traverses the document to get all "b".
- `[a,b]` returns values of elements having a or b keys.
- `[?(conditions)]` returns elements matching the conditions.

As shown in figure 18.7, we use `$.paths.*` to get all path objects. To get all `post` and `put` operations, we can use `$.paths.*.post` and `$.paths.*.put` for separate paths or

a single `$.paths.*[post,put]`. Finally, `$.paths.*[post,put].responses` returns their responses objects.

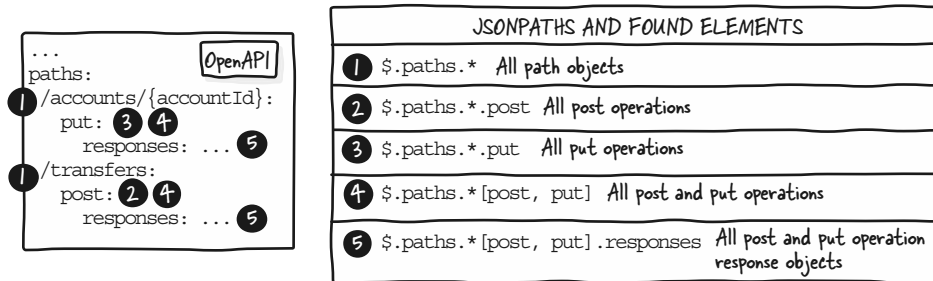


Figure 18.7 Use `a.*` and `[a,b]` to target all post and put operations' responses objects with JSONPath.

Figure 18.8 shows how to get all query and path parameters. We must consider path-level, operation-level, and reusable parameters: respectively, `$.paths.*.parameters`, `$.paths.*.*.parameters`, and `$.components.parameters`. A single `$.parameters` can replace the three. With `$.parameters[?(@.in === "query" || @.in === "path")]`, we get all query and path parameters wherever they are defined. The `@` represents the current element (hence, an item of the parameters list), and `@.in` is the `in` property of this element. The rest is similar to JavaScript: `===` indicates equality (the opposite is `!==`), and `||` represents the “or” logical operator (use `&&` for “and”). Alternatively, we could use a regex filter `[?(@.in.match(/query|path/))]`; but when used with `..`, we must add a `@.in &&` condition because the filter runs on any elements, including those not having an `in` field.

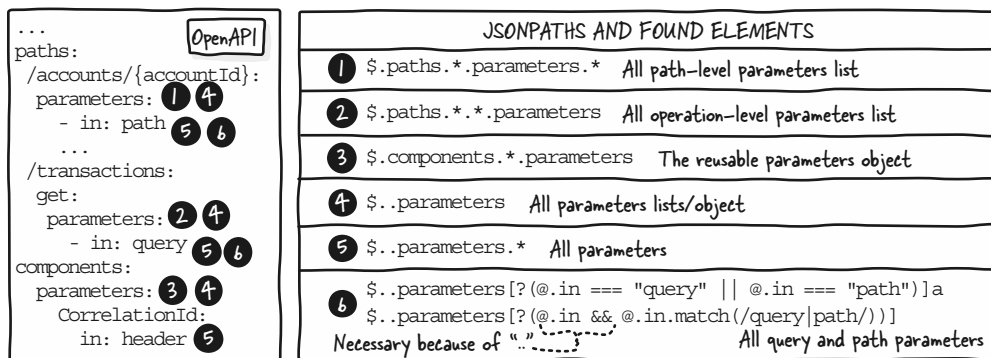


Figure 18.8 Use `a.*`, `..b`, and `[?(conditions)]` to get all query and path parameters with JSONPath.



NOTE For clarity and accuracy, define multiple paths instead of a single complex path whenever possible. The `$.b` path may grab unexpected `b` properties

in schemas; add more levels after `..` to prevent difficulties (`$.b.c`). Check section 18.9.2 to see how to ignore false positives.

18.5.3 Dealing with references to local or shared components

The OpenAPI documents we lint may contain elements defined inline or via `$ref` targeting the local components or shared ones from our OpenAPI library. Our API linter must help us handle that complexity.

With Spectral, we don't need to worry about whether elements are hidden behind `$ref`, even if the reference targets another OpenAPI document. By default, a Spectral rule is executed on a “resolved” document where all references are replaced by their value. A rule with a `resolved` flag set to `false` can work on the raw OpenAPI document that still contains its `$ref`. Figure 18.9 contrasts the two behaviors.

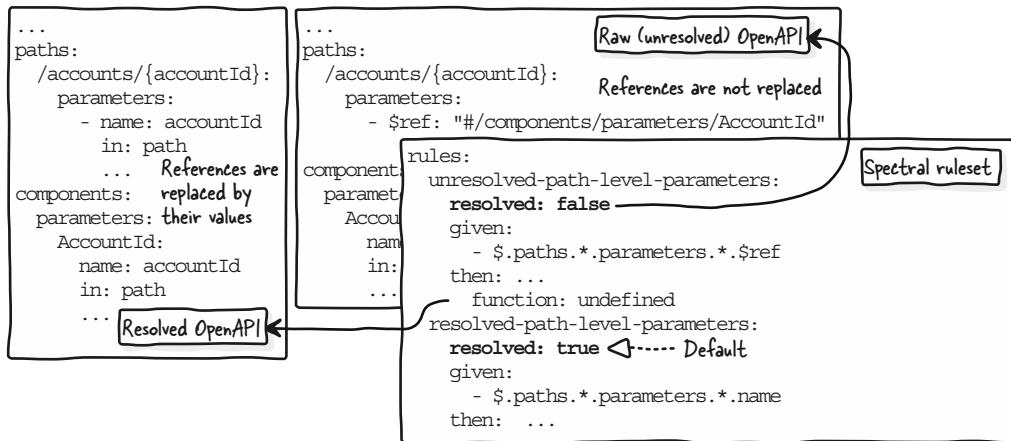


Figure 18.9 Set the Spectral rule's `resolved` flag to `false` to work on unresolved documents that still have their `$ref` elements.

The `resolved-path-level-parameters` rule targets `$.paths.*.parameters.*.name`. Because its `resolved` flag is undefined (or `true`), it gets `accountId` even though the parameter is defined via a `$ref`. The `unresolved-path-level-parameters` rule targets the `$ref` property of path-level parameters. The rule gets the `#/components/parameters/AccountId` pointer because `resolved` is `false`. Each rule's `given` will return nothing if we inverse the rule's `resolved` flag.



NOTE Section 18.6.4 will use `resolved` to ensure that references to local or shared components are used.

18.5.4 Creating a library to target typical elements

A library that helps us target generic and guidelines-specific elements, avoid duplication and errors, and clarify our rules would be helpful. We'll likely need to target the

same elements, such as “all parameters” (generic) or “search operations” (guidelines-specific), with various rules. Spectral uses *aliases*, which are reusable given values that can be used as is or extended to create new paths. Figure 18.10 illustrates how to define, use, and extend Spectral aliases.

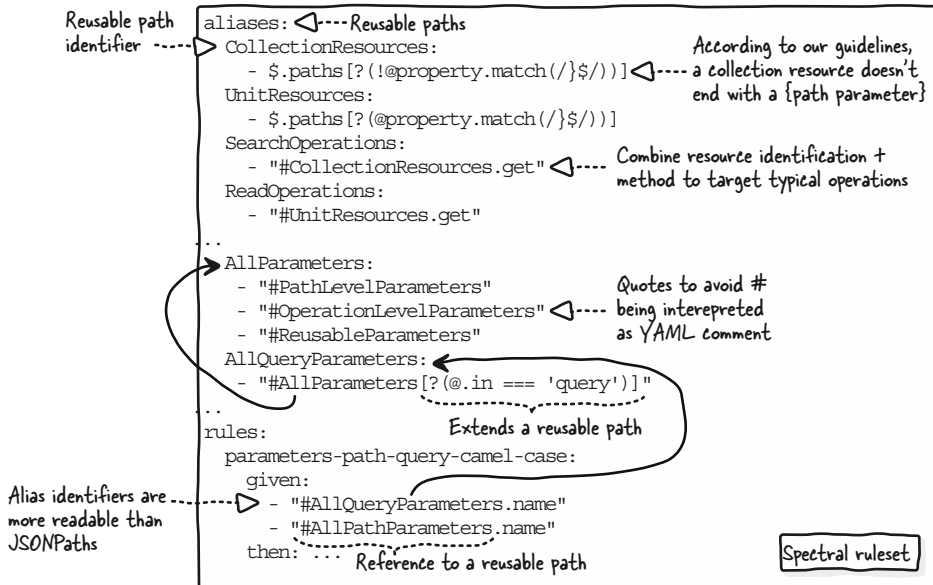


Figure 18.10 Define Spectral aliases, which are reusable given values, under `aliases` and use them with `#AliasName` to prevent error-prone duplication of JSONPaths across different rules.

Spectral aliases are defined under `aliases` at the root of the ruleset file (the same level as `rules`). Each alias has an identifier, which is a key under `aliases`. Once defined, it can be used with `#AliasName` in any rule's `given` or in other aliases. An alias can be extended with `#AliasName.relative.JSONPath`.

Because our guidelines indicate that a collection has a `/path/to/resources` and a unit resource has a `/path/to/resources/{resourceId}`, we defined `CollectionResources` and `UnitResources` aliases that detect these patterns using a `@property.match(/}$/)` filter, where `@property` represents the keys under `$.paths`. We can combine resource type and HTTP method to identify typical operation; the `SearchOperation` alias is `#CollectionResource.get`. The `#CollectionResource` alias name makes the path clearer than a pure JSONPath.

We can also define more generic aliases. `AllParameters` combines `#PathLevelParameters`, `#OperationLevelParameters`, and `#ReusableParameters`, whose values are from section 18.5.2. We can use aliases similarly at the rule level, as shown in the rule `parameters-path-query-camel-case`.

18.6 Checking element values

Now that we have targeted elements in the OpenAPI documents, we can check whether their values conform to our expectations and, hence, to our guidelines. With Spectral, this means adding items to the rule's `then`. This section discusses the typical checks we need to perform when implementing our guidelines with an API linter:

- Performing basic checks on values or keys, such as property casing or a “parameters required” flag
- Ensuring that an element is absent or present, such as a property, response, or parameter
- Ensuring that local or shared components are used, such as a local reusable resource model or a standard 500 response
- Ensuring that part of a JSON schema applies a pattern, such as a search response data model
- Performing cross-element checks, such as comparing the request and response



NOTE Remember to remove (or comment with #) the temporary call to the undefined function (section 18.5.1).

18.6.1 Performing basic checks on values and keys

We'll need to perform basic checks on values or object keys to ensure, for example, that parameter and property names follow our naming conventions and that query parameters are not required. Figure 18.11 shows how we can do this with the `then` part of a Spectral rule.

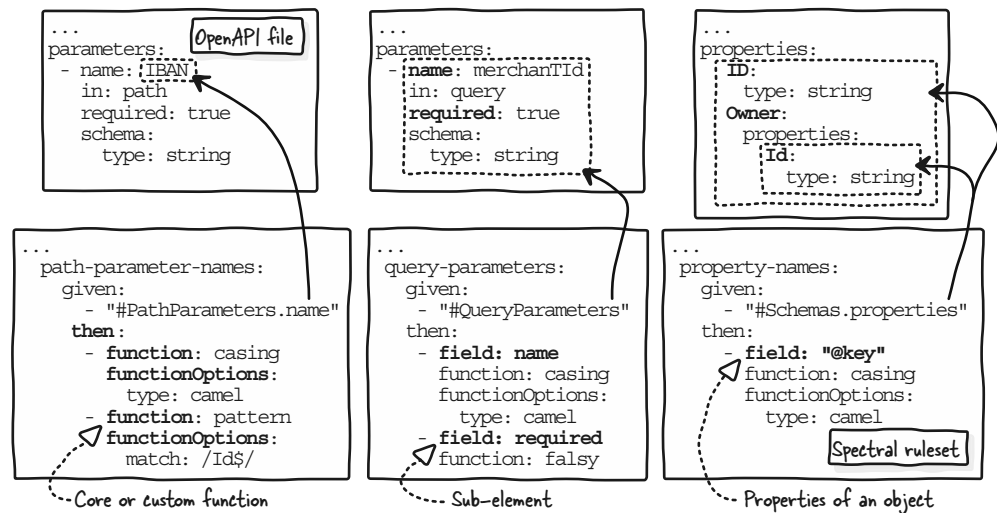


Figure 18.11 Define one or multiple checks with `then`. Indicate a function and, optionally, add `field` to check specific properties. Set `field` to `"@key"` to check a property name.

A Spectral rule's then can define one or multiple checks. The `path-parameter-names` rule ensures that path parameter names are camel-cased and end with "Id." The `property-names` rule only ensures that property names are camel-cased.

Each check uses a function: a "core" function provided out of the box (check <https://github.com/stoplighdio/spectral/blob/develop/docs/reference/functions.md> for a complete list) or a custom one we create (discussed in section 18.6.6). A function may or may not have `functionOptions`. The `pattern` function expects a `match` (or `notMatch`) option, a regular expression, to verify that a value matches it. Similarly, `casing` requires a `type`. The `falsy` function ensures that a value is not undefined, null, 0, false, or an empty string and needs no options (opposite: `truthy`).

A then item can target sub-elements of the found value using `field`. For example, in the `query-parameters` rule, we check that `name` is camel-cased and `required` is false or not provided (making the parameter optional). In that case, we may question our choice of checking different aspects of one element: the rule's purpose and feedback may be unclear. We can use the `@key` special `field` value to verify the keys of an object, as in the `property-names` rule.

18.6.2 Ensuring that an element is defined

To implement our guidelines, we'll typically need to check that a 500 response is defined under the `responses` object of any operation or that a parameter is present in a `parameters` list: a `Correlation-Id` header (which helps to track requests between systems), for example.

We can use the `defined` function to detect missing elements in an object with Spectral, which requires `then.field`. Indeed, given: `$.paths.*.responses.500` will not trigger then if no 500 response is defined. Figure 18.12 illustrates this with a less obvious use case: making us always consider what should be required in a data model (section 5.2.4).

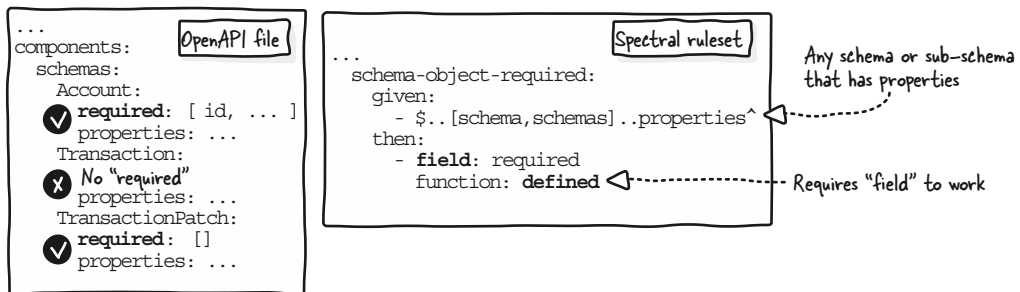


Figure 18.12 Combine `field` and the `defined` function to ensure that an element is present in an object with Spectral.

The `schema-object-required` rule targets all schemas defining properties with `$.paths.*.responses.500` (a `^` means "parent of a"). Under `then`, we add

the defined function and set field to required. That way, we're sure to define the required properties; if there are none, we can set required to []. For the 500 case, use `$.paths.*.responses` and then `field: "500"`.

The technique differs when checking the presence of an element in an array. We also need `then.field`, but we use the `schema` function, which ensures that a value matches the JSON schema defined in `functionOptions.schema`. Figure 18.13 shows how to ensure that a `Correlation-Id` request header is defined in the path-level parameters list.

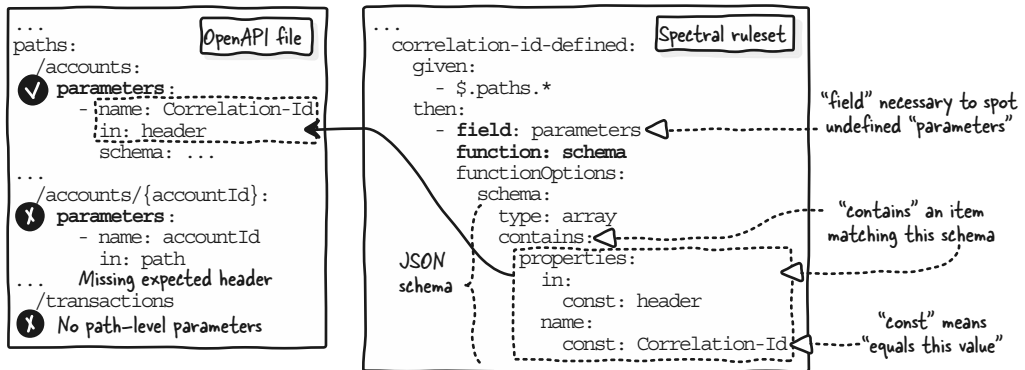


Figure 18.13 Use the `schema` function and the `contains` JSON Schema keyword to ensure that an element is present in an array with Spectral.

To ensure that the `correlation-id-defined` rule's `then` is triggered even if the parameters list is not defined, we use the `$.paths.*` path and add `field: parameters`. The JSON schema we provide to the `schema` function ensures that we get an array (`type: array`) that contains an item (`contains`) with `in` set to `header` and `name` set to `Correlation-Id` (`const`).

18.6.3 Ensuring that an element is not defined

We'll need to ensure that elements are not defined to implement our guidelines. For example, operations on a resource path without path parameters must not have a 404 response, and the `Correlation-Id` from section 18.6.2 must not be defined at the operation level. We already know how to do this with Spectral: we just need to use the undefined function on the appropriate targets. With `$.paths[?(!@property.match(/{/}))].*.responses.404`, we get the 404 responses of operations whose paths don't contain `{pathParameter}` (inspired by the `CollectionResource` alias from section 18.5.4). With `$.paths.*.parameters[?(@.name === "Correlation-Id")]`, we get operation-level parameters named `Correlation-Id`.

18.6.4 Checking references

Our guidelines include ready-to-use components defined in an OpenAPI library to simplify design work and ensure consistency; a standard 500 response or the `Correlation-Id` request header, for example. Our OpenAPI authoring guidelines suggest referencing local components, such as request or response bodies. We already know how to create Spectral rules to guide us through these concerns.



NOTE Remember to set the Spectral rule's `resolved` flag to `false` when working on `$ref` (section 18.5.3).

Figure 18.14 shows that we proceed similarly to section 18.6.3 to enforce using our library's `CorrelationId` shared component. The `correlation-id-standard` rule targets all path objects and then focuses on the `parameters` field. The JSON schema of the schema function only differs at the `contains` level. JSON Schema's `const` accepts any value, so we use it to check whether the `parameters` array contains the appropriate reference object (`$ref: ".../CorrelationId"`). We proceed similarly for the 500 standard response. The JSON schema differs slightly; we just need to check that the value found by `given` is `$ref: ".../Error500"` with `const`. To see whether an element such as a response body uses a reference to a schema instead of an inline schema, we just need to proceed as in section 18.6.2 and check for the presence of a `$ref` field.

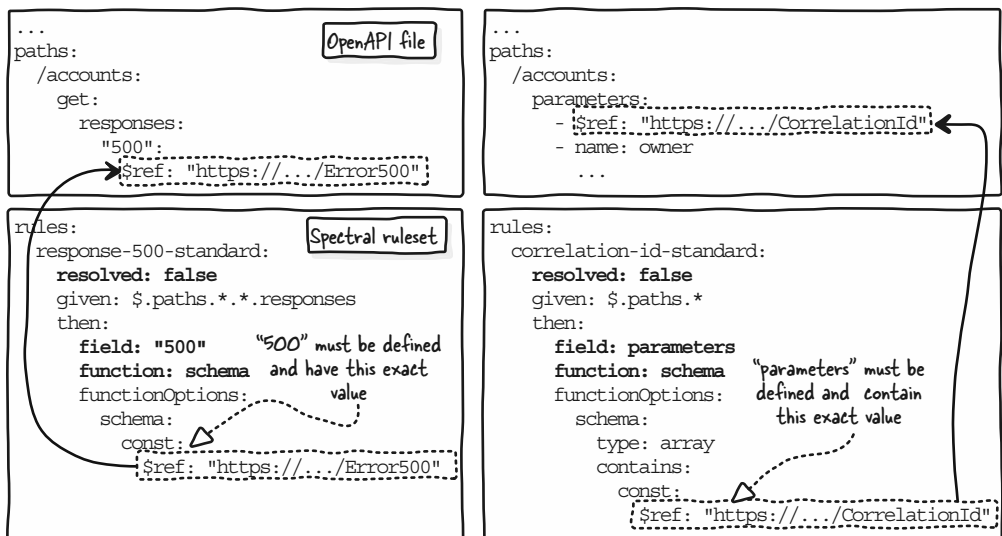


Figure 18.14 Use `then.field` and the `schema` function to ensure that a shared component is used in an object or an array.

18.6.5 Checking partial JSON schemas

Our guidelines define parts of schemas that our data models must comply with. For example, the search operation response format must be an object with a data array containing objects of any kind. As is often the case with Spectral, the `schema` function helps us with this.

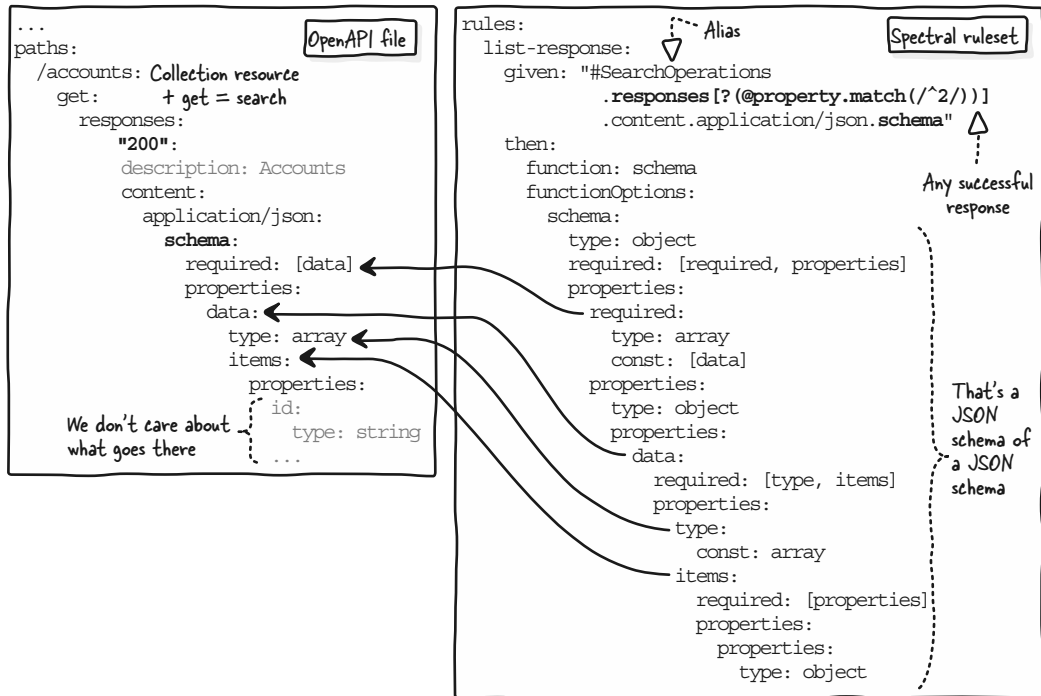


Figure 18.15 Be mindful that when the value is a JSON schema, the schema must be thought of as a JSON schema of a JSON schema!

As illustrated in figure 18.15, we use the `SearchOperation` alias from section 18.5.4 (collection resource plus `get` method) to get the schemas of search operations. We add `responses[?(@property.match(/^(2/)))]` to get all successful responses. Finally, `content.application/json.schema` gives the schemas we're looking for.

For the `schema` function, we can't use `const` and copy and paste the expected schema because the definition of the `items` in the `data` array will vary from one operation to another. Instead, we define a partial JSON schema of the expected JSON schema. It ensures that the found value has `required` and `properties` keys, `properties` has a `data` key, and `data` has a `type` key set to `array` and an `items` key with a `properties` object.



TIP The JSON Schema validation feedback of the `schema` function may be unclear; check section 18.7 to make it user-friendly.

18.6.6 Performing cross-element checks

To implement our guidelines, we may need cross-element checks, such as ensuring that the request and response data models of a creation operation are consistent or that search filters are consistent with the response of a search operation. We can use Spectral custom JavaScript functions to perform such checks. This section examines an example enforcing the use of the same read-write schema reference in a request and response (section 17.2.5).



NOTE For more information about custom functions, such as their configuration, limitations, performance, and security, check the Spectral documentation at <https://github.com/stoplighdio/spectral/blob/develop/docs/guides/5-custom-functions.md>.

In figure 18.16, the `consistent-rw-reference` rule targets `post` and `put` operations and uses the `consistentReferences` custom function. This function is available because it's listed in the `functions` at the root of the Spectral ruleset. When Spectral starts, it looks for a `functions/consistentReference.js` file (relative to the ruleset). The specific name of the function in the JavaScript file (`compareReferences`) doesn't matter; it must be the default export and have three arguments:

- `input`—A value found by `given`
- `options`—The `functionOptions` of the `then` item
- `context`—An object containing information such as input location and the OpenAPI document, useful for verifying distant parts based on input

This function returns a list of problems, which is empty if no problem is found.

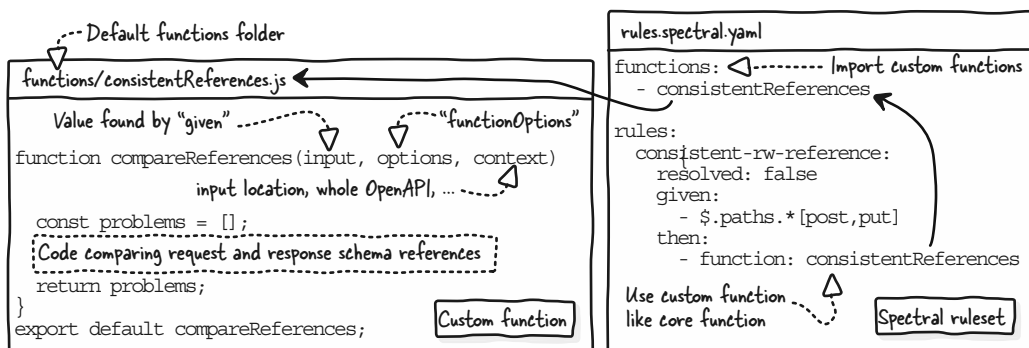


Figure 18.16 To use a `myFunction` custom function, create a `./functions/myFunction.js` file and add `myFunction` to the `functions` list in a Spectral ruleset.

Our Spectral custom function aims to compare the reference (`$ref`) used in an operation's request and response body. In figure 18.17, the function uses the `input` argument to set the `requestSchemaRef` and `responseSchemaRef` variables; `input` is a

regular OpenAPI operation object because of the `$.paths.*[post,put]` JSONPath used in the rule's given.

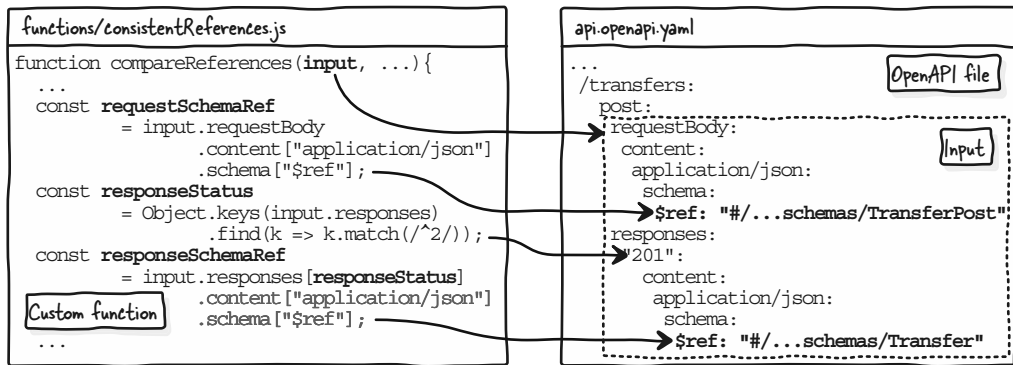


Figure 18.17 The input argument is a value found by the given of the rule calling the function.

In figure 18.18, the function compares the values of the request and response body references. If they don't match, it adds two elements to the `problems` list. Each problem includes a message and a path, based on the original input location found in `context.path`. If no path is defined, Spectral uses `context.path`.

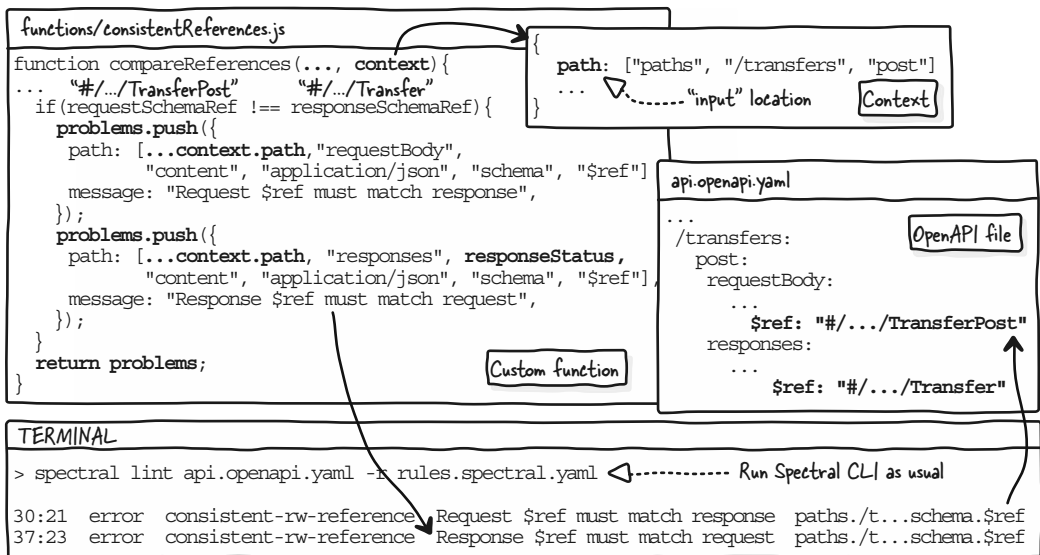


Figure 18.18 For each problem detected in the function, add an object with a message and an optional path to the `problems` list that the function returns. The CLI will display each problem with its message and path.

We run Spectral as usual to lint an OpenAPI document with the ruleset containing the `consistent-rw-reference` rule, which uses the `consistentReferences` custom function. The CLI output shows two problems for each faulty operation where the request and response body don't use the same read-write schema reference.

18.7 Returning helpful feedback when problems are detected

When we run Spectral (or any other API linter) and some of our rules detect problems in our OpenAPI document, we must ensure that the feedback helps us or others understand and solve the problems if needed. As illustrated in figure 18.19, our linter output must indicate

- Meaningful problems
- Exhaustive list of problems
- Their locations
- Their importance or nature
- How to solve them

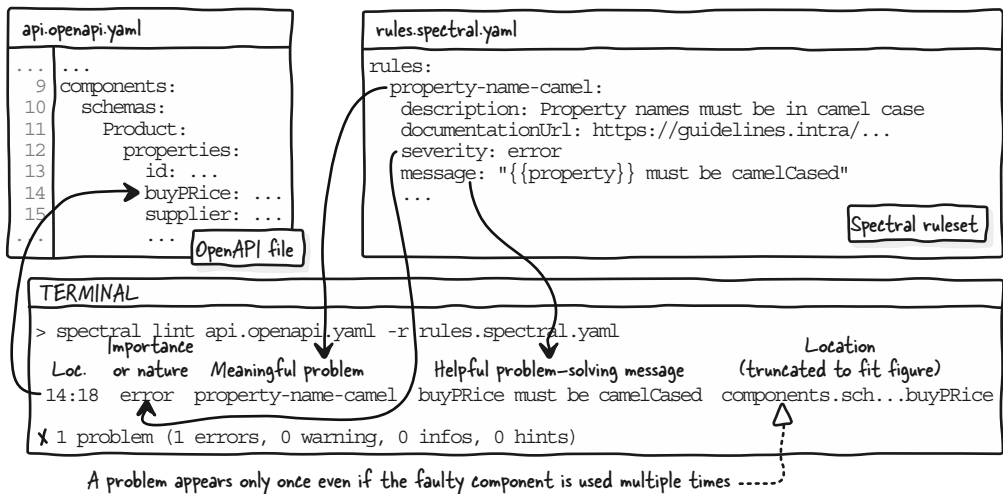


Figure 18.19 A helpful API linter lists meaningful problems, their location, their importance or nature, and how to solve them.

We already took into account detecting meaningful problems when we carefully considered what our rules verify and their names in section 18.4. Spectral shows all problems detected by a rule's multiple then items and doesn't repeat the same problem multiple times if it comes from an element used via a `$ref`. Spectral locates the problems with lines and columns (14:18) and dotted paths (`components.sch...buyPRice`). This section discusses what remains: using the rule's severity to indicate

how important a problem is or its nature, and tweaking the output message to help solve a problem.

18.7.1 Stating the importance or nature of a problem with a severity

Not all rules are equally important. It's crucial to differentiate between critical problems and optional patterns. We can also create rules to highlight elements that need further investigation. Figure 18.20 shows how we can use a Spectral rule's `severity` field to do so.

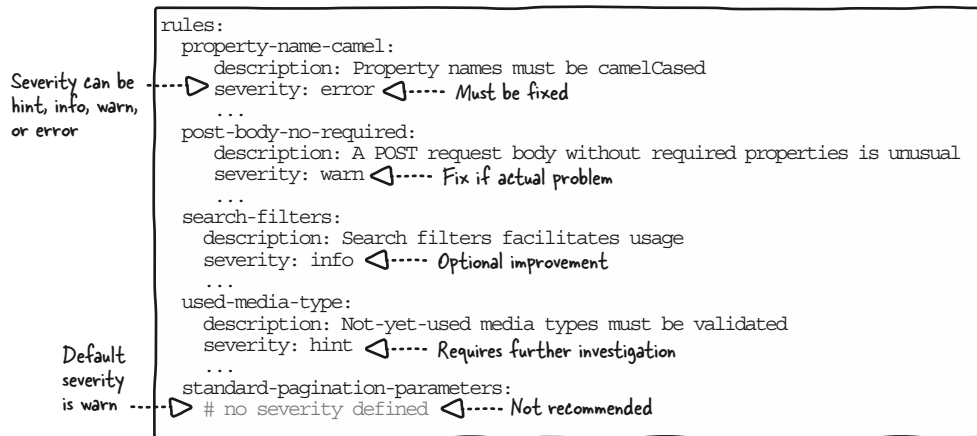


Figure 18.20 Always define a rule's severity to consider the effect of the detected problems and guide how to react to them. Define the meaning of each level.

The `severity` field has four possible values; it's up to us to decide how to interpret them. Here's my interpretation based on the "MUST," "SHOULD," and "MAY" from RFC 2119 (used in section 16.4.1 for our guidelines):

- An `error` **MUST** be fixed.
- A `warn` **SHOULD** be fixed if it's an actual problem.
- An `info` indicates an optional improvement (**MAY**).
- A `hint` requires further investigation.

For example, an `error` like missing security scopes or a non-camel-cased property name must be fixed. A `POST` operation request body without any required property is a `warn`; it rarely happens and likely needs to be fixed. With `info`, we can propose optionally adding search filters. With a `hint`, we can request an impact analysis for using a media type indicating a file upload.

A rule's default `severity` is `warn`. However, I recommend always explicitly indicating `severity` to consider the importance or type of problems the rule aims to detect and to guide our reaction.

18.7.2 Returning problem-solving message

We can rely on the problem message if our meaningful rule name alone is insufficient to determine how to fix the problem. This message has different sources; as illustrated in figure 18.21, Spectral first looks for the rule message, then it looks for the description, and finally it shows the message returned by the function. The rule's message supports `{{placeholders}}` that are replaced at runtime: `{{description}}` (rule description), `{{error}}` (function message), `{{path}}` (the problem's path), `{{property}}` (the last segment of the problem's path), and `{{value}}` (the found element value). For example, we can combine the rule description and function message with message: `"{{description}}:{{error}}"`.

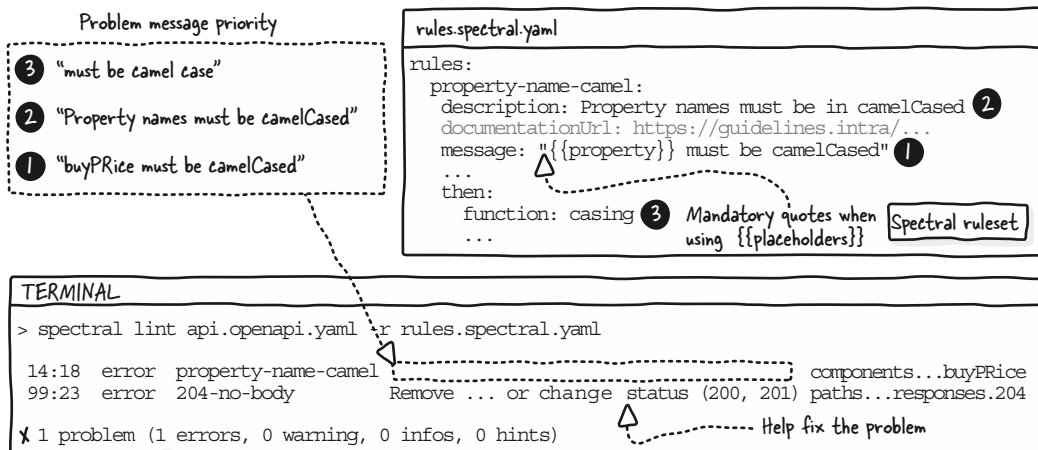


Figure 18.21 The message for a problem comes from the rule's message or description or is the message returned by the function. The message field can contain `{{placeholders}}` (requires quoting).

For the `property-camel-case` rule, the function message "must be camel case" falls short because users may not know what "camel case" means. Adding the "Property names must be in camelCase" description illustrates what's expected. We can be even clearer about the source of the problem by adding a message set to `{{property}}` must be camelCased, which will give "buyPRice must be camelCased" at runtime.

Similarly, for the `204-no-body` rule, which ensures HTTP compliance by checking that no content property is defined under a 204 No Content response, we can have a message set to "Remove the response body (content) or change status (200, 201)" instead of "content property must be undefined." It helps solve the problem by providing different alternatives.

18.7.3 Splitting rules due to severity or message concerns

We may need to split a rule to provide an accurate severity and a clear message. If a missing element results in a warn, info, or hint severity, we need error rule(s) to signify its incorrect definition. For instance, suppose our guidelines indicate that a search operation may have a `q` search parameter, which is a string. This could lead to a `search-q-defined` rule with the `info` severity (checking the parameter presence) and a `search-q-valid` rule with the `error` severity (checking the parameter schema).

We may be unable to provide a clear message due to a rule performing various checks. For example, a message set to `{{property}}: {{message}}` may still be unclear if a rule verifies different complex fields using `then.field` and the `schema` function. Spectral doesn't allow use to tweak the message for each `then`, so we'll need to split the rule to separate the checks and provide clear feedback. This problem may also indicate inaccurate purpose and granularity (section 18.4).

18.8 Organizing rules

We may end up with many linting rules; organizing them in smaller sets can be helpful for development and use. Spectral allows the import of other rulesets with the `extends` keyword. As illustrated in figure 18.22, we can organize our rules in sub-rulesets and have a main ruleset import them all. Additionally, as is the case for a rule, a ruleset can have a description explaining its purpose and a `documentationUrl` pointing to our guidelines' relevant page.

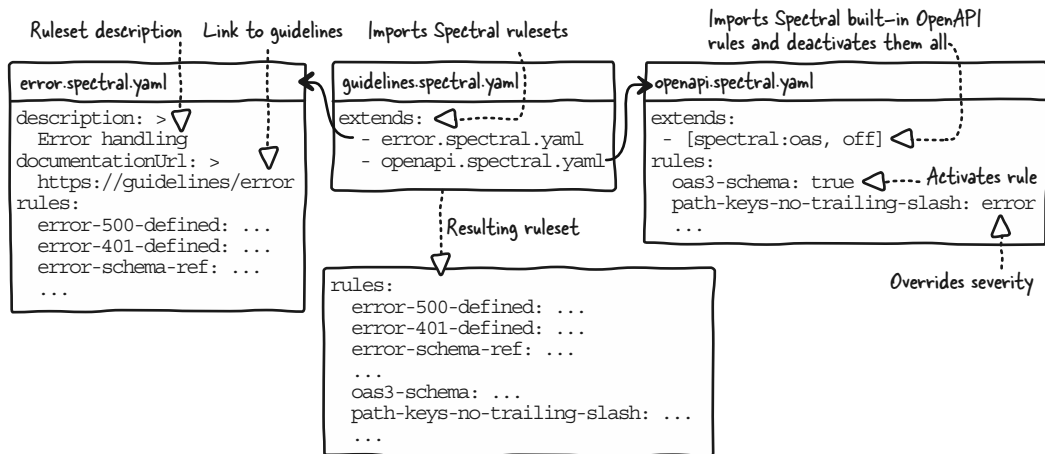


Figure 18.22 The `extends` list imports other rulesets and Spectral built-in rulesets. Rules can be deactivated on import to activate only the needed ones. Imported rule severity can be overridden.

We import the `errors.spectral.yaml` and `openapi.spectral.yaml` rulesets in `guidelines.spectral.yaml` by adding them to the `extends` list. We can use relative or absolute paths and URLs. The `errors` ruleset contains all rules related to errors. The `openapi`

ruleset implements our OpenAPI authoring guidelines; it uses custom rules and some Spectral built-in OpenAPI rules (see <https://github.com/stoplighio/spectral/blob/develop/docs/reference/openapi-rules.md>). It imports them with `[spectral:oas, off]`; `off` deactivates all rules (use `spectral:oas` to import the rules activated). We activate the rules we need by adding their name to `rules`; `oas3-schema: true` activates the rule that checks whether an OpenAPI document is syntactically valid, and `path-keys-no-trailing-slash: error` activates and overrides the severity of the rule that prevents having `/resources/` paths (which can cause problems).



NOTE Version your Spectral rulesets in a code repository; they must evolve with the design guidelines. Keep the different versions available to avoid breaking CI/CD pipelines of existing APIs with guideline modifications that introduce breaking changes in API linting unless you want to (section 16.5.5).

18.9 Using our automated guidelines when designing APIs

When designing an API, we need to be able to

- Use the shared rules that automate our guidelines.
- Customize the rules that apply to an API.
- Ignore specific problems.

18.9.1 Importing and tweaking the guidelines ruleset

As illustrated in figure 18.23, we can have a `.spectral.yaml` Spectral ruleset where we store the rules used to lint the OpenAPI document describing our API. This filename allows us to run the Spectral CLI without the `-r <ruleset>` option.

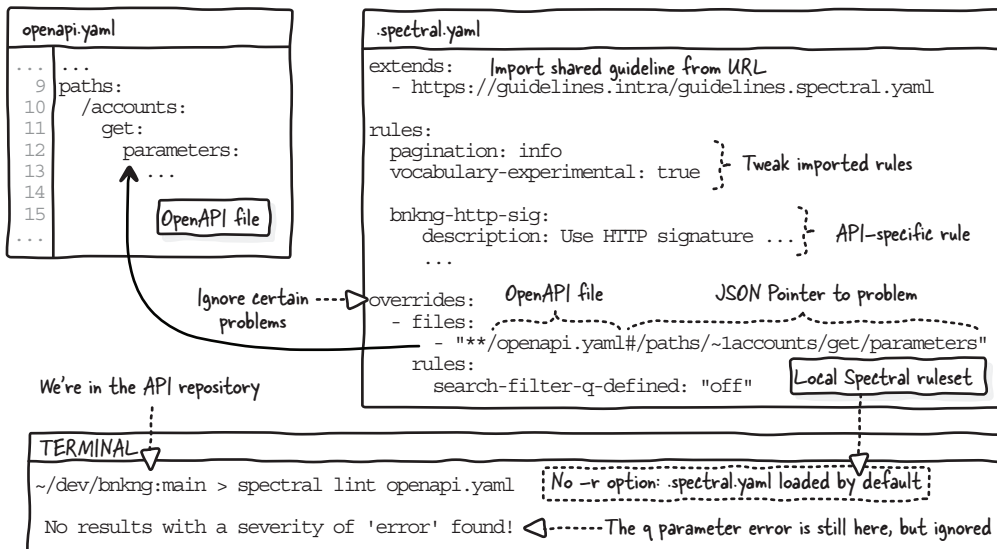


Figure 18.23 We can tweak our guidelines locally to adapt them to our needs and ignore non-fixable problems.

In this local-to-our-API Spectral ruleset, we reference our automated guidelines by adding the `extends` keyword (as seen in section 18.8). This allows us to import all of our rules or selected ones, deactivate some rules, or change their severity. This may be needed if we have different types of APIs following slightly different guidelines (which we should avoid, but we don't live in an ideal world). We can also target an older or unpolished recent (alpha) version of our guidelines.

We can mark some rules as `recommended: false` in our guidelines so they're not activated by default on `extends` or when directly running the guidelines ruleset. For example, we may have an experimental rule verifying the vocabulary used in the API design. We can activate it with `vocabulary-experimental: true`.

We may need rules specific to the API we design that wouldn't make sense in our shared guidelines, such as a specific message signature mechanism. We can add them under `rules` (or have them in another ruleset and use `extends`).

18.9.2 Ignoring certain problems

Not all “problems” detected by our API linter are meant to be solved; some may be recommendations we don't need (section 18.7.1). As shown in figure 18.23, we can ask Spectral to ignore problems using the `overrides` list. Under `files`, we indicate the OpenAPI document filename (patterns like `**/*.yaml` are accepted) followed by a JSON Pointer to the problem. Under `rules`, we indicate the rule(s) to ignore at the locations indicated in `files`. Here, we ignore a problem detected by `search-filter-q-defined` at `#/paths/~1accounts/get/parameters` in the `openapi.yaml` file. The JSON Pointer is based on the original dotted path shown in the Spectral CLI output `path./accounts.get.parameters`. A leading `#` is added, the `.` is replaced with `/`, and the `/` special character is replaced by `~1`.

Summary

- Linting is the process of analyzing source code with a program (linter) for errors or style problems. API linting involves analyzing an OpenAPI document with an API linter to detect API design and OpenAPI authoring problems (and prevent avoidable later breaking changes); use it to automate API design guidelines.
- Spectral is an API linter. I recommend using it because it automates a significant part of the guidelines, customizes checks, reuses elements, and returns helpful problem-solving feedback. It also allows for organizing rules in groups and sharing them, and it lets you tweak rules and ignore specific problems during linting. Alternatives must cover the same minimum requirements.
- In each Spectral rule under `rules`, set one or more JSONPaths in `given` to find the elements to check with one or more of `then.function`.
- Add a `documentationUrl` pointing to the appropriate section of the API design guidelines to create only valid and needed rules.
- Use guidelines' smaller statements and shared OpenAPI components to create meaningful linting rules; combine rules to cover broader topics.

- Ensure appropriate rule granularity with a concise name and description.
- Target any location in the OpenAPI document with JSONPath's `$` (root), `a.b` ("b" of "a"), `a.*` (all "a" values), `..b` (all "b" of the document), `[a,b]` ("a" or "b"), and `a[?(conditions)]` (elements of "a" matching JavaScript-like conditions).
- To limit duplication and errors, define reusable JSONPaths under aliases, use them with `#AliasName`, and extend them with `#AliasName.some.jsonpath`.
- Combine a resource type (based on path) and HTTP method to target typical operations.
- Check atomic values using Spectral core functions such as `pattern`, `casing`, `falsy`, and `truthy` (naming conventions or required flags, for example).
- Use `then.field: "@key"` to check object keys (data model property names, for example).
- Use `then.field: name` and the defined function to check whether name exists (500 response, for example).
- Use the `schema` function with the `contains` and `const` JSON Schema keywords to check whether an item is defined in an array (parameter, for example).
- Use the `undefined` function to ensure that an element doesn't exist (404 when there is no path parameter, for example).
- Use `resolved: false` to check whether references to shared or local OpenAPI components are used.
- Use the `schema` function with a JSON schema of JSON schema to ensure that a data model applies a pattern (search response data model, for example).
- Use Spectral custom functions to perform cross-element checks (request versus response, for example).
- Use the rule `severity` to indicate an actual or possible error or improvement or whether further investigation is needed.
- Return a problem-solving message, possibly using `{{placeholders}}`.
- Organize rules in different groups; use `extends` to import Spectral rulesets with a relative or absolute filename or URL.
- To design an API, define a `.spectral.yaml` file near the OpenAPI document to pull and customize shared guidelines with `extends`, add API-specific rules, and ignore problems that don't need to be solved with `overrides`.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 18.1

Write a Spectral rule that enforces semantic versioning, including the document version (1.0.0, for example) for the API version in an OpenAPI document. Ensure that a user will be able to fix the problem. The following listing shows a faulty OpenAPI document.

Listing 18.1 Faulty OpenAPI document

```
openapi: 3.1.0

info:
  title: Banking
  version: athena-1.0.0.bugfix

paths: {}
```



Not semantic version

Exercise 18.2

Some search operations may be represented with a POST `/resources/search` operation instead of GET `/resources` to work around query parameter limitations (sensitive or overly complete data). Write a rule to check that such an operation has no query parameters. The following listing shows an example of a faulty document (but it may not show all possible errors).

Listing 18.2 Faulty OpenAPI document

```
openapi: 3.1.0
info: ...
paths:
  /owners/search:
    post:
      parameters:
        - name: name
          in: query
          schema:
            type: string
      responses:
        "200":
          description: Found owners
```

Exercise 18.3

Every operation in any of our APIs must return a standard 503 error to notify consumers that the operation or the API is temporarily unavailable. The response body contains an object with two properties: a required string `message` and an optional ISO 8601 `endTime`. Write a Spectral rule that *efficiently* enforces this.

Exercise 18.4

Write a rule ensuring that all 201 response schemas define a required string or integer `id` property, among other properties.

Exercise 18.5

How can you verify which elements are targeted by the `$.paths[?(@property.match(/}$/))].*.responses[?(@property.match(/^(4|5)/))]` JSONPath in an OpenAPI document?

19

Enriching API design artifacts

This chapter covers

- Adding an overview of the API, concepts, and use cases
- Sharpening data models
- Illustrating data, operations, and flows with examples
- Enhancing or adapting artifacts for implementers
- Considering a simulated or a prototyped API
- Considering API functional tests

Artifacts such as the filled API Capabilities Canvas (refining initial user needs), the OpenAPI document (describing the API), and our API design guidelines, including the OpenAPI library and linting rules (detailing the API's standard look and behavior), help us design, discuss, review, and document an API that matches expectations. These artifacts form the core of an “API design reference kit” that serves as support and a deliverable of the API design process and is helpful across the entire API lifecycle.

In addition to supporting us during design, the API design reference kit is essential to help developers (who may be us) implement, test, and deploy the API

accurately and efficiently. This kit can also be a base for the elements available on a public API portal to help third-party developers create applications that consume the API. In the case of private APIs, the API design reference kit will likely be the only resource available for developers (who may also be us) using the API.

The API design reference kit we've crafted is already solid. We may want to enrich it to facilitate the design process or the next steps of the API lifecycle. We can enhance the available information to facilitate understanding or describe the API more precisely. We can also consider artifacts beyond textual or machine-readable descriptions, such as simulated or prototyped APIs or functional tests verifying that the coded API matches the design.

This chapter first provides an overview of the content and usage of an API design reference kit throughout the API lifecycle and lists what we already have and the possible enhancements we can consider. Then we describe the possible enhancements, illustrating them by using OpenAPI and JSON Schema; the chapter also mentions complementary formats and tools when relevant.

19.1 *Crafting an API design reference kit*

An API design reference kit contains artifacts and information that fully describe an API (the interface). It primarily supports the API design process and is an essential input for the development and testing of the API implementation. It can also be beneficial in other stages of the API lifecycle.

Figure 19.1 shows the API design reference kit as the final deliverable of the design process and its final step: “Enrich the API design artifacts.” But this final step occurs parallel to the previous steps; we started to craft the API design reference kit at the beginning of the design process and may enhance it any time we need to.

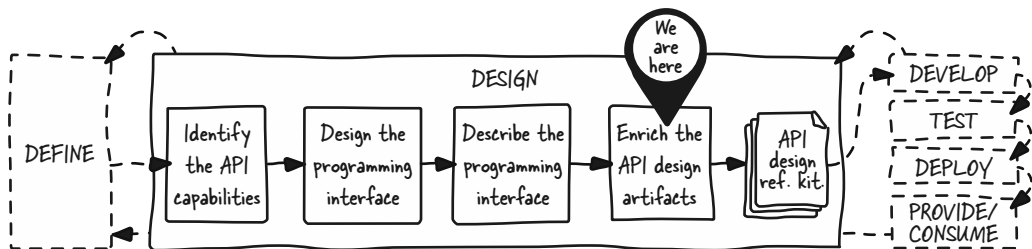


Figure 19.1 Although “Enrich API design artifacts” is the final step in the sequential design process diagram, we work on the API design reference kit across all the steps. Its content is helpful during design and the subsequent API lifecycle stages.

This section describes the content of an API design reference kit and how it can be used throughout the API lifecycle. The section ends by listing what we already have and potential enrichments, which are detailed in the rest of the chapter.

19.1.1 What an API design reference kit can contain

A complete API design reference kit contains the information and artifacts illustrated in figure 19.2. We analyze the initial user needs from the Define stage to identify users, use cases, and operations via the API Capabilities Canvas. From it, we identify the concepts the API deals with to create the REST resources, detailing HTTP operations' data and behavior in the OpenAPI document. Operation flows are outlined through use cases in the API Capabilities Canvas, with the option to add them to OpenAPI (section 19.2). Security considerations are integrated into the OpenAPI document by using scopes and clarifying the handling of sensitive data.

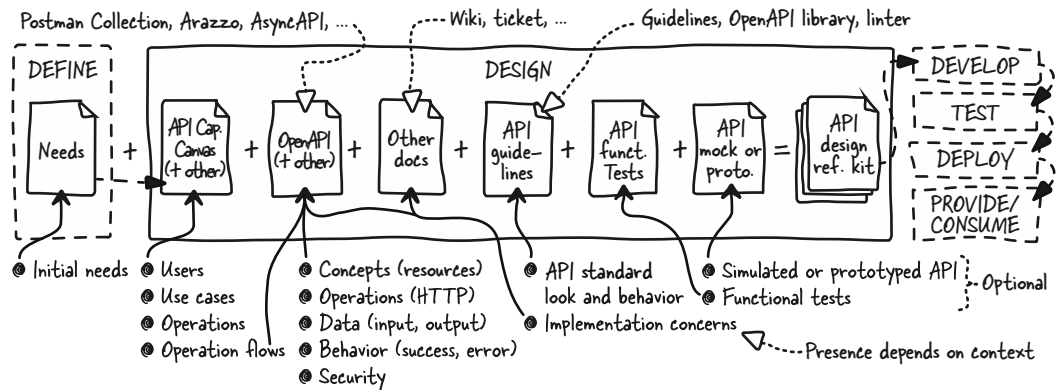


Figure 19.2 The API design reference kit contains all the artifacts necessary to describe an API and help understand it fully. Artifacts can be documents (such as a wiki page or guidelines), machine-readable documents (such as OpenAPI), applications (such as a simulated API or linter), or pieces of code (such as tests).

This information defines the API from an external perspective but doesn't address what happens behind it. Sometimes we collect details about the future implementation, like the effect of existing systems or the source of specific data, which can be in separate documents or in the OpenAPI document.

To design our API, we use our API design guidelines, which include our OpenAPI library and linting rules. They describe our API's general look and behavior.

Sometimes, we may consider creating an application that simulates (a "mock") or is a prototype of the yet-to-be-developed API. We may also want to create functional tests to validate that an application exposing an API implements specific areas or the entire API design correctly.



NOTE Adapt the API design reference kit's content to your context. Ensure all essential information is present, and use formats familiar to the people using the kit and compatible with their tools.

This book focuses on the API Capabilities of Canvas and OpenAPI, but we can use additional and alternative methodologies and formats. AsyncAPI can replace OpenAPI for asynchronous APIs (section 6.1.1). The Arazzo format from the OpenAPI initiative can help formalize HTTP operation flows (www.openapis.org/arazzo). We can also use Postman (<https://postman.com>), a widely used API tool, and its Postman Collections to describe operation flows or for testing.

19.1.2 Using the kit to design the API

During the design stage of the API lifecycle, using the input needs, API Capabilities Canvas, and OpenAPI document, we and other stakeholders can understand, discuss, and evaluate whether the API design matches those needs. We use the API design guidelines to help us design the API, fostering aspects such as usability, interoperability, evolvability, and performance. Playing with the API, we can design via a simulation (called a *mock*) that exposes static data, or a more dynamic implementation prototype can help us see what's missing or the possible improvements. It also enables early consumer prototyping to better validate the design.

19.1.3 Using the kit to develop the API

The API Capabilities Canvas, OpenAPI document, and implementation notes provide developers with a clear vision for accurate API coding, outlining expected operations, data, behavior, and already-known implementation concerns. These elements also help developers implement accurate security, ensuring that consumers and intermediaries see and do only what they should. Developers can use the OpenAPI document to generate implementation code or JSON schemas to validate inputs. However, remember that an implementation should return exhaustive errors, not just basic validation failures (see section 9.8.5). Design guidelines clarify standard operations and behaviors, aiding in code centralization and standardization.

19.1.4 Using the kit to test the API

Developers, QA engineers, and security experts can use the API Capabilities Canvas and OpenAPI document to create tests. OpenAPI's and JSON Schema's machine readability allows them to partially create tests automatically, although such a process will not cover all cases. The kit's ready-to-use functional tests, created by an API designer with deep knowledge of expectations, can provide quick and accurate feedback on development. However, developers must add more detailed tests to cover their implementation entirely.

19.1.5 Using the kit to deploy the API

The OpenAPI document initially created during design contains information that can be used during API deployment. An API gateway can use the JSON schema and security information. An API gateway will typically check security scopes to grant access to operations. However, be mindful that an API gateway that checks requests for validity using JSON Schema or OpenAPI information before sending them to the

implementation may undermine the implementation's efforts to return exhaustive and user-friendly errors (section 9.8.5).

19.1.6 Using the kit to provide and consume the API

When providing and consuming the API, the API design reference kit will be used differently depending on whether the API is public (or partner) or private. The API design reference kit is one of the inputs for technical writers when creating resources for a public or partner API portal that third parties will use to consume the API. A typical portal includes detailed, user-friendly reference API documentation and OpenAPI documents describing all APIs, operations, data, and errors (based on the kit's OpenAPI documents and guidelines), user guides explaining how the API(s) behave from a general perspective (based on the guidelines), and why and how to achieve specific goals (based on the API Capabilities Canvas). The portal may offer a playground or demo environment, potentially based on the mock or prototype of the kit.

Internal developers consuming a private API will likely rely on its raw API design reference kit. The internal API portal can provide reference documentation generated from the OpenAPI document; we can add links to the API capabilities and guidelines for more information. A static API mock generated from the OpenAPI document can facilitate development and testing. Although far from being as polished or comprehensive as public API portal resources, the API design reference kit is a valuable resource, especially with the low-hanging-fruit enhancements discussed in this chapter (and in any case, it's this kit or nothing).



TIP I recommend including Postman Collections in the final public (and also private) API documentation (section 19.1.1). They can provide examples of requests, responses, and sequences of API calls, which can help users quickly achieve what they want with an API.

19.1.7 What we already have and what we may want to add

The artifacts we already created (API Capabilities Canvas, OpenAPI document, and guidelines) constitute a solid API design reference kit; we could stop here. We may want to enhance these items or add new ones to facilitate designing, implementing, developing, testing, deploying, providing, and consuming the API. However, we must tailor our efforts to the context and avoid unnecessarily overworking the API design reference kit. The rest of this chapter discusses the following enrichments (quick wins and those requiring more effort) that we can consider, along with their benefits:

- Providing an overview of the API with OpenAPI
- Enhancing the precision of data models with JSON Schema
- Providing examples to illustrate data and operations
- Enhancing and adapting OpenAPI for implementers
- Considering API mocking or prototyping
- Considering API functional testing

19.2 Providing an overview of the API design with OpenAPI

A good public API portal provides users with a clear overview of the API's objectives, concepts, and use cases, helping potential users determine its fit for their needs; current users identify what they need and have all the links to go further. We should consider this perspective in the context of an API design reference kit. Such an overview with easy access to documents can be helpful to ensure that the design aligns with initial requirements. It can help implementation developers, future consumers of our private API, and tech writers who create resources for our public API. We won't try to match the quality of a public API portal; that's not our job. But we can provide a decent overview and make the OpenAPI document a central hub.

19.2.1 Adding links to other artifacts and describing the API

As illustrated in figure 19.3, we can add a summary (`info.summary`) or longer description (`info.description`) to the OpenAPI document if we feel the name is insufficient to capture the essence of the API (don't overwork this before reading section 19.2.2). The description field supports markdown so that we can add a Links section with a list of links to other artifacts related to the API. If the OpenAPI document is not the hub for all artifacts, we can use the root `externalDocs` object to link to the page hosting all the API documentation.

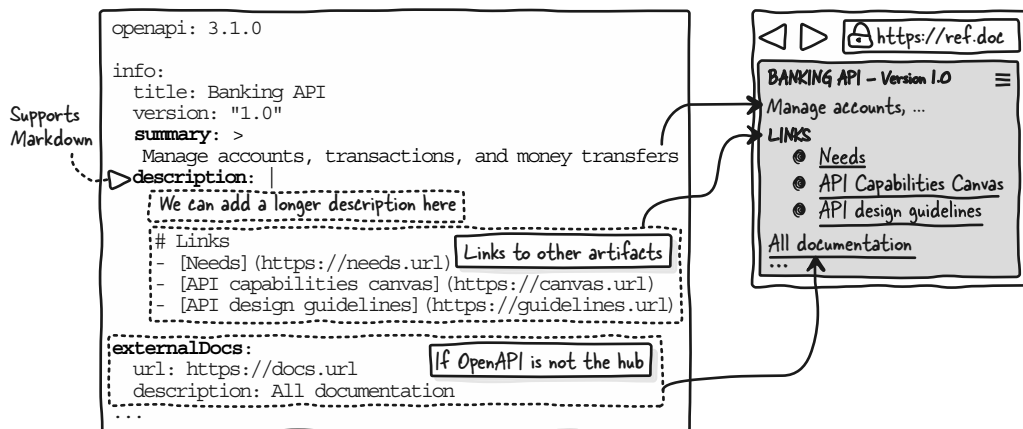


Figure 19.3 Add an `info.summary` if necessary for understanding. Use Markdown in `info.description` or the root `externalDocs.url` to link the OpenAPI document to other artifacts.



TIP Most OpenAPI objects support description and `externalDocs`. We can add links to API design guidelines on each component defined in our OpenAPI library.

19.2.2 Organizing operations around concepts and use cases

Tools show the operations of an OpenAPI document as a (possibly long) flat list, ordered as defined. This may not facilitate understanding the API, the concepts it deals with, or what can be achieved with it. People interested in the API may look at the API Capabilities Canvas. But as shown in figure 19.4, we can also use OpenAPI tags to group operations by concepts and use cases and provide a good API overview at a glance of a rendering of the OpenAPI document.

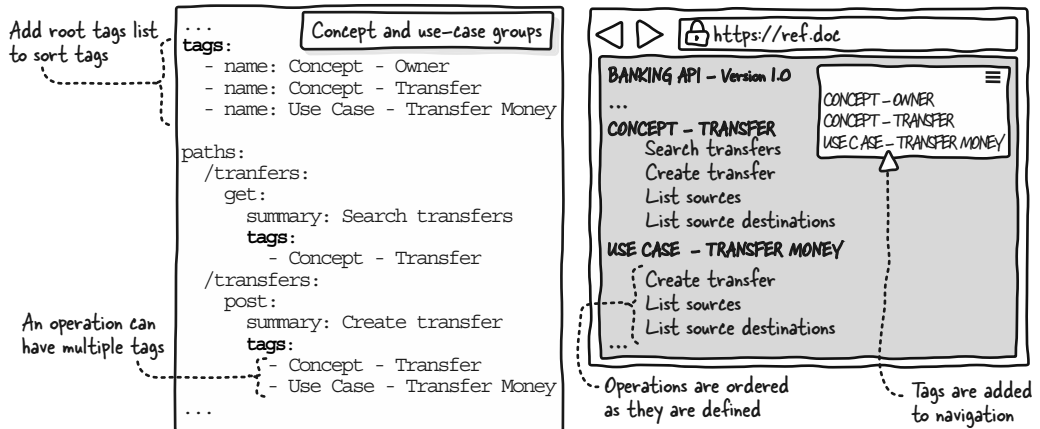


Figure 19.4 Group operations by adding tags. Sort groups by adding the root-level tags list. Operations can't be reordered in groups.

An OpenAPI operation object can have a `tags` list containing one or more tag names that we can choose freely. Tools using OpenAPI add the tag names to the navigation, which is convenient for providing an overview of the API.

Although an operation's resource can be guessed by its URL, it's not always obvious. Additionally, we may want to highlight only high-level concepts related to different resources. For example, in the OpenAPI document of the Banking API, we add the operations dealing with transfers, sources, and destinations under the same "Concept - Transfer" tag. By default, tools order tags as they are defined. We can change this order by adding a root-level `tags` list. At this level, the items are objects with a `name` property holding the tag name.

We can proceed similarly for use cases and create a tag for each one listed in the API Capabilities Canvas. It's not perfect, as the operations under a tag can't be reordered to reflect the use-case steps, but it's still effective for giving an idea of the API use cases and what's used in a specific use case; check section 19.2.3 for a possible workaround.

19.2.3 Describing use cases

We may want to enhance the use-case tags (root level) to facilitate access to information if the API design reference kit is the only documentation the users of our private API will have. We can add a direct link to the appropriate location in the API Capabilities Canvas (using `externalDocs`; figure 19.5) or a Markdown description (with an ordered list of operations, sequence diagram image, or table copied from the canvas; figure 19.6).

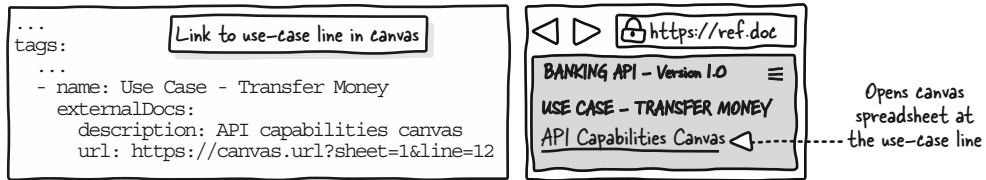


Figure 19.5 By using a tag's `externalDocs`, we can add a link to the sheet and line corresponding to the use case in the API Capabilities Canvas.

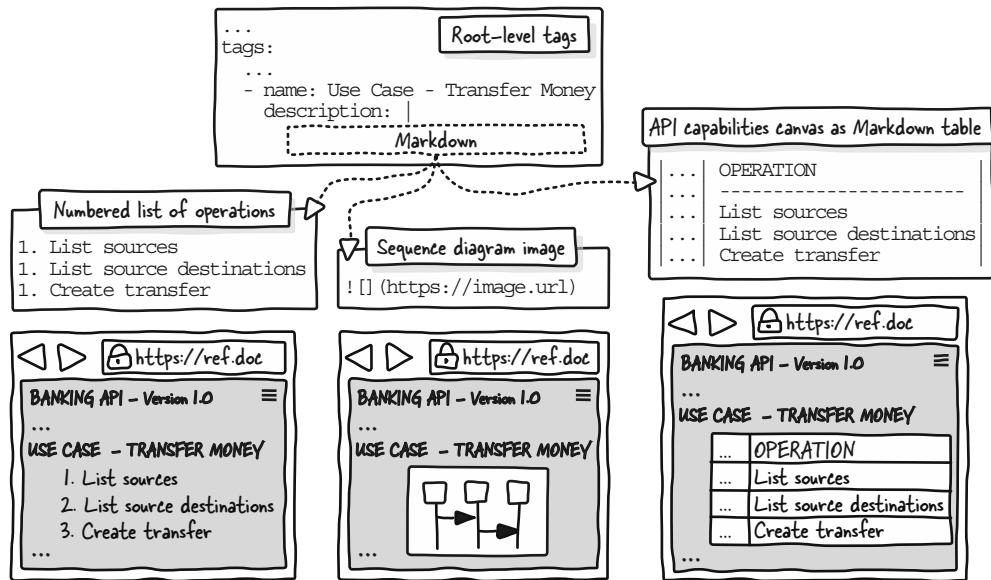


Figure 19.6 The root tag object supports a Markdown description. We can use it to describe the order of the operations, but that may be a lot of work.

Tag descriptions can help create nice private API documentation in a standard portal coming out of the box for an API gateway. Nevertheless, describing use cases in OpenAPI can be a lot of work because the format is not made for this. Instead, we can

keep the tags without further information and consider creating a Postman Collection from the tagged OpenAPI document (check the Postman documentation). This will be more helpful because the format supports examples (see section 19.4), and users will be able to test the requests once the API is available. Alternatively, we may want to use the Arrazzo format if our tooling supports it.

19.3 Enhancing the precision of data models with JSON Schema

Adding more machine-readable details to our data models can be beneficial when using documentation, code generation, testing, or mocking tools. In section 7.4, we described properties with types and optional descriptions, such as “amount > 0” and “currency is USD or EUR.” However, only AI-powered JSON Schema tools can use such a textual description. Fortunately, JSON Schema has keywords that make our data models more precise and usable by non-AI-powered tools. This section covers these typical cases:

- Number ranges
- Array size ranges
- String length ranges
- String regular expressions
- Enumerations
- Default values



NOTE Stay focused on identifying data during the first pass of data modeling; use textual description, and keep such JSON Schema details for a second pass if you aren’t yet at ease with JSON Schema. Also, refer to the documentation for less common but convenient keywords (www.learnjsonschema.com). Consider exploring JSON Schema’s logical keywords (`if`, `else`, `not`, `dependentSchema`, and `dependentRequired`), but verify compatibility with your tools.

19.3.1 Describing a number or element size range

Figure 19.7 illustrates defining numeric ranges or setting array or string lengths. Defining such limits may be interesting for various reasons, such as subject-matter concerns (“an account owner must be 12 years old or more”), efficiency concerns (“a Search transactions response returns 100 elements per page maximum”), subject-matter concerns, and database limits (“A name can’t be empty and must be 200 characters long maximum”).

The range of a number (integer or float) is defined with `minimum` (greater than or equal), `exclusiveMinimum` (greater than), `maximum` (less than or equal), or `exclusiveMaximum` (less than); a range can be open-ended. Suppose an owner’s age must be greater than or equal to 12 and less than 100. In that case, we indicate `minimum: 12`, `exclusiveMaximum: 100`. If the amount of a wire transfer is greater than 0 and there is no limit, we only add `exclusiveMinimum: 0`.

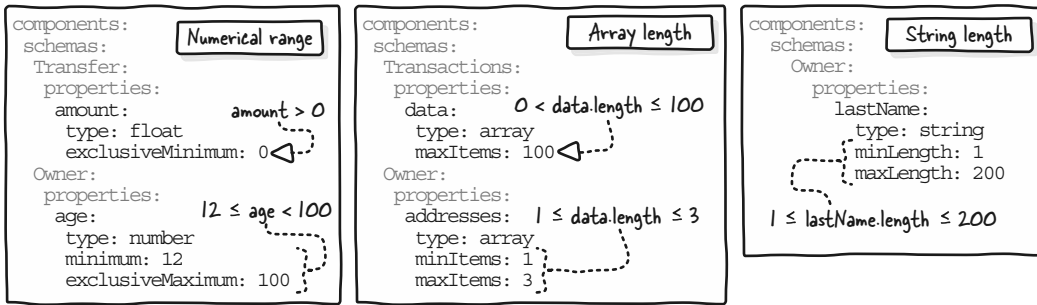


Figure 19.7 JSON Schema allows us to define numerical ranges and array and string lengths. These ranges can be open-ended.

The length of an array is defined with `minItems` and `maxItems`. If the generic data array returned when searching for transactions can contain up to 100 transactions, we add `maxItems: 100` to its definition. If an `addresses` array can contain between one and three addresses, we add `minItems: 1` and `maxItems: 3`.

The length of a string is defined with `minLength` and `maxLength`. If an owner `lastName` must contain at least 1 character and at most 200, we add `minLength: 1` and `maxLength: 200` to its definition.



NOTE If a range or size depends on business or configuration logic, you must still rely on a textual description for it. You may consider adding an operation to get such dynamic range values and limit the risk of consumer errors (section 10.3.5).

19.3.2 Describing a value with pattern, enum, and default

Figure 19.8 illustrates defining a string regex, an enumeration, and a default value with JSON Schema. We can indicate that a string matches a specific regular expression with `pattern`. We may want to add a type to all our IDs, allowing us to determine what it identifies by glancing at its value. For example, an account ID must be a string starting with `acc-` followed by at least one digit, such as `acc-12345`. We can indicate this in the JSON Schema with `pattern: "acc-[0-9]+"` (`[0-9]` indicates a character from 0 to 9, and `+` means “1 or more”; check <https://regex101.com> to master regular expressions).

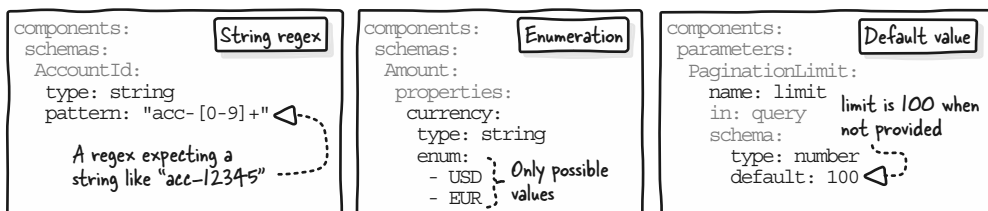


Figure 19.8 JSON Schema can describe a value with pattern (regex), enum (authorized values), and default (default value).



CAUTION Use ID prefixing only with new custom IDs that don't rely on existing standards (local to your organization or international). Modifying existing IDs shared among systems would reduce interoperability.

If a value belongs to an enumeration, which is a known and finished set of values, we can define it with the `enum` array. Values in `enum` can be anything (strings, numbers, or even objects). If currency can be only USD or EUR (or 1 or 3), we can add `enum: [USD, EUR]` (or `[1, 3]`).

We have learned to require minimal data and use default values. If the default value is static, we can use `default` to indicate it. The `limit` query parameter is optional and defines the number of elements per page in a search operation. When not provided, its value is set to 100; we can indicate `default: 100` in its schema.



NOTE If an enumeration is dynamic or changes often, don't use JSON Schema's `enum`; consider adding an operation that delivers the possible values (section 10.3.5).

19.4 Providing examples to illustrate data and operations

The OpenAPI document comprehensively outlines the operations' data. However, tangible data samples and complete examples of requests and responses may sometimes be necessary to provide a better understanding. API documentation tools can generate samples from OpenAPI, and these can be improved with detailed JSON schemas (section 18.3). Nonetheless, generated samples often include unhelpful placeholders like "string" and "lorem ipsum". AI-powered tools can enhance quality, but inaccuracies can still occur. We can provide partial or complete examples with JSON Schema and OpenAPI for a clearer understanding when necessary. This section discusses

- Adding property examples with JSON Schema
- Adding examples of parameters, request and response bodies, and headers with OpenAPI
- Authoring accurate and realistic examples
- Reusing OpenAPI examples
- Connecting examples to each other

Example-based API design process

You may want to use an example-based design process, depending on your context. After you analyze your users' needs, draft operation requests and responses (using all you've learned in this book and a Postman Collection or similar) to support the design discussions. Example-based modeling sacrifices having an independent formal source of truth (which you may be OK with) and omits details, such as required versus optional elements (which can be clarified during implementation). Using a linter is still possible; you can generate an OpenAPI definition from code, although it may lack completeness compared to the one from this book and won't use components from a shared library. Nonetheless, it can help guide your API design during implementation.

19.4.1 Adding property examples with JSON Schema

We can add examples to clarify pieces of data in JSON schemas. Figure 19.9 shows how to add an example value or examples list. Documentation tools use them to enhance data samples and may show them in the schema details.

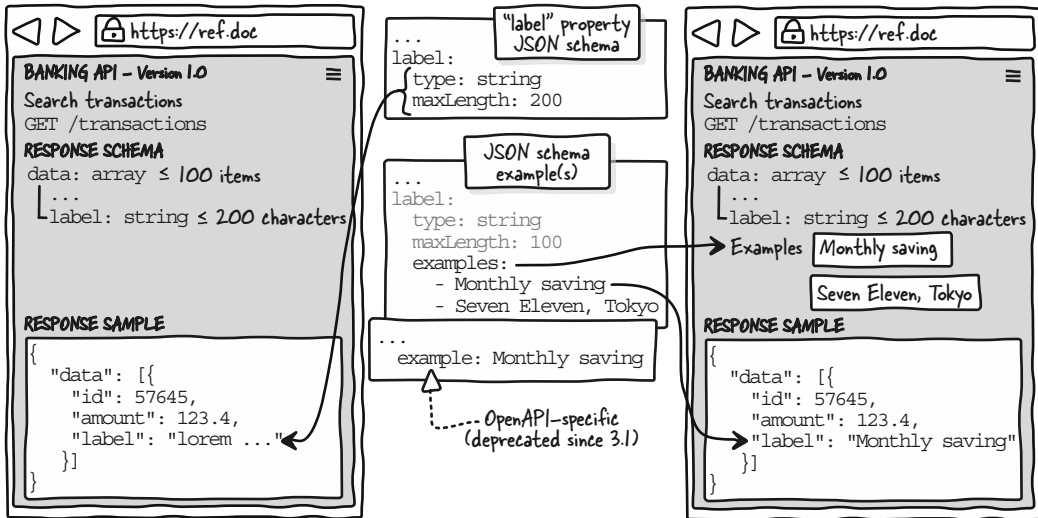


Figure 19.9 To better describe data, add a single example or a list of examples to any part of a schema. It will improve the schema documentation and generated samples.

If a Transaction JSON schema has a string `label` property with `maxLength` set to 200, it doesn't clearly describe what such a value looks like. A sample generated by a documentation tool will likely be "string" or "lorem ..." (with a length between 0 and 200), which doesn't help. To clarify this data value, we can add `example: Monthly Saving` or `examples: ["Monthly saving", "Seven Eleven, Tokyo"]` to its schema. The value (or values) may be shown by documentation tools in the schema details of the `label` property. The initial "lorem ..." value is replaced by the example value or the first value of `examples` in the samples.



CAUTION Check the compatibility of your tools. The `example` key has been deprecated since OpenAPI 3.1 and is specific to OpenAPI; it doesn't exist in JSON Schema. The OpenAPI documentation recommends using standard JSON Schema `examples` instead. However, some OpenAPI tools may offer limited or unreliable support for `examples` and provide better support for `example`.

19.4.2 Adding examples of parameters, request and response bodies, and headers with OpenAPI

We may need examples to better illustrate a parameter, request or response body, or response header data. Although we could technically use root-level JSON Schema examples (section 19.4.1), I recommend using the OpenAPI examples map. It can be added near the schema of these elements, letting us identify, name, and describe schema examples, and it is well-supported by tools.

In figure 19.10, we added an examples map to the 200 response body of the GET /transactions operation. We added the NoTransaction and Transactions keys to identify two different examples. The minimum required information in an OpenAPI example object is the value, and we can add summary (“No data found”) and description (“Not a 404!”). The value can be in YAML (tools convert it to JSON) or JSON. The summary is often used as a name that documentation tools show on tabs and in lists.

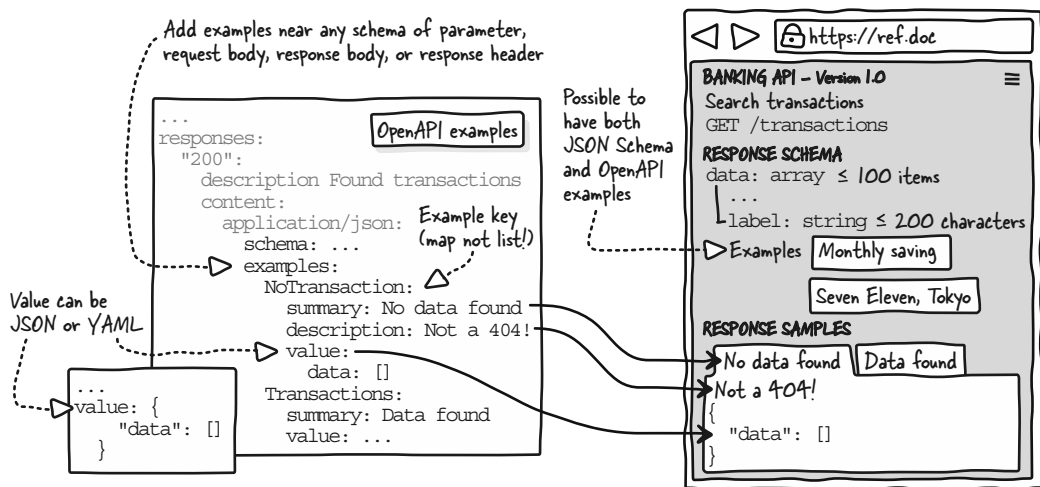


Figure 19.10 Add an examples map near the schema of a parameter, request or response body, or response header to illustrate it better with one or more documented examples.



CAUTION Beware of typos; OpenAPI also provides the `example` field at the same level. It is similar to the `example` from section 19.4.1; it provides only one undocumented value (no key, summary, or description). I do not recommend it because the OpenAPI `examples` field is richer and well-supported by OpenAPI tools. You can define a linting rule to prevent using `example`.

19.4.3 Authoring accurate and realistic examples

Authoring examples, especially bodies, can be cumbersome and error-prone. To easily create request or response body examples, copy the sample the documentation tool generates from your JSON schema, paste it into the OpenAPI document, and tweak it

to your liking. You may also get help from AI when you're out of inspiration and need to add an example for a piece of data or a body. Copy and paste the element JSON Schema into your favorite AI chat, and ask for one or more examples.



TIP I recommend adding the `oas3-valid-media-example` and `oas3-valid-schema-example` Spectral built-in rules to your OpenAPI ruleset to ensure that OpenAPI and JSON Schema examples match the schema of the element they illustrate (section 18.8).

19.4.4 Sharing OpenAPI examples across operations

In most cases, we shouldn't duplicate examples across different operations if we use reusable parameters, responses, and headers. However, that may happen sometimes, typically when applying patterns defined in our guidelines. OpenAPI can prevent duplication; it supports defining reusable OpenAPI examples under `components.examples`, as illustrated in figure 19.11.



NOTE The OpenAPI example (singular) and the JSON Schema examples and example can't use reusable examples.

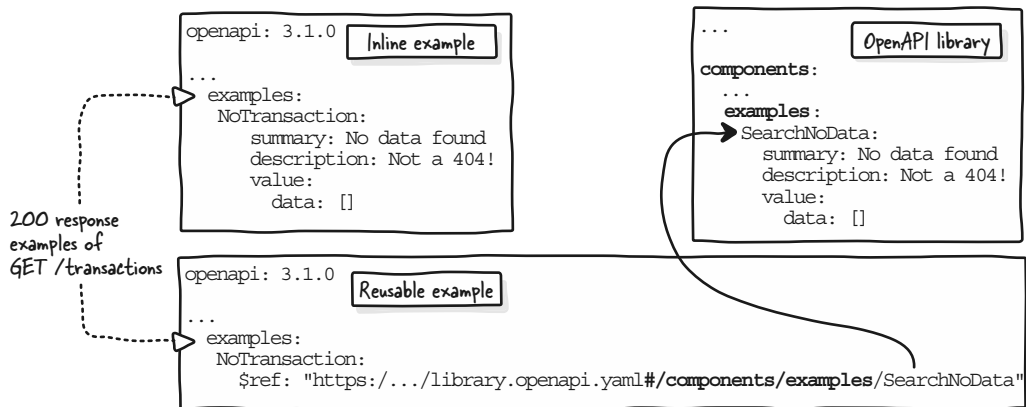


Figure 19.11 Define examples under `components.examples`, and use them with a `$ref` to avoid duplication.

Suppose all our search operations must return 200 OK with an empty data list. We can't have a shared 200 response in our library because the data schema differs from one operation to another; the only common part across operations is the empty data when nothing is found. To illustrate this pattern, we could add a `NoTransaction`, `NoAccount`, or `NoOwner` example in the 200 responses' examples of the search transactions, accounts, and owners operations. But instead of duplicating the same example, we can define a `SearchNoData` example under `components.examples` in our OpenAPI library (section 17.6) and then reference it with `$ref: https://guidelines.intra/`

`library.openapi.yaml/#components/examples/SearchNoData`. This technique also works with examples defined locally.



TIP We can create a linting rule to enforce using this shared example in the 200 response of all search operations, reminding us that a search operation doesn't return 404 when it finds no element.

19.4.5 Connecting examples to each other

In some cases, consistent examples covering an operation's entire request and response can help us better grasp an operation's data. We can define OpenAPI examples for multiple elements that share the same key and summary when needed. Figure 19.12 shows that the request and response bodies of the POST `/transfers` operation can have examples identified with the `Success` key and "Success" summary. Readers of documentation generated from OpenAPI can easily connect them, whether the tool shows the key or summary.

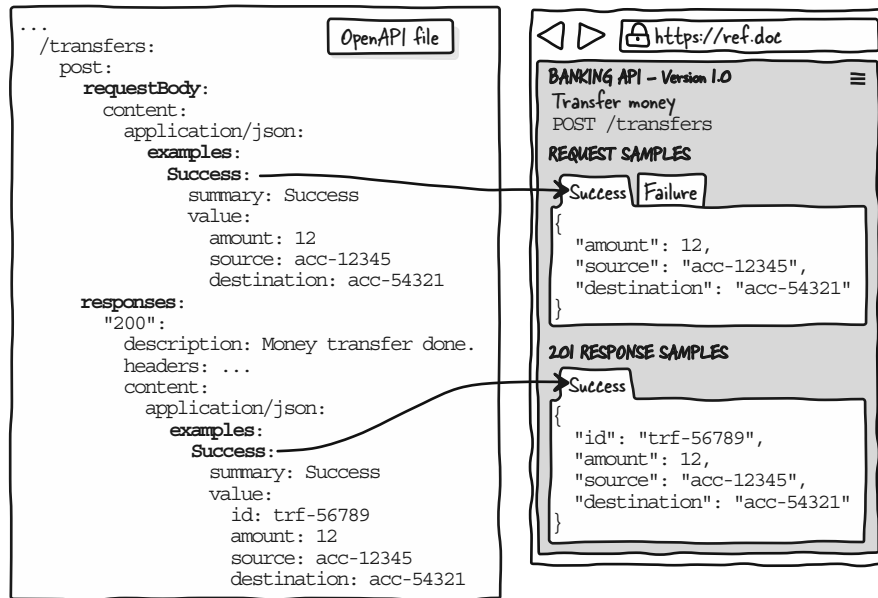


Figure 19.12 Use the same example key (for tools) and summary (for documentation readers) to enable connecting examples.

But because request and response example data is separated in OpenAPI, defining and managing numerous examples can become complex. We may want to use an alternative to OpenAPI to store many complete examples. A Postman Collection is the typical format; it lets us define complete examples that cover all input and output data, including HTTP status. Complete sets of request-plus-response examples can be helpful for simulated or mocked APIs (section 19.6).



CAUTION Think carefully before crafting examples extensively and exhaustively during design (whatever the format used). We’re not creating the public API portal documentation but facilitating understanding of the API in the design context. Complete sets of realistic examples can be easily created for the final documentation or simulated from actual requests and responses after implementation. Complete examples also make sense to illustrate API design guidelines.

19.5 *Enhancing and adapting artifacts for implementers*

We’re designing an API to be implemented, and we must ensure that implementation developers (who may be us) have all the necessary information and can efficiently use the API design reference kit. Fortunately, that was mostly true before we began this chapter. The kit may grow better with the enhancements discussed in this chapter, but it was already in solid shape to help developers implement the API (section 19.1.3). There are two practices we can consider to facilitate the use of the kit by implementers:

- Embedding implementation notes in artifacts
- Enhancing or adapting OpenAPI for code generation

19.5.1 *Embedding implementation notes in artifacts*

During API design, we can gather implementer-specific information, such as API-specific details and generic implementation concerns. Instead of recording them in separate documents, we can add them to OpenAPI or our guidelines to centralize information access.

API-specific implementation notes, such as “Use label instead of the description column in the TRA02 table” for a transaction’s label property and “Sort the owners list alphabetically in the Account model,” can be added to our OpenAPI document. A straightforward method is to include them in the element’s description. We can put them under ##### Implementation notes to separate API contract and implementation information. The level-5 Markdown heading avoids cluttering the schema UI; adjust it based on your tooling. Alternatively, OpenAPI’s extensibility allows `x-something` properties, arrays, and objects to be included nearly anywhere in an OpenAPI document; regular parsers will ignore them, although some documentation tools may display them. If supported, we could add `x-implementation-notes: Use label instead of the description column in TRA02 table` to the label property’s schema.

Generic implementation details, such as hints about implementing cursor pagination, must be included in the relevant section of the design guidelines, possibly by adding a link to detailed implementation guidelines or the part of a development framework to use (section 16.4.5). These generic details should also be integrated into the related OpenAPI library components for quick access (using the same technique described earlier or the `externalDocs` URL from section 19.2.1).



CAUTION Such information may be OK in private API documentation, but not for partner or public APIs used by third parties. People in charge must

remember to remove implementation details if the kit is the basis for the final API documentation (section 19.1.6). Command-line tools such as `jq` and `yq` (search for `jq` on my blog at <https://apihandyman.io>) or a few lines of code using `JSONPath` can easily clean an OpenAPI document of `x-implementation-notes` or `##### Implementation notes` sections in descriptions.

19.5.2 Enhancing or adapting OpenAPI for code generation

Implementation developers may generate code from the OpenAPI document, facilitating development. If code generation is an important part of our workflow when creating APIs, we should ensure the smooth use of our OpenAPI document. We can add tags and operation identifiers and may want to limit how we use JSON Schema.



NOTE OpenAPI-based code generation can also help create tests (section 18.7) or develop consuming applications.

Code generators may let us use tags to group operations in classes, modules, and files. Specific tags can be selected or ignored (like the concepts and use-case tags from section 19.2.2).

Code generators can infer function names using operations' HTTP methods and resource paths. However, the result may not always be user-friendly for paths with multiple segments or consistent across generators. If that's a problem, we can set an `operationId` on each operation; generators use it as the function name. For example, we can add `operationId: searchTransfers` to the `GET /transfers` operation.

Code generators may not handle all JSON schema keywords effectively, erroring or resulting in convoluted code. Polymorphism-related keywords like `allOf` (section 17.2.6), which combine multiple schemas, may cause problems. To ensure smooth code generation, we can avoid such keywords, but this could affect the accuracy or organization of our OpenAPI document.



TIP Add the `operation-operationId` and `operation-operationId-unique` built-in `oas Spectral` rules to your OpenAPI ruleset (section 18.8). Doing so ensures that operation identifiers are always defined and unique. Optionally, create custom rules to enforce adding details that aid code generation (section 18.3) or prevent keywords unsupported by code-generation tools or other tools.

19.6 Considering API mocking or prototyping during API design

During API design, we may consider creating an API mock or an implementation prototype that simulates the yet-to-be-developed API. Playing with such a simulation can help us and other stakeholders evaluate our design (section 19.1.2). We can easily create a basic mock based on our OpenAPI document when needed. For more complex needs, I usually recommend an early implementation prototype.

19.6.1 Creating a basic mock with OpenAPI

A basic API mock simulates an API with static data. Creating one using our OpenAPI document without configuration is easy. Search for OpenAPI-based mocking tools: Microcks (<https://microcks.io>) and Postman (<https://postman.com>) are my favorites. Such tools can return data on a `GET /accounts` request defined in the OpenAPI document based on the JSON schema (the more detailed, the better; check section 19.3) or examples (section 19.4). Some tools support input validation based on OpenAPI and JSON Schema, but error responses may not match what is expected.

19.6.2 Favoring an early prototype over a complex mock during design

API mock tools can enable more realistic behaviors. Templating allows mocks to return enhanced input data with random IDs or dates for a `POST /transfers` request. Scripting allows a delayed or immediate transfer type to be returned based on the presence and value of an input date. Mock tools, such as Microcks, enable us to save data between requests, so `GET /transfers/123` retrieves data from a prior `POST /transfers` call that returned the 123 ID.

Nevertheless, spending time on complex mocks with heavy configuration and scripting during API design should be approached cautiously. Complete mocks are extremely useful later: internal developers can use them to build their applications in isolation, and public API users can experiment in a demo environment based on such mocks. But our focus is on API design, not creating all potential final artifacts useful for the entire API lifecycle.

If realistic behavior is needed during design, consider the early implementation of a prototype instead of a complex mock. Implementation developers (who can be us) can create a basic skeleton that returns static data (generated from the OpenAPI document) and add logic as we refine the design. Once the design is finalized, implementation can continue with adding the remaining code to the prototype. If necessary, others (or ourselves in another role) can create a complete API mock after design and development (to ensure entirely realistic behavior and benefit from sample data from the implementation).

19.7 Considering creating functional API tests during API design

API testing is shown as a single stage in the API lifecycle but encompasses various aspects, such as verifying data, behavior, performance, and security. Different profiles (developer, QA engineer, security expert) may handle these aspects at various times using the API design reference kit (section 19.1.4). Although testing is distinct from design in the lifecycle, we may want to add functional tests that cover specific parts or the entire API to our API design reference kit created during the design stage to help developers ensure accurate API implementation. This section discusses a few typical scenarios.

19.7.1 Clarifying logic

When we create a simple API design, part of the logic behind it may be complicated to implement, even with a thorough OpenAPI, JSON Schema, and descriptive text. In such cases, creating functional tests can clarify expectations and help developers quickly verify their code. We can use our usual testing framework or a Postman Collection, which allows post-request scripting to ensure that the response meets expectations. Section 19.6's mock or prototype can aid in developing these tests. However, such early functional tests will cover only some parts of the implementation; all test stakeholders must add their own.

19.7.2 Smoothing collaboration

Creating tests during design may be unnecessary in a small, cohesive team. However, if design and development are separated due to organizational silos or third-party involvement, including functional tests in the API design reference kit can reduce late-stage errors and improve collaboration. In such cases, a portable test format that operates independently of implementation code, like the widely adopted Postman Collection format, is advantageous.

19.7.3 Designing standard APIs

Designing a private standard API, such as a file-upload or health-check API implemented by different teams, can promote consistency and interoperability. Providing ready-to-use tests helps prevent flawed versions from being re-created in each implementation and streamlines development. I recommend creating a reference implementation to ensure the standard's feasibility and adding it to the kit as an example for other implementations.

When designing an industry-wide API standard for multiple organizations, comprehensive compliance or conformance tests are essential to validate that implementations match the standard (reference implementations are required, too). However, creating such tests is part of the final documentation; this situation is similar to the API documentation of public APIs, which we excluded from the design stage. Thus, this task can be deferred to later in the standard's lifecycle.

Summary

- An API design reference kit contains artifacts and information that fully describe an API and is helpful across the API lifecycle, including the design, development, test, deployment, and provide/consume stages.
- A complete API design reference kit covers initial needs, users, use cases, operations, operation flows, concepts, HTTP operation representations (path, method, data, behavior), security, and implementation concerns.
- A complete API design reference kit includes artifacts like the needs description, the API Capabilities Canvas, an OpenAPI document, design guidelines

(including OpenAPI library and linter), and, optionally, an API simulator (mock or prototype) and functional tests.

- The API Capabilities Canvas, OpenAPI document, and guidelines constitute a solid API design reference kit but can be enhanced to better support design and next steps.
- Measure the effort in API design reference kit enhancements. An API design reference kit isn't meant to be final documentation for public or partner APIs (but supports their creation); the kit is often the only resource for private API consumers.
- Add a list of links in the Markdown-compatible `info.description` to make the OpenAPI document a central hub, or use the `externalDocs` link to reference the API resources central hub.
- To facilitate understanding and review, provide an overview of concepts and use cases by adding `tags` to operations in the OpenAPI document and ordering them with the root-level `tags` list.
- Enhance the precision of JSON schemas with ranges, enumeration, or default values; this is helpful for understanding, code generation, testing, and mocking.
- Provide JSON Schema `example` or `examples` to clarify pieces of data.
- Provide OpenAPI documented `examples` to illustrate parameters, request or response bodies, and response headers.
- Add shared examples to your OpenAPI libraries to avoid duplicating examples for the same design patterns.
- Use the standard OpenAPI `description` or custom OpenAPI extensions to add implementation notes where needed (`x-implementation-notes`, for example).
- Adapt the use of OpenAPI and JSON Schema keywords to better support code generation if it's part of the API creation process.
- Use the OpenAPI document to create a basic mock that simulates the API based on JSON Schema or examples; it can be helpful for thinking about and discussing the API design.
- Consider creating an early implementation prototype for more complex API simulation needs.
- Consider creating functional API tests during design to clarify specific points for implementation developers or limit the risk of late errors when implementation is performed by another team or a third party.

Exercises

This section contains exercises to help you practice some key skills in this chapter. You'll find the solutions in the online appendix (<https://mng.bz/260N>). I encourage you to solve them and read their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Exercise 19.1

Modify the OpenAPI document in listing 19.1 so that an OpenAPI viewer will group operations in a “Book and Author” group based on the data they return. Ensure that the groups are shown in that order.

Listing 19.1 OpenAPI document

```
...
paths:
  /authors:
    get:
      responses:
        "200":
          description: Authors
  /authors/{authorId}/books:
    parameters:
      - $ref: "#/components/parameters/AuthorId"
    get:
      responses:
        "200":
          description: Books
  /books:
    get:
      responses:
        "200":
          description: Books
  /books/{bookId}/authors:
    parameters:
      - $ref: "#/components/parameters/BookId"
    get:
      responses:
        "200":
          description: Author
```

Exercise 19.2

Complete the schemas in listing 19.2 to comply with the following:

- A speed’s `id` starts with `sp-`, followed by three characters from *a* to *z* and at least one number from 0 to 9 (also add an example).
- A speed’s value is greater than 0, and its default value is 0.
- A speed’s unit can be `kph` or `mph` and defaults to `kph`.
- A speed’s direction is greater than or equal to 0 and less than 360.
- A list of speeds contains between 1 and 100 objects.

Additionally, indicate to implementation developers that a speed’s direction comes from the SPD23 column of the ZSPD table.

Listing 19.2 Schemas

```
...
components:
  schemas:
    Speed:
      properties:
        id:
          type: string
        date:
          type: string
          format: date-time
        value:
          type: number
        unit:
          type: string
        direction:
          type: integer
    Speeds:
      type: array
      items:
        $ref: "#/components/schemas/Speed"
```

appendix

Solutions to the exercises

This appendix contains the solutions to the exercises in the book. I encourage you to solve them before reading their solutions, which include detailed explanations, references to relevant sections, and additional comments.

Chapter 2

Solution to exercise 2.1

As seen in section 2.3.1, users can be end users, applications, or their developers. Potential users of an API for an HR tool that manages time-off requests can be

- Employees (who request time off)
- Managers (who approve or deny requests)
- HR administrators (who manage time-off policies and oversee usage)
- Time-off application and its developers (used by employees, managers, and HR administrators)
- Payroll staff (who need time off data for accurate salary calculations)
- Payroll system and its developers (who need to integrate the time-off data)

As seen in section 2.3.1, you should probably prioritize analyzing employees' use cases. Employees are directly involved in creating time-off requests, the core functionality from which everything else flows (approval, policies, payroll integration). As seen in section 2.3.2, analyzing the “Requesting time off” use case is a good start, because it is the functionality that employees use the most. Additionally, it's why we chose to focus on employees. However, as seen in section 2.4.3, if you prioritize analyzing employees' needs, remember not to neglect other users. For example, time off won't be validated without the manager's use cases being fulfilled.

Solution to exercise 2.2

As seen in section 2.3.2, focus on identified user needs when listing use cases. The needs of sales representatives probably won't include "Verify data synchronization processes." However, a system administrator of the CRM may find such a use case useful. You should check with stakeholders to see whether the API should cover this user's needs and this specific use case (section 2.6.1).

Solution to exercise 2.3

The list was missing the "Read menu" step:

- Search restaurant (input: user-defined search filters list of all restaurants handled by the API, outcome: restaurants)
- Read menu (input: a user-selected restaurant from the search results, outcome: menu)
- Add dish to order (input: a user-selected dish from the menu, order handled by the API, outcome: dish in order)
- Pay order (input: order, outcome: validated order)

You can spot the missing step by realizing that you need a menu to select a dish to add to the order or that you're not using the restaurants returned by the search (section 2.3.5). Note that you may also wonder where the order will be delivered. The API may know the customer's address, or you can add an address selection step.

Solution to exercise 2.4

The unique and versatile operations needed to fulfill the identified event management API could be as follows:

- "Search for events" (verify whether the event already exists, and search for available events)
- "Create an event" (create the event)
- "Add tickets to order" (add the tickets to the order, and add the tickets to the gift)
- "Pay order" (pay the order, and validate the gift)

As done in section 2.5.2, you discern the true intent of the steps by analyzing their descriptions, inputs, and outcomes to identify unique operations. Verifying the existence of an event and searching for available events both boil down to finding events (outcomes) that match certain criteria (inputs) among all events (inputs). The reasoning for order- and gift-related steps is similar, although less evident. Discussions with SMEs lead you to conclude that the user adds and pays for tickets in both scenarios. However, when offering event tickets, the event organizer applies a coupon code, resulting in an order with a zero amount. This coupon code input raises another question (section 2.3.5): how do event organizers manage coupon codes?

Solution to exercise 2.5

The fixed steps for the “Return a borrowed book” use case are

- Search for the borrowing record based on book ID.
- Update the borrowing record to indicate that the book has been returned (which updates the user’s account under the hood).

The original use-case steps expose the provider’s business logic and data (section 2.8). If the consumer fails to execute both “Update the borrowing record to indicate that the book has been returned” and “Remove the borrowing record reference from the user’s account,” the library management system data can become corrupted. To prevent this, the consumer should only see a single step to update the borrowing record, which will update both the borrowing record and the user’s account data.

Chapter 3**Solution to exercise 3.1**

Based on the provided operation descriptions and your learning about resources (section 3.3.2), relations (section 3.3.4), and actions (section 3.4.1), you can identify the following resources and actions:

- 1 Resource: courses, action: set up
- 2 Resource: courses, action: search
- 3 Resource: course, action: verify
- 4 Resource: course, action: modify

For the relations, you can say that the “courses” resource contains many “course” resources.

Solution to exercise 3.2

Based on the provided information and your learnings about resources (section 3.3.2), operation inputs (section 3.4.2), and how to separate resources and inputs (section 3.4.3), the operation resources and inputs are as follows:

- Search for flights: the resource is flights, and the inputs are airport, destination, departure date, date, and airline.
- Book a flight: the resource is flights, and the inputs are the flight (probably a flight number), date, and passenger (a reference or all its information, such as name, passport number, etc.).

Solution to exercise 3.3

Based on what you learned in section 3.4.4, you can see that the outputs of “View meal plan logs and sleep logs” and “List workout history and nutrition advice logs” combine heterogeneous fitness and wellness concepts in a list. These operations should probably be split to deal with one concept at a time, similar to the “Search for trainers”

operation. However, heterogeneity is not always a problem; “Get today’s dashboard” summarizes heterogeneous data, but the dashboard is a whole.

Solution to exercise 3.4

As seen in section 3.4.5, the success output of the “Read a book” operation is a book; how the book’s information (borrowed status) is interpreted depends on the context.

Chapter 4

Solution to exercise 4.1

- 1 Read a book’s review: `GET /books/12345/reviews/678`. The review ID is a resource identifier (section 4.2.5).
- 2 List a borrower’s books: `GET /borrowers/7890/books`. Fixed path hierarchy to match “borrower’s books” resource (section 4.2.1).
- 3 Search for books available and written in English: `GET /books?available=true&language=en`. `available` and `language` are resource modifiers (section 4.4.5).
- 4 Get book information: `GET /books/12345`. Most other collection resources are plural in this API (and the outside world); you can also switch all collections to singular (section 4.2.5).
- 5 Search for science fiction authors: `GET /authors?genre=science-fiction`. `genre` is a resource modifier, and you’ll probably need other filters like `language` (section 4.4.5). Additionally, the `author` collection name is singular (section 4.2.5).

Solution to exercise 4.2

Refer to section 4.3.6 for mapping typical operations to HTTP methods and section 4.4.7 for typical data locations.

- 1 `GET` because it’s a search operation. Event type and date range are query parameters.
- 2 `POST` because it’s a create operation. The body contains the event (its identifier), number of seats, and attendee information.
- 3 `PUT` or `PATCH` because it’s an update operation. Input data locations: the new number of seats is in the body.
- 4 `DELETE` because it’s a delete operation. The booking (its identifier) is a path parameter.
- 5 `GET` because it’s a search operation. The event type and date range are query parameters. Because the resource is “user’s bookings,” the user (identifier) is probably a path parameter (`GET /users/{userId}/bookings`). But as seen in section 4.4.6, it may also be known by the API (`GET /bookings`). If the operation was “Fetch bookings,” the user (identifier) would be a query parameter.

Solution to exercise 4.3

- 1 200: A search returning no result is a success and usually returns 200 OK (section 4.5.6). 404 Not Found indicates that the resource hasn't been found.
- 2 201 or 202: Creating a resource usually returns 201 Created instead of 200 OK (section 4.5.7). You can also use 202 Accepted if the action hasn't yet been executed but will probably succeed.
- 3 404: Not finding a resource like `/reservations/{reservationId}` is 404 Not Found (section 4.5.8). 500 Internal Server Error indicates an unexpected server crash.
- 4 204, 200, or 202: Returning 204 No Content with no data on what looks like a deletion is valid (section 4.5.4). However, if data is returned, you must return 200 OK instead. You can also use 202 Accepted if the action hasn't yet been executed but will probably succeed.
- 5 400: Any 4xx HTTP status indicates an error in client requests, so 455 can be valid. However, this is an invented HTTP status, which should be avoided (section 4.5.1). At this learning stage, you can use a standard 400 Bad Request instead.

Solution to exercise 4.4

This create operation (`POST /subscriptions`) should return 201 Created (section 4.5.7) and a `Location` header indicating the URL of the created resource (section 4.6.2).

Solution to exercise 4.5

You can use the “do” operation recipes from section 4.8. For this case, I recommend turning the action into a business concept with `POST /translations`. An action resource like `POST /translate` also works. In this case, focusing on the result is probably not a good option; it can result in awkward `POST /translated-texts`.

Chapter 5

Solution to exercise 5.1

As seen in section 5.4.1, a read operation returns the complete resource model, whereas a list operation usually returns a summarized model: a subset of the complete model. Listing A.1 shows that the complete movie model was missing the `stars` property, which is present only in the summarized model. Listing A.2 indicates that the summarized model lacked the `id` property, which is essential for retrieving all movie data. The `music` property was removed because it seems unnecessary to include the composer but not the director's name; this data is considered secondary. An alternative could have been adding the director's name to the summarized data, treating both as primary. As seen in section 5.3.2, another option could be to return the fixed complete model when searching for movies.

Listing A.1 Fixed “Read a movie information” response

```

{
  "id": "ZFqoFq",
  "title": "Ghost In The Shell",
  "releaseYear": "1995",
  "duration": "83",
  "director": "Mamoru Oshii",
  "music": "Kenji Kawai",
  "language": "ja",
  "stars": 5
}

```

← Added missing stars

Listing A.2 Fixed “Search for movies” response

```

[
  {
    "id": "ZFqoFq",
    "title": "Ghost In The Shell",
    "language": "ja",
    "stars": 5,
  }
]

```

← Added missing resource identifier

← Removed secondary data (music)

Solution to exercise 5.2

As seen in section 5.4.2, the `POST /recipes` request (save recipe) is a create operation that should expect the creation model of the recipe resource. This model is a subset of the complete model stripped of server-handled properties. Therefore, the server-generated `id` and `creationDate` properties should be removed.

As seen in section 5.5.1, you must ensure that consumers can provide all requested data. `ingredients` has `id`; although the IDs are human-readable, the consumer must know them. The “Add a new recipe” use case analysis probably was missing a “Select ingredients” step. Adding a “Search for ingredients” operation that returns predefined ingredients and their IDs can fill the gap.

Solution to exercise 5.3

You must keep the `id` for resource identification (section 5.4.1). Section 5.5.3 addresses other properties. The `duration` and `type` are necessary for the application and are justified from a subject matter perspective. Although the UI does not use them, you retain `distance` and `date` because they also make sense from a subject matter perspective; another application, such as the mobile application, may use them. Finally, you can eliminate `lastDbSync`, which is probably internal database information.

Solution to exercise 5.4

As seen in section 5.4.5, similarly named elements may represent different business concepts and resources. The “car” resource (`/cars/{carId}`) is not the same in both

use cases; you need two resources, and thus two operations. Customers choose a car model (`GET /car-models/{carModelId}`; listing A.3), whereas mechanics need a specific car (`GET /cars/{carId}`; listing A.4). The listings simply divide the data, but you can also reconsider the `make` and `model` for the car model resource. When renting, customers may select a family or type of car instead of a specific model. You can replace `make` and `model` with a `types` list, including equivalent options like “Volkswagen Golf,” “Ford Focus,” and “Honda Civic.” However, that answer may not fit a vintage car rental API; customers may choose a specific vehicle and not a generic model. But they will not require the same maintenance data as the mechanic.

Listing A.3 Data returned when getting car model details

```
{
  "id": "m12345",
  "make": "Volkswagen",
  "model": "Golf",
  "rentalPricePerDay": 50,
  "features": ["air conditioning", "GPS", "automatic transmission"],
  "maxPeople": 5,
  "maxLuggage": 3
}
```

Listing A.4 Data returned when getting car details

```
{
  "id": "12345",
  "make": "Volkswagen",
  "model": "Golf",
  "mileage": 80000,
  "yearOfManufacture": 2018,
  "currentCondition": "No issues reported",
  "engineType": "1.4L TSI Turbocharged",
  "fuelType": "Petrol",
  "transmission": "Manual",
  "chassisNumber": "WVWZZZ1JZ9W123456",
  "lastInspectionDate": "2024-01-15",
  "nextInspectionDue": "2024-07-15",
  "tireCondition": "80% tread remaining",
  "brakeCondition": "Good",
  "batteryStatus": "Fully charged"
}
```

Chapter 6

Solution to exercise 6.1

Listing A.5 shows how to fix the path definition. As seen in section 6.2.5, you must be careful with block indentation; the `summary` and `parameters` lists were at the same level as the path and should be indented to be inside. As seen in section 6.4.3, a path parameter definition must be required and have a name matching the path template (`{bookReference}`). You could also change the path to `/books/{bookId}`.

Listing A.5 The fixed resource path definition

```
paths:
  /books/{bookReference}:
    summary: A book
    parameters:
      - name: bookReference
        in: path
        required: true
        schema: {}
```

Fixed indentation

Same name in path and parameter

A path parameter must be required.

Solution to exercise 6.2

The resource path shown in listing A.6 represents a specific course taught by a specific instructor; it requires two path parameters to identify the course and instructor. As seen in section 6.4.3, path parameters are listed in the path-level `parameters` list.

Listing A.6 A path with two path parameters

```
paths:
  /instructors/{instructorId}/courses/{courseId}:
    summary: A specific course of a specific instructor
    parameters:
      - name: instructorId
        in: path
        required: true
        schema: {}
      - name: courseId
        in: path
        required: true
        schema: {}
```

Solution to exercise 6.3

The resource path shown in listing A.7 represents the list of segments of a specific trail; it requires a path parameter to identify the trail. The “retrieve” action is represented by the `get` HTTP method. A query parameter lets you filter on difficulty. As seen in section 6.4.3, you define the path parameter in the path-level `parameters` list. As seen in section 6.6.1, you define the query parameter in the operation-level `parameters` list.

Listing A.7 Path and query parameters

```
paths:
  /trails/{trailId}/segments:
    summary: A trail's segments
    parameters:
      - name: trailId
        in: path
        required: true
        schema: {}
```

```

get:
  summary: List segments of a trail
  parameters:
    - name: difficulty
      in: query
      schema: {}

```

Solution to exercise 6.4

As shown in listing A.8, the `{hotelId}` path parameter wasn't defined (section 6.4.3), the 200 HTTP status wasn't quoted (section 6.7.1), and the `application/json` media type was missing under `content` (section 6.8.1).

Listing A.8 Fixed description

```

paths:
  /hotels/{hotelId}:
    summary: A hotel
    parameters:
      - name: hotelId
        in: path
        required: true
        schema: {}
    get:
      summary: Get hotel details
      responses:
        "200":
          description: Hotel details successfully retrieved
          content:
            application/json:
              schema: {}

```

← Path parameter definition was missing

← HTTP status code wasn't quoted

← Missing media type

Solution to exercise 6.5

As shown in listing A.9, the destinations of a travel package are represented by `/packages/{packageId}/destinations` with a path parameter identifying the package defined at the path level (section 6.4.3). The `post` HTTP operation represents the “add” action. It needs the new destination information in a request body (section 6.6.2). The operation indicates that a destination has been created with a “201” HTTP status (section 6.7.1). The response has a `Location` header (section 6.8.3) and a response body (section 6.8.1).

Listing A.9 A creation operation with a path parameter

```

paths:
  /packages/{packageId}/destinations:
    summary: A travel package's destinations
    parameters:
      - name: packageId
        in: path
        required: true
        schema: {}

```

← Travel package identifier

```

post:
  summary: Add a new destination to a travel package
  requestBody:
    description: Destination info.
    content:
      application/json:
        schema: {}
  responses:
    "201":
      description: Destination added to the travel package
      headers:
        Location:
          description: Destination URL
          schema: {}
      content:
        application/json:
          schema:
            description: Destination info.

```

Info for adding the new destination

The HTTP status indicates creation.

URL of the added destination

The added destination data

Chapter 7

Solution to exercise 7.1

Listing A.10 shows the `Screen`, `Pixels`, and `Pixel` reusable schemas defined under `components.schemas` (section 7.3.1); `Pixels` and `Pixel` are used via a `$ref` (section 7.7.3). Refer to section 7.4 for property, object, and array definitions and marking properties as required.

Listing A.10 Screen schemas

```

...
components:
  schemas:
    Screen:
      type: object
      required:
        - id
        - pixels
      properties:
        id:
          type: string
        pixels:
          $ref: "#/components/schemas/Pixels"
    Pixels:
      description: A matrix of pixels (array of array)
      type: array
      items:
        type: array
        items:
          $ref: "#/components/schemas/Pixel"
    Pixel:
      type: object
      required:
        - rgb

```

Reference to a reusable schema

Array of array

Brightness is optional.

```

- on
properties:
  rgb:
    description: "[r, g, b]"
    type: array
    items:
      type: integer
  brightness:
    type: number
  on:
    type: boolean

```

← Quotes are necessary because of the brackets.

← An integer is “integer,” and a float is “number.”

Solution to exercise 7.2

Listing A.11 shows the operation description, and listing A.12 shows the data models used in its 200 response. Note that the `comment` property is optional, and `mainArtist` could also have been defined via an inline model. Refer to section 7.4 for reusable schema definition (`components.schemas`), section 7.6 for the path and query parameters description, and section 7.7 for the response.

Listing A.11 Operation description

```

paths:
  /artists/{artistId}/albums:
    parameters:
      - name: artistId
        in: path
        required: true
        schema:
          type: string
    get:
      summary: Search an artist's albums
      parameters:
        - name: releaseYear
          in: query
          required: false
          schema:
            type: integer
      responses:
        "200":
          description: Albums found
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/AlbumSummary"

```

← The path parameter is a string.

← The query parameter is an optional integer.

← Content is in JSON

← Reference to array items schema

Listing A.12 Schemas

```

components:
  schemas:
    AlbumSummary:

```

```

required:
  - id
  - name
  - mainArtist
  - releaseYear
properties:
  id:
    type: string
  name:
    type: string
  mainArtist:
    $ref: "#/components/schemas/ArtistSummary"
  releaseYear:
    type: integer
  comment:
    type: string
ArtistSummary:
  required:
    - id
    - name
  properties:
    id:
      type: string
    name:
      type: string

```

← **Comment is optional**

← **The main artist could have been inlined.**

Solution to exercise 7.3

As shown in listing A.13, the provided data was the “job offer complete” resource model you can use to design a create or replace a model by stripping out its read-only properties. Listing A.14 shows how to use these models for the create operation (POST /job-offers), and listing A.15 shows how to use them for the replace operation (PUT /job-offers/{jobOfferReference}). Refer to section 7.4 for the description of schemas under `components.schemas`, section 7.6 for the path parameter, and section 7.7 for the request and response bodies.

Listing A.13 Schemas

```

components:
  schemas:
    JobOffer:
      required:
        - reference
        - created
        - title
        - description
      properties:
        reference:
          type: string
        created:
          type: string
          format: date
        title:

```

← **Job offer resource complete model**

← **“created” is a YYYY-MM-DD date.**


```

        type: string
      description:
        type: string
    JobOfferCreateOrReplace:
      required:
        - title
        - description
      properties:
        title:
          type: string
        description:
          type: string

```

← JobOffer minus read-only properties

Listing A.14 Creating a job offer

```

paths:
  /job-offers:
    post:
      requestBody:
        description: Job offer info.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/JobOfferCreateOrReplace"
      responses:
        "201":
          description: Job offer created
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/JobOffer"

```

References to schemas under components.schemas

Listing A.15 Updating (replacing) a job offer

```

paths:
  ...
  /job-offers/{jobOfferReference}:
    parameters:
      - name: jobOfferReference
        in: path
        required: true
        schema:
          type: string
    put:
      requestBody:
        description: Job offer info.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/JobOfferCreateOrReplace"
      responses:
        "201":
          description: Job offer created

```

Identifies a job offer

Same models as for create

```

content:
  application/json:
    schema:
      $ref: "#/components/schemas/JobOffer"

```

Same models
as for create

Chapter 8

Solution to exercise 8.1

As shown in listing A.16, to make the data user-friendly and interoperable, you can

- Replace airport numbers with interoperable standard IATA airport codes (section 8.4.4; see also www.iata.org/en/publications/directories and <https://github.com/ip2location/ip2location-iata-icao>).
- Add the airport name (section 8.4.2).
- Group the airport identifier and name in an object (section 8.6.1).
- Replace each UNIX timestamp (without the time zone) with an ISO 8601 date and time, including the relevant time zone (section 8.5.3). Providing actual departure and arrival times makes the data more user-friendly by providing processed data (section 8.4.3); without them, consumers must find the arrival and departure time zones to calculate the times.
- Add a processed flight duration (section 8.4.3).
- Represent the added duration with an ISO 8601 duration (section 8.5.3).
- Rename `endTime` to `arrivalTime` to be consistent with `arrivalAirport` and `departureTime` (section 8.9.4).
- Replace 150 and "50.00" (unknown currency, inconsistent types) with an object containing a value and supporting ISO 4217 currency code (section 8.4.2). The generic amount object can be reused for any amount of money.
- Add a processed total price (section 8.4.3).

Listing A.16 Fixed flight data

```

{
  ...
  flight: {
    "number": "AF1234",
    "departureAirport": {
      "id": "CDG",
      "name": "Charles de Gaulle"
    },
    "departureTime": "2024-12-01T14:30:00+01:00",
    "arrivalAirport": {
      "id": "ARN",
      "name": "Stockholm Arlanda"
    },
    "arrivalTime": "2024-12-01T17:10:00+01:00",
    "duration": "PT2H35M"
  },
  "class": "economy",

```

Airport IATA
code number

ISO 8601 date time
with time zone

Consistent name

Processed duration
(ISO 8601)

Human-readable
class code

```

"price": {
  "base": { value: 150, currency: "EUR" },      ← Amount and currency
  "taxes": { value: 50, currency: "EUR" },
  "discount": 0.1,
  "total": { value: 180, currency: "EUR" }      ← Processed total
}

```

Solution to exercise 8.2

As seen in section 8.7.2, `albums` and `tracks` lists should not be embedded in the artist data and should be available as separate resources, because they'll probably require pagination, filtering, and sorting features. The `genres` list doesn't cause any problems because it is essential artist information that is probably a short list of elements.

Solution to exercise 8.3

The following make the book data non-user-friendly:

- Inconsistent casing: `publication_year` versus `bookReference`.
- Inconsistent identifier names: `bookReference` versus `artistsCode`.
- Unclear resource identifier (if you don't know it's a book): `bookReference` versus `artistsCode`.
- Abbreviated names: `authDob` or `ctry`.
- Randomly sorted data: `published_year` is surrounded by author data.
- Unclear purpose: Is `ctry` the author's or the book publication's country? Is genre for the author or the book?

Listing A.17 shows how you can fix the book data to make it user-friendly and consistent.

Listing A.17 Fixed book data

```

{
  "id": "B12345",
  "title": "The Eternal Champion",
  "publicationYear": 1970,
  "publicationCountry": "GBR",
  "genre": "Fantasy",
  "author": {
    "id": "A123",
    "name": "Michael Moorcock",
    "birthDate": "1939-12-18"
  }
}

```

To make the book's data user-friendly and consistent, you can do the following:

- Sort the data (book's data first, then author).
- Group author data in an author object (probably an `AuthorSummary` model).
- Consistently name resource identifiers `id`.

- Use similar prefixes for related data (publication year and country).
- Avoid abbreviations (country).

Note that `authDob` has been renamed `birthDate` instead of `dateOfBirth` to add an `xxxDate` naming pattern (`publicationDate`, `modificationDate`, etc.). Refer to section 8.8 for naming data, section 8.6 for organizing data, and section 8.9.3 for naming identifiers.

Chapter 9

Solution to exercise 9.1

Listing A.18 shows the fixed search-exercises request, and listing A.19 shows its response.

Listing A.18 Fixed search-exercises request

```
GET /users/5678/exercises
    ➤ ?type=walking
    ➤ &fromStartTime=2024-12-20
    ➤ &toEndTime=2024-12-23
```

Meaningful path and appropriate method

Optional search filters consistent with the output

Listing A.19 Fixed search-exercises response

```
200 OK

{
  "metadata": { ... }
  "data": [
    {
      "id": 123,
      "type": "walking",
      "startTime": "2024-12-20T13:05:00Z",
      "endTime": "2024-12-20T13:45:00Z",
      "duration": "PT45M",
      "distance": { "value": 3.5, "unit": "km" }
    }
  ]
}
```

Pagination, sort, and filter metadata

Consistent request and response typing

ISO 8601 duration instead of a custom-formatted string

Object instead of a formatted string

As seen in section 9.3, you must use meaningful paths and appropriate HTTP methods to make your operations easy to understand and guessable. A `POST .../exercise` request is expected to create an exercise rather than searching for exercises; you must use `GET` instead of `POST`. The `/fitness/tracking/summary` segments have unclear roles and don't help define the resources you interact with; you can remove them.

As seen in section 9.4, you must choose appropriate input data locations and minimize the required elements. The `userId` query parameter is mandatory, although it can be optional for search filters. However, you won't filter across all users because

you are looking for a user's exercises. If SMEs approve, you can change it to a path parameter (`/users/{userId}/exercises`). All properties in the body should be query parameters because they are search filters. Dates are mandatory, but you can make them optional, defaulting to the last 10 days or the current week's exercises.

There are local (section 9.4) and global (section 9.10) inconsistency problems. The names of the `fromStartDate` and `endDate` query parameters are inconsistent; `fromStartDate` and `toEndDate` would be clearer. However, their names and types don't align with `startTime` and `endTime` in the response. The request uses a `Date` suffix and UNIX timestamp, and the response uses `Time` and ISO 8601. To make everything consistent, you should base the request on the response.

As seen in section 9.1, your operations must use user-friendly, interoperable data. The response contains strings with custom formats: `duration` is "45 minutes", and `distance` is "3 kilometers". You can use an ISO 8601 duration and an object with a value and unit for the distance.

Finally, as seen in section 9.6, a search operation should handle pagination and possibly sorting (not shown in the example request) in addition to search filters. You need to add metadata for these features in the response.

Solution to exercise 9.2

It's essential to use the same features consistently (section 9.10). These operations provide two ways to get XML, JSON, or CSV data: using a `format` query parameter or indicating a `.{format}` extension at the end of the resource path. You must choose a single way to do this.

The query parameter is preferable to the `.{format}` resource path extension (path parameter) because it serves as a resource modifier rather than an identifier and also prevents unnecessary operations for each format. You can enhance the query parameter by making it optional and defaulting to JSON (section 9.4.4). However, I suggest using a standard feature: HTTP content negotiation (section 9.7.1). An `Accept` request header can specify the expected format (`application/json`, `application/xml`, or `text/csv`). JSON will be returned by default if the header is absent in a call like `GET /games/{name}/classes` or `GET /games/{name}/spells`.

Solution to exercise 9.3

As seen in section 9.8, this API hides one capability inside another. The operation used to reschedule an event is "Update an event." Although modifying the date of an event implicitly reschedules it, a separate operation would be better because rescheduling an event is probably an important action when managing events: it implies proposing and validating the event, for example. Additionally, the data can help you detect something wrong in this case. Although the `date` property makes sense for an event, `reason` is tied to rescheduling more than to the event itself, pointing to the need for a dedicated operation.

Solution to exercise 9.4

As seen in section 9.8.2, you must use appropriate HTTP status codes and avoid custom ones. Although the 432 HTTP status code clearly indicates an error caused by the consumer (4xx class), it's not standard. The API designer picked an unassigned HTTP status in the IANA registry and decided it meant `Missing` and `Invalid Data`. The operation should return a generic 400 `Bad Request` or a more specific 422 `Unprocessable Content` (indicating an acceptable request body content type but an incorrect body value).

As seen in section 9.8.3, an error must provide informative and problem-solving feedback. Simply indicating that there's an error related to the request doesn't help fix it. Listing A.20 illustrates how you can use the Problem Details for HTTP API format and enhance it with a list of errors describing each problem, as seen in section 9.8.6.

Listing A.20 Fixed error response

```
400 Bad Request
Content-Type: application/problem+json

{
  "status": 400,
  "type": "https://api.iam.net/validation-error",
  "title": "Invalid request",
  "description": "Missing lastName and role",
  "errors": [
    {
      "type": "https://api.iam.net/validation-error/required",
      "title": "Missing required property",
      "description": "lastName is missing",
      "source": {
        "location": "body",
        "name": "lastName",
        "pointer": "#/lastName"
      }
    },
    {
      "type": "https://api.iam.net/validation-error/invalid-value",
      "title": "Invalid value",
      "description": "role must be admin or user",
      "source": {
        "location": "body",
        "name": "role",
        "pointer": "#/permissions/O/role"
      },
      "values": ["admin", "user"]
    }
  ]
}
```

Standard HTTP status

Exhaustive problem-solving feedback

Clear error location

Solution to exercise 9.5

The first problem is the inconsistency between the create/replace request and the complete response models (section 9.4.2). The `systolic` and `diastolic` properties are numbers in the request but objects in the response. Suppose a consumer needs to correct the diastolic value. In that case, they must re-create the proper input model with the inconsistent structure instead of just modifying the needed data in the complete model response and sending it back using `PUT /blood-pressure/BP7890` (the server ignores additional properties that aren't in the create/replace model; section 9.8.1). You have two options to optimize this part of the design: use an object in both the request and response but make the unit optional in the request (`"systolic": { "value": 120 }`), or use a number in both, assuming the blood pressure unit is a standard that doesn't need to be indicated.

The second problem involves the subject matter. Allowing the server to determine when measurements are taken can cause problems. For instance, if data isn't sent in real time, the measurement time will be inaccurate (section 9.4.4). The `measurement-time` should probably be required in the request.

Chapter 10

Solution to exercise 10.1

There are two levels of problems: the operations themselves and the flow as a whole. Each operation of a flow must be user-friendly (section 10.1.1). However, listing doctors and getting slots don't propose filtering, resulting in consumer-side filtering on doctor specialty and slot availability. The last operation hides "Scheduling an appointment" inside a slot's update.

The flow lacks flexibility (section 10.1.3). Users must select a doctor before finding an appointment date and time. Some users may prefer to find the earliest appointment regardless of the doctor. Additionally, if consumers know the doctor's ID, they cannot directly schedule an appointment at a given date and time because an opaque slot ID is needed.

You can fix the flow as follows:

- Find appointment availabilities (`GET /appointment-availabilities`) with the doctor name, specialty, and date and time range as optional filters.
- Schedule an appointment (`POST /appointments`), which expects a doctor ID and date and time.

As in section 10.3, you can aggregate operations to create a more flexible, use-case-focused operation. The new "Find appointment availabilities" operation combines "List doctors" and "Get doctor's slots." The optional filters can be used separately or together, enabling all search options.

The new "Schedule an appointment" operation replaces "Update slot" with patient details; it clearly shows its purpose, making it user-friendly (section 10.1.1). Replacing

the slot ID with an interoperable date and time enables the flow to be entered directly in the last step with a doctor ID (section 10.1.3).

Solution to exercise 10.2

This highly inflexible flow forces consumers to collect data in a specific order (section 10.4). It also prevents modifications with PUT requests that are usable only once. Additionally, it requests unnecessary information (the number of rooms) and returns an estimate only after all data is provided. From a resource design perspective, you can also question the fine-grained resources representing quotes and room properties (room-count and condition, for example).

The following flow collects vital information first and lets you flexibly save partial or complete data, modify it, and get an estimate at any time during and after providing data:

- POST /quotes that accepts an optional description and a list of rooms with more or less data. It returns the provided data plus an estimation.
- PATCH /quotes/{quoteId} to update the description and list of rooms with their details.
- GET /quotes/{quoteId} to return the same data as the creation.

If a project can have many rooms requiring filtering, sorting, and paginating, you may want to handle rooms independently with POST /quotes/{quoteId}/rooms and PATCH or DELETE /quotes/{quoteId}/rooms/{roomIdOrIndex}.

Chapter 11

Solution to exercise 11.1

As seen in section 11.2.2, you can use your subject matter knowledge to identify independent operation sets by looking at flows. The following list shows how you can organize the operations into two or four groups:

- Employee administration (in the future, you can split it to have time-off operations in a dedicated API)
 - Add employee time off (POST /employees/{employeeId}/time-offs)
 - Update employee time off request (PUT /time-offs/{timeOffId})
 - List employee time offs (POST /employees/{employeeId}/time-offs)
 - List employees (GET /employees)
- Student and course administration (that you can subdivide)
 - Course
 - Search courses (GET /courses)
 - Create a course (POST /courses)
 - Read a course (GET /courses/{courseId})
 - Update a course (PUT /courses/{courseId})
 - Cancel a course (DELETE /courses/{courseId})

- Student
 - Add student (POST /students)
 - Update student (PUT /students/{studentId})
 - Delete students (DELETE /students/{studentId})
 - Search students (GET /students)
- Enrollment and record
 - Enroll a student in a course (POST /courses/{courseId}/students)
 - Handle student withdrawal from a course (DELETE /courses/{courseId}/students/{studentId})
 - Track course enrollment (GET /courses/{courseId}/students)
 - Define course exam date (POST /courses/{courseId}/exams)
 - List all exams (GET /exams)
 - Record or update student grades for an exam (PUT /exams/{examId}/grades/{studentId})
 - List grades for students, courses, or exams (GET /grades)

Acting as a junior SME, you can guess that there are at least two groups you can use to separate employee-related operations from the others. You can confirm this by building flows around these operations and demonstrating that they are not connected to other operations. For now, this group mainly focuses on time off, but separating time-off operations from “List employees” doesn’t make sense (yet). If more employee-related operations are added, you may want to split this group in the future.

You can create a “Student and course administration” group for the rest of the operations. You can also consider dividing it into smaller APIs. The course and student groups are almost independent; course and student IDs are needed to create an exam or enroll a student, probably inducing calls to “Search courses and students.” However, as seen in section 11.2.3, having that pattern is OK as long as all blocks share the same interoperable IDs.

Solution to exercise 11.2

As seen in section 11.3.2, a name expresses what the API covers. Zeus API clearly means “the API for the Zeus application.” However, not everyone may know what the job of this application is due to its cryptic code name. Logistics API is clearer because it is subject-matter-oriented. However, logistics usually covers more than just a shipment. It may be an option if the Zeus application covers all logistics functions. So, you can challenge the granularity of this API (section 11.2). ReGenesis Shipment API includes the ReGenesis project name, which doesn’t bring any value in understanding the API name and will quickly become outdated. But it also includes “Shipment,” which best captures the API’s global intent. A better option is “Shipment API” or “Shipment” (leading to a /shipment base path).

Chapter 12

Solution to exercise 12.1

You first need to determine which data is sensitive in the context of sharing it (section 12.3.1). When reading a user, the phone number, name, email, and birthdate can be considered sensitive when the user profile is made available to all other users of the platform. For the workout, the GPS coordinates are sensitive.

As seen in section 12.3.2, you can remove the sensitive data, which works well for the PII data (phone number, email, and birth date). However, removing the GPS coordinates also eliminates the possibility for consumers to calculate valuable information, such as the path's distance and level of difficulty, which could be interesting for comparing different users' performance. You might consider replacing these GPS coordinates with the indicators calculated from them. Actually, as you learned in section 8.4.3, you may even want to make this ready-to-use data part of the original workout data.

The two operations deal with sensitive data. As seen in section 12.3.4, you can question using the same operations in the owner and sharing contexts and consider creating dedicated operations. You can also let the implementation handle removing the sensitive data when the operation is called for a user other than the owner.

Solution to exercise 12.2

The `id` parameter can hold any possible identifier available in any country, including highly sensitive identifiers such as Social Security numbers. However, as seen in section 12.6.1, query parameters can be logged and are visible in many places. A simple fix is to move the `id` parameter in the request body; the operation can be redesigned in `/POST /identifiers/search` (section 12.6.2). You could also consider encrypting the value to use a query parameter, but that would make the API more complex to use (section 12.6.4).

Solution to exercise 12.3

To handle this securely, you need a scope (section 12.7), and you must ensure that the implementation performs the appropriate check (section 12.4.1). With a scope such as `patient_conditions` attached to the operation, you can ensure that only specific applications, such as Patient Folder and Treatment, can call this operation, and Schedule can't. However, that's not enough, because different doctors may access these applications. In the operation description in the OpenAPI document, you must clarify that only doctors treating the patient may access this data; it will be up to the implementation developers to code this check.

Solution to exercise 12.4

In this API, each operation is attached to a single resource-based scope (books scope for `/books` and book for `/books/{bookId}`); the scope grants access to all of a resource's

operations. As seen in section 12.8.2, such a strategy may open security gaps. For instance, an application that only needs to know which books are available and each book's details will need the books and book scopes, gaining access to the operations to add or modify books that the application doesn't need. It would be better to propose finer-grained operation-level scopes (section 12.8.1). However, this could lead to many scopes. Another approach would be to create scopes for use cases (section 12.8.2) or based on end-user or consumer profiles (section 12.8.5).

Solution to exercise 12.5

As seen in section 12.10.2, if B doesn't have access to this event and you think they may request permissions to access it, the operation can return `403 Forbidden`. If not, it should return `404 Not Found`. If B already has access to this event, the operation returns `200 OK`.

Chapter 13

Solution to exercise 13.1

Because the "Search courses" operation doesn't propose any filters and only returns the course names, the consuming application will list all courses and then read each one to perform filtering. As seen in section 13.1.1, this may result in a long response time perceptible by end users, a high volume of output data, and a high load on the infrastructure.

The consuming application can cache data (section 13.4); however, without directives, the search results may be incorrect, may be missing new courses, and may include outdated, removed, or updated course data (section 13.4.2). Additionally, it's not the consumers' job to filter data.

To be more efficient, you need to enhance the list operations with pagination, sort, and filtering possibilities (section 13.6) and relevant data (section 13.3.3), which probably means all data returned by the "Read course" operations minus the detailed lessons. If you want to gain a few bytes, you can return only the domain code and add operations to search and read domains (section 13.5.4). But if you go that way, you can also reevaluate the flow (section 13.3.4): if users don't search cross-domain, a domain-focused search (`GET /domains/{domainId}/courses`) may better meet user needs.

Solution to exercise 13.2

Before discussing HTTP headers, you must determine the caching policy (section 13.4.3). As new books are added at a fixed date, data can be cached up until the 12th of each month at 00:00. However, based on the search criteria, you can deduce that the data returned for a book includes at least availability, title, author, genre, and description. Although title, author, genre, and description won't be modified, availability can be updated at any time. Therefore, consumers may cache data but must validate it before use.

Therefore, the response to `GET /books` should contain `Cache-Control: no-cache` and `ETag: "12345abc"` to indicate that consumers can cache data but must validate it before use (section 13.4.4). To validate that the data is up to date, the consumer sends a `GET /books` request with an `If-None-Match: "12345abc"` header (section 13.4.5). If the available status is not returned, you can set `Cache-Control: max-age:46800` to indicate that the data is valid for the next 13 hours (given that the request was made on the 11th at 11:00 a.m.).

Solution to exercise 13.3

This operation proposes pagination (cursor-based) and filtering, which are good for efficiency because they let you return only the needed data. However, the default page size (`limit`) is 10,000, and its maximum is 1,000,000, which may cause performance problems. As seen in section 13.6.1, you must optimize page size so consumers and end users get significant data as fast as possible without affecting the infrastructure.

Consumers will systematically receive 10,000 elements unless developers think of overriding that value with a smaller one. These are probably far more elements than end users need by default, and they will have to wait to download all this data. You can reduce it based on the number of threads a typical UI will show, as well as user habits. Returning more elements than are visible, let's say 50, will enable consumers to load the next threads in the background (with `GET /threads?next={cursor}`), offering end users a smooth experience.

However, consumers can override the limit to 1,000,000, which can heavily stress the infrastructure and probably doesn't make sense for most users unless the operation is used in a use case related to archiving user data or performing research. You can make the timeline operation focus on the typical user and set its maximum to the default value, only letting users retrieve fewer threads. Research and archiving can be handled via other operations that are accessible only to selected applications and users. You can also condition the maximum value on user permissions or scopes (section 12.8.5).

Chapter 14

Solution to exercise 14.1

Before considering the 503 `Service Unavailable` status (section 14.2.3), you can challenge this constraint (section 14.1.3). You can ask whether human validation is actually needed. The implementation can probably validate a request based on the dimensions and type of wood, given the available stock and workload.

Solution to exercise 14.2

Enabling rate limiting (13.2.2) may protect the Very Fast Shipping infrastructure but probably won't meet the company's user needs. If consumers repeatedly call operations, it's probably so they can get their shipment status in real time. Instead of having

consumers needlessly call operations, the API provider can define a webhook to notify them about status modifications for shipments they follow (section 14.6).

Solution to exercise 14.3

Based on this chapter's content, this design has the following problems:

- The same file may be re-uploaded several times to extract different information. For example, once instruments have been identified, a consumer may decide to extract an audio file for a specific instrument.
- The processing time can be long and incompatible with the usual synchronous request-response mechanism; a file with 10 minutes of audio can take up to 10 minutes for a single complex task.
- If the consumer has a problem when waiting for and receiving the response, they must re-request the file processing, including the file upload.

It's unrelated to this chapter, but the unique operation tries to tackle different capabilities, hiding what the API is capable of (section 9.9).

You can improve the design by using generic files (section 14.3.1), long operations (section 14.7.1), and callbacks (section 14.7.2). An improved design could lead to the following flow:

- The consumer uploads generic files with `POST /files`, passing raw binary data in the request body (getting rid of the Base64 overhead in the process).
- The consumer starts a long processing operation with `POST /processing-tasks` and passes a `fileId`.
- The provider notifies the consumer of the end of processing with a callback.
- The consumer lists the processing results with `GET /processing-tasks/{processingId}/results`; depending on the task, a result may contain the `fileId` of a generated file, such as the ID of a backing track audio file.
- The consumer retrieves generated files with `GET /files/{fileId}`.

You could also consider replacing the unique `POST /processing-tasks` with more focused operations, such as `POST /backtracks`.

Chapter 15

Solution to exercise 15.1

The `info.version` contains information specific to the implementation. You can guess that the application exposing the API is called Athena. However, it's something that consumers don't need to know, and it doesn't specify the API's version. Additionally, the `GET /v1/courses` and `POST /v2/courses` paths are confusing. You may wonder what is versioned: the API, resources, or operations.

As seen in section 15.4 and illustrated in listing A.21, you should follow the most commonly used versioning scheme: version the entire API with a semantic version

number, and add the major version number to the base URL. Note that the new version is 1.1 because the previous version was the initial one (1.0). Adding an operation only requires bumping the minor version. As seen in section 15.7, you use the `servers` list to indicate the API base URL instead of defining it on each resource. The API is for third parties, so you may want to use a 1.1.0 version number to indicate the OpenAPI document changes.

Listing A.21 Fixed OpenAPI document

```
openapi: "3.1.0"

info:
  title: Online Course Platform
  version: "1.1"

servers:
  - url: /v1

paths:
  /courses:
    get:
      summary: Search courses
    post:
      summary: Add a course
```

Fixed version number

Base URL for all resources

New operation added to the course resources

Solution to exercise 15.2

Listing A.22 highlights the modifications (section 15.2.2):

- The required `difficulty` property becomes optional (backward-compatible). Unmodified consumers will continue to send it systematically.
- The integer `duration` property is replaced with ISO 8601 duration strings (non-backward-compatible). The server will return an error because unmodified consumers will send an integer.
- The `insane` value is added to the `difficulty` enumeration (backward-compatible). Unmodified consumers won't send it.
- The `activities` array now has `maxItems` defined (non-backward-compatible). Unmodified consumers may send more items than expected because no limit was fixed before; if they do, they will get an error.
- The `dur` property (under `activities`) is renamed `duration` (non-backward-compatible). The server will receive requests with a `dur` property instead of the required `duration`, causing an error.
- The `dur` property also has its type changed to an ISO 8601 duration string (non-backward-compatible). Consumers won't see related errors because they'll send a `dur` property instead of `duration`.

Additionally, the schema's writing has changed. Although moving `required` to the top has no consequences, reordering properties may affect consumers that rely on this

order (section 15.2.7). However, they do so at their own risk, because the order of properties in an object isn't supposed to matter and may change at any moment.

Listing A.22 Request body data after modifications

```

required:
  - duration
  - activities
properties:
  duration:
    type: string
    description: ISO8601 duration.
  difficulty:
    type: string
    enum:
      - easy
      - challenging
      - insane
  activities:
    type: array
    minItems: 1
    maxItems: 20
    items:
      required:
        - name
        - repetitions
      properties:
        name:
          type: string
          description: Plank, burpees, etc.
        duration:
          type: string
          description: ISO8601 duration.
        repetitions:
          type: integer
          description: Number of repetitions

```

Moving "required" has no effect.

Making "difficulty" optional is OK.

Reordering properties has no effect (usually).

Changing "type" is not OK.

Adding "value" to "enum" is OK.

Adding a max array size is not OK.

Renaming is not OK.

Changing "type" is not OK.

Solution to exercise 15.3

Listing A.23 highlights the modifications (section 15.2.1):

- The optional date property becomes required (backward-compatible). Consumers are used to sometimes getting it; they'll systematically receive it now.
- The ID format (regex) has been modified. This may imply that the ID has been replaced, making the change probably non-backward-compatible if consumers know or use the ID on their side. However, it may be backward-compatible if consumers only retrieve the ID from previous calls and never store it.
- The timestamp (integer) date becomes an ISO 8601 string (non-backward-compatible). This will cause parsing errors on the consumer side.

- `conditionComment` and `conditionStatus` have been moved to a `condition` object (non-backward-compatible). Consumers won't find these two properties.
- The `machineIdentifier` property has been renamed `machineId` (non-backward-compatible). Consumers won't find it.
- A maintenance value has been added to the enum of `status`. This is non-backward-compatible if the value is interpreted but backward-compatible if it's only shown to end users.
- The description of `temperature` indicates that the units have changed from Celsius to Fahrenheit (non-backward-compatible). Consumers will interpret the received temperature as Celsius.
- The `pressure` property has been added (backward-compatible). Consumers will not use it.

Listing A.23 Response body data after modifications

```

required:
- id
- date
- machineId
- condition
properties:
  id:
    type: string
    pattern: "[a-z]{3}-[0-9]{10}"
  date:
    type: string
    format: date-time
  condition:
    required:
      - conditionComment
      - conditionStatus
    properties:
      conditionComment:
        type: string
      conditionStatus:
        type: string
      enum:
        - green
        - orange
        - red
  machineId:
    type: integer
  status:
    type: string
    enum:
      - running
      - stopped
      - maintenance
  temperature:
    type: number
    description: Fahrenheit.

```

← Making "date" required is OK.

← Changing the ID format is probably not OK.

← Changing "type" is not OK.

← Moving properties in an object is not OK.

← Renaming a property is not OK.

← Adding value to "enum" is probably not OK.

← Changing units is not OK.


```
pressure:
  type: number
  description: Kilopascals.
```



Adding data
is OK.

Solution to exercise 15.4

As seen in section 15.2.2, although adding an optional query parameter is backward-compatible, returning only the first 50 events instead of all events isn't. Non-updated consumers will miss all other events, showing incomplete data to end users. To be backward compatible, you must ensure that unmodified consumers get all events with `GET /events`. To enable pagination only for modified consumers, you can add the optional page query parameter as initially intended but virtually set its default value to "all events," staying within the initial "limits." That way, the server will return paginated results only on `GET /events?page={index}` requests. You can later change the server's behavior when no page parameter is provided, once you're sure all consumers have been updated.

Solution to exercise 15.5

As seen in section 15.6.3, a design should favor extensibility. Here, the operation will return, for example, a plain `12345`. If you later want to add other information, it will cause a breaking change because you'll need to replace the plain integer with an object. To fix this design, you can replace the integer with an object that has an `id` property. Additionally, to be HTTP-compliant, you should consider adding the missing `Location` header, which should accompany a `201 Created`.

Chapter 16

Solution to exercise 16.1

It's risky to proceed this way. As seen in section 16.1.3, the question is how much you can trust that other API. You may be unlucky and pick the only one that uses singular resource nouns while all others use plural (or the opposite). You should look for guidelines, hoping they exist, or ask a tech lead or architect. As a last resort, you can look at several APIs to get a wider view of the company's API landscape (and suggest creating guidelines).

Solution to exercise 16.2

As seen in section 16.5.3, guidelines should include proven recipes. It makes sense if a practice is already installed and validated, and adding recipes to the guidelines only implies describing what exists. But if no one has needed to upload and download files so far, it's too early for such a complex topic that can be affected by the context. Wait until you need that feature to define a solution that works and add it to the guidelines.

Solution to exercise 16.3

As seen in section 16.1.2, the scope of choosing a resource name seems more local to the API you're designing and not applicable to all other APIs. Therefore, there's no need to add such a rule to the company's guidelines, which deal with cross-API concerns (section 16.3). However, if the guidelines don't mention that resource names are plural, they must be updated, because this decision affects all APIs. This decision, using plural nouns for resources, must be accepted by the API design guidelines' stakeholders.

Solution to exercise 16.4

As seen in section 16.4.2, if your guidelines contain hundreds of rules without further explanation, API designers may not apply all the rules correctly, and you may create contradicting rules. It's essential to have ready-to-use recipes.

Solution to exercise 16.5

Each rule added to the guidelines must bring value and have a reason to exist (section 16.5.4). Therefore, a first quick response can be to ask why you should always use UUIDs. However, the governance team may say, "It's to ensure that all IDs are globally unique and don't contain sensitive information," which is a valid argument for the UUID option. But always using UUIDs has drawbacks. You can use an ADR document (section 16.2.3) to compare the pros and cons of each option and demonstrate why you should decide about resource IDs on a per-resource basis. Typically, only using UUIDs kills interoperability. To fill this ADR, you should support your reasoning (section 16.1.5), in particular by looking for interoperable IDs that are already used across your systems or in the outside world.

Chapter 17**Solution to exercise 17.1**

As seen in section 17.5, you can define reusable responses for 401 and 500 responses, shared by all operations, under `components.responses` and then use them with the appropriate `$ref`. The 200 responses are specific to each operation and won't benefit from being defined as reusable responses. As seen in section 17.6, you can also consider adding the `UnauthorizedError` and `UnexpectedError` responses to a library file and use them with a reference such as `$ref: "pathOrUrl/library.openapi.yaml#/components/responses/UnauthorizedError"`.

Listing A.24 Optimized OpenAPI document

```
openapi: 3.1.0
info: ...
paths:
  /authors:
    get:
```

```

responses:
  "200":
    description: Found authors
    content: ...
  "401":
    $ref: "#/components/responses/UnauthorizedError"
  "500":
    $ref: "#/components/responses/UnexpectedError"
/books:
  get:
    responses:
      "200":
        description: Found books
        content: ...
      "401":
        $ref: "#/components/responses/UnauthorizedError"
      "500":
        $ref: "#/components/responses/UnexpectedError"
components:
  responses:
    UnauthorizedError:
      description: Invalid token
      content: ...
    UnexpectedError:
      description: Unexpected error
      content: ...

```

Reusable response definitions

Reference to reusable responses

Solution to exercise 17.2

As seen in section 17.3.1, you can use the path-level parameters to prevent duplicating the `authorId` path parameter definition between operations under `/author/{authorId}`. To prevent the duplication of the schema of an author's ID, you can define a reusable `Author` schema and target its `id` property with a deep reference (section 17.2.3). Finally, although doing so isn't necessary here, you can define a reusable `AuthorId` parameter if other paths need it (section 17.3.2).

Listing A.25 Optimized OpenAPI document

```

openapi: 3.1.0
info: ...
paths:
  /authors/{authorId}:
    parameters:
      - $ref: "#/components/parameters/AuthorId"
    get:
      responses:
        "200":
          description: An author
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Author"
    delete:

```

Path-level parameters

Reusable parameter

```

    responses:
      "204":
        description: Author deleted.

components:
  parameters:
    AuthorId:
      name: authorId
      in: path
      required: true
      schema:
        $ref: "#/components/schemas/Author/properties/id"
  schemas:
    Author:
      properties:
        id:
          type: string

```

← **Reusable parameter**

← **Deep reference**

Solution to exercise 17.3

As seen in section 17.2.5, you can create a unique read-and-write `Author` schema with a read-only `id` property and use this schema in the bodies of both the request and the response.

Listing A.26 Optimized OpenAPI document

```

openapi: 3.1.0
info: ...
paths:
  /authors:
    post:
      requestBody:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Author"
      responses:
        "201":
          description: Author created.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Author"

components:
  schemas:
    Author:
      properties:
        id:
          type: string
          readOnly: true
        name:
          type: string

```

← **The readOnly flags “hide” id in the request body.**

Solution to exercise 17.4

The `AuthorSummary` model is a subset of the `Author` model. As seen in section 17.2.6, you can reuse the summary model to define the complete model by using the `allOf` keyword. The `id` property initially didn't have a `readOnly` flag in the `AuthorSummary` model; it was in the `Author` model. The flag does not affect the summary model used in the response of a search operation (`GET /authors`). The `id` being read-only affects the request bodies, such as the `POST /authors` request body, which uses the complete model.

Listing A.27 Optimized OpenAPI document

```
openapi: 3.1.0
...
components:
  schemas:
    AuthorSummary:
      properties:
        id:
          type: string
          readOnly: true
        name:
          type: string
    Author:
      allOf:
        - $ref: "#/components/schemas/AuthorSummary"
        - properties:
            genres:
              type: array
              items:
                type: string
```

← Applies to both schemas

← Aggregates schemas

Chapter 18

Solution to exercise 18.1

The API version is located in `info.version` in an OpenAPI document. This gives the `$.info.version` JSON given path for the rule in listing A.28 (section 18.5). You use the `pattern` function to check whether the value resembles 1.0.0 with a regex (section 18.6). You add a message indicating the expected format to help users fix the problem (section 18.7).

Listing A.28 Spectral rule

```
rules:
  semantic-versioning:
    description: Semantic versioning must be used
    message: "{{value}} is not a valid semantic version (1.0.0, for example)"
    severity: error
    given:
      - $.info.version
```

← API version

```

then:
  - function: pattern          ← Regex validation
    functionOptions:
      match: "^ [0-9]+\\. [0-9]+\\. [0-9]+ $"

```

Solution to exercise 18.2

Listing A.29 shows a rule ensuring that no query parameter is defined on POST `/any/thing/search` operations (`@property.match(regex)`; section 18.5.4). The tricky part is that this rule must target both path- and operation-level parameters, keeping only the query parameter (`in`). It also ensures that no element is found with the undefined function (section 18.6.3). You may want to create a separate rule limiting the use of path-level parameters, either excluding query parameters or only allowing path parameters (excluding header and query parameters); you would then only need to target operation-level parameters in this rule.

Listing A.29 Spectral rule

```

rules:
  post-search-no-query:
    description: POST /search must not have query parameters
    severity: error
    given:
      - $.paths[?(@property.match(/\/search$/))]
        ➡ .parameters[?(@.in === "query")]
      - $.paths[?(@property.match(/\/search$/))].post
        ➡ .parameters[?(@.in === "query")]
    then:
      - function: undefined

```

Solution to exercise 18.3

You must ensure that each operation has a 503 response defined and that the response has the expected schema. To do this efficiently, use a shared OpenAPI library defining the expected 503 response, because this applies to all your APIs (section 18.6.4), leading to the rule in listing A.30. You target all `responses` objects and then work at the 503 field level; having the field defined implies that a problem will be detected if no 503 response is found. There, instead of checking all of the response's elements, including the schema, you verify the presence of the appropriate reference to your (trusted) OpenAPI library. However, to make this work, you must set the rule's `resolved` flag to `false`, so the rule works on the raw document that includes the original `$ref` values.

Listing A.30 Spectral rule

```

rules:
  service-unavailable-ref:
    description: An operation must return a standard 503 response
    message: Use 503 response from the OpenAPI library
    resolved: false

```

```

severity: error
given:
  - $.paths.*.responses
  then:
    - field: "503"
      function: schema
      functionOptions:
        schema:
          const:
            $ref: "library.openapi.yaml
            ➡ #/components/responses/ServiceUnavailable"

```

← All responses

← Checking the reference to the library

Solution to exercise 18.4

Listing A.31 shows a rule targeting all 201 response schemas and ensuring that a required string or integer id property is defined while accepting other properties with the schema function. As seen in section 18.6.5, you can't use the `const` JSON Schema keyword as you do, for example, when checking whether a reference is used for a response. You define a schema of the expected schema. It expects to find properties named `required` and `properties`. You ensure that the `required` list of the found schema contains an `id` value (allowing for other values). And you ensure that this schema defines an `id` property with a `type` set to `string` or `integer`.

Listing A.31 Spectral rule

```

rules:
  required-resource-id-201:
    description: |
      Response 201 must have a required id
      that is a string or integer
    severity: error
    given:
      - $.paths.*.responses.201.content.application/json.schema
    then:
      - function: schema
        functionOptions:
          schema:
            required:
              - required
              - properties
            properties:
              required:
                contains:
                  const: id
            properties:
              required:
                - id
            properties:
              id:
                required:
                  - type
                properties:
                  type:

```

← The schema has "required" and "properties".

← Requires "id"

← Defines "id"

← "id" is "string" or "integer".

```
enum:
  - string
  - integer
```

Solution to exercise 18.5

As seen in section 18.5.1 and illustrated in listing A.32, you can add the undefined function to a rule's then to see what the given paths find.

Listing A.32 Spectral rule

```
rules:
  my-new-rule:
    given:
      - $.paths[?(@property.match(/}$/))].*.
        <lineararrow/>responses[?(@property.match(/^(4|5)/))]
    then:
      - function: undefined          ← Shows what "given" finds
```

Chapter 19

Solution to exercise 19.1

As seen in section 19.2.2, you can add tags to each operation to group them in an OpenAPI UI (only the first two operations are shown; proceed similarly for the others). You add a root tags list to ensure that the groups are ordered appropriately. You can add a description if necessary.

Listing A.33 Groups

```
tags:
  - name: Book
  - name: Author          ← Orders the groups

paths:
  /authors:
    get:
      tags:
        - Author          ← Adds an operation to the Author group
      responses:
        "200":
          description: Authors
  /authors/{authorId}/books:
    parameters:
      - $ref: "#/components/parameters/AuthorId"
    get:
      tags:
        - Book            ← Adds an operation to the Book group
      responses:
        "200":
          description: Books
  ...
```


Solution to exercise 19.2

Listing A.34 shows the updated schemas based on section 19.3. As seen in section 19.5.1, you add implementation notes with a level-five Markdown section in the description of the direction property.

Listing A.34 Detailed schemas

```
...
components:
  schemas:
    Speed:
      properties:
        id:
          type: string
          pattern: "sp-[a-z]{3}-[0-9]+"      ← Regex
          example: sp-abc-12345
        date:
          type: string
          format: date-time
        value:
          type: number
          exclusiveMinimum: 0
          default: 0                        ← Default value
          ← Value can't be negative
        unit:
          type: string
          enum:
            - kph
            - mph
          default: kph
          ← kph or mph
        direction:
          description: |
            ##### Implementation
            SPD23 column of the ZSPD table
          ← For implementers' eyes only
          type: integer
          minimum: 0
          exclusiveMaximum: 360
          ← Direction >= 0 and < 360
      Speeds:
        type: array
        minItems: 1
        maxItems: 100
        items:
          $ref: "#/components/schemas/Speed"
          ← Array size
```


Numerics

100 Continue status 344
1XX class 85
200 OK status 71
201 Created status 424
204-no-body Spectral rule 456
204 No Content status 144
207 Multi-Status status 323
2XX class 85, 249
303 See Other status 345
308 Permanent Redirect status 370
3XX class 85, 250
400 Bad Request status 224
404 Not Found status 84, 144, 413
405 Method Not Allowed status 371
412 Precondition Failed status 284
413 Content Too Large status 344
415 Unsupported Media Type status 224
422 Unprocessable Content status 224
429 Too Many Requests status 372
4XX class 85, 344
503 Service Unavailable status 336, 461
5XX class 85

A

Accept 220–221
 application/json 221, 224, 340
 application/json HTTP header 222, 340
 application/pdf 224
 application/pdf HTTP header 224
 application/xml 221
 text/csv 224
 text/csv HTTP header 224
Accept header 221, 315, 340, 377
Accept HTTP header 220–221
Accept-Language 222
Accept-Language header 222
Accept-Language HTTP header 222
adapting API design to context, delegating file downloads and uploads 345–346
adapting to context, provider-sourced events with web-hooks
 choosing event data granularity 349–350
 dealing with failures 351–352
 defining expected behavior 351
 describing with OpenAPI 353
 designing operations 348
 designing securely 350–351
 overview of 347
 should be optional 348
 using standard event format 348–349
Add operation 247
ADR (architectural decision record) 401–402, 439
allOf keyword 479
Allow 260
Allow HTTP header 260
Allow response header 260
alternate relation 261
alternative paths 35–38
 adding alternative branches on each use case 37–38
 analyzing alternative users and use cases 38
 analyzing for each step 36–37
amount property 150, 191, 215–216
API Blueprint 124
API capabilities, identifying
 avoiding integrating too specific consumers' perspective 41–43
 differentiating steps from operations 38–39
 identifying unique and versatile operations 39–40
 overview of 24–28
 refining steps to identify operations 38–40
API capabilities, nominal paths 31–35

API Capabilities Canvas 28–30
 observing from REST
 angle 55–57
 observing operations from
 REST angle 57
 overview of 29–30
 reorganizing and
 expanding 56
 tools to use along with 30

API consumer 4

API design
 bulk operations 321–325
 handling data and files
 337–342
 layered 16–18
 long operations 353–356

API design context, REST API
 alternatives 356–358

API design guidelines 428–429
 automating 442–446

API design reference kit, provid-
 ing overview of API design
 with OpenAPI 468–471

api-guidelines rule 440

API-KEY 373

API-KEY header 373

API-KEY HTTP header 373

API linter 435–437

API provider 4

APIs (application programming
 interfaces)
 user-friendly, API names
 258–259

*Architectural Styles and the Design
 of Network-based Software
 Architectures* (Fielding) 94

architecture, design guidelines
 405

arrays
 contrasting JSON Merge
 Patch and JSON Patch
 for array updates
 318–319
 grouping data with 191
 sorting data in 192
 types 158, 168

artifacts
 design guidelines 405
 enhancing and adapting for
 implementers 478–479

AS (authorization server) 271

AsyncAPI 124

atomic types and formats
 186–189
 formatting numbers as
 strings 187
 managing dates and times
 188–189
 managing non-human-read-
 able codes 187–188

authoring data model in
 OpenAPI 152–153

Authorization 271, 287, 294, 344

Authorization header 272

Authorization HTTP
 header 271, 287, 294, 344

automating API design guide-
 lines
 API linting 433–435
 organizing rules 457–458
 Spectral 437–439
 using when designing
 APIs 458–459

B

behavior, standardized 229–230

BFF (backend-for-frontend) 325

BLOCKED status 241–242

body property 324

Boolean type 156

breaking changes 365–373
 invisible contract 372
 modifying HTTP statuses 371
 modifying input data 368–370
 modifying operation
 flows 372
 modifying operations or their
 HTTP methods 371
 modifying output data
 365–367
 modifying resource paths 370
 preventing unintended
 modifications 372–373

browsing, interoperable API
 browsing with HTTP and
 hypermedia APIs 259–263

bulk operations 321–325
 create bulk behavior
 scope 325
 designing all-or-nothing
 responses 324
 designing mixed responses
 323–324

designing requests 321–322

error policy 322–323

optimizing request
 responses 324

optimizing requests 322

partitioning access to 325

C

cache 94–95

Cache-Control 313

Cache-Control header 313

Cache-Control HTTP header
 313

caching 94–95
 enabling 311–314
 overview of 311–312

callback APIs
 avoiding polling with 354
 describing with OpenAPI
 354–356

capabilities, identifying, avoid-
 ing exposing provider's
 perspective 43–46
 business logic 44–45
 data organization 43
 software architecture 45–46

casing function 438

changelog, adding 387

CI/CD (continuous integration
 and continuous delivery)
 373, 434

CISO (Chief Information Secu-
 rity Officer) 276

CLI (command-line interface)
 434
 Spectral 438

client/server separation 94–95

code-first approach 129–131

code on demand 94–95

complete models 114–115

COMPLETE status 249

completion flag 249

components block 420

conditional readings
 enabling 311–314
 overview of 311–312

conditional updates, enabling
 and enforcing 283–284

Confirm-Duplicate
 true 283
 true HTTP header 283

consistentReferences function
 452, 454
 const keyword 420
 constraints, integrating in API
 design 17
 consumer 4
 Content-Length 344
 Content-Length HTTP
 header 344
 content negotiation, using to
 select hypermedia or plain
 JSON format 220, 262
 content property 140, 144, 339,
 456
 Content-Range 344
 Content-Range HTTP header
 344
 Content-Type
 application/json 315
 application/json HTTP
 header 315
 application/xml 221
 application/xml HTTP
 header 221
 text/csv 221
 text/csv HTTP header 221
 Content-Type HTTP header
 application/cloudevents+json
 349
 application/hal+json 262
 application/json 221, 315
 application/json-patch+json
 319
 application/merge-
 patch+json 319
 application/pdf 222, 224
 application/problem+json
 226
 application/vnd.siren+json
 262
 application/xml header 221
 text/csv header 221
 context, integrating into API
 design 331–334
 challenging constraints and
 limitations 333–334
 contextual factors affecting
 design 332
 making trade-offs 334
 seeking constraints and limita-
 tions during design 333
 Conway's law 46

Correlation-Id 448–450
 correlation-id-defined rule 449
 Correlation-Id HTTP
 header 448–450
 correlation-id-standard rule 450
 create actions, representing with
 HTTP methods 79
 create operations
 choosing HTTP statuses
 for 87
 inputs and success outputs
 111
 creation models 115
 CRM (customer relationship
 management) tool 47
 cross-API consistency 426–428
 defining library of reusable
 components 426
 ensuring library files are
 editable independently
 427–428
 using shared component in
 API 427
 CRUD (create, read, update,
 and delete) operations,
 input and output data
 modeling 113–116
 CSV (comma-separated
 values) 221
 curl command 51–52
 currency enumeration 369
 currency property 150, 191
 cursor-based pagination 320

D

data
 centralizing redundant data
 in dedicated operations
 317
 designing user-friendly,
 interoperable 177–178
 granularity and scope
 192–194
 grouping with arrays 191
 grouping with objects
 189–191
 handling 337–342
 interoperability, standardiza-
 tion 197–199
 modeling 113–118
 organizing 189–192

 selecting and crafting ready-
 to-use data 184–186
 sorting in arrays and objects
 192
 standardization 197–199
 standardized 229
 translating 222
 tweaking returned data 222
 user-friendly and interoperable
 179–181, 201–202
 user-friendly names 195–197
 data array 324, 391, 451, 472
 data integrity 281–284
 correctly implementing HTTP
 methods 282
 corrupting data with regular
 API calls 281
 enabling and enforcing condi-
 tional updates 283–284
 preventing request replay 283
 data list 322, 477
 data models
 defining consistent 413–419
 inputs and outputs 108–113
 JSON Schema 471–473
 data object 316, 339, 342
 data property 220, 349
 data-saving flows 246–250
 carefully aggregating saving
 operations 248
 constraining consumer
 flow 247
 enabling full and partial data-
 saving flows 249–250
 enabling partial data-
 saving 247–248
 redirecting consumer to final-
 ized resource 250
 smoothing validation and sep-
 arating from completion
 249
 data schema 477
 date property 215–216, 367
 dates and times, managing
 188–189
 date string format 188
 date-time string format 188
 DDD (domain-driven
 design) 257
 DDoS (distributed denial of
 service) 269
 deep references 414–415

- default Boolean flag 191
- defined function 448–449
- Define stage
 - asking why to investigate any problem 41
 - focusing on proper perspectives 41
 - output of 25–26
 - staying within needs scope 40–41
- delete actions, representing with HTTP methods 78–79
- DELETE HTTP method 77–80, 282, 335, 338
- delete operations
 - choosing HTTP statuses for 87
 - inputs and success outputs 113
- deprecated flag 385
- describing data
 - deriving complete resource model to create other reusable models 165
 - mixing inline schema and reference 167–169
 - using references to resource models in request bodies 167
 - using references to resource models in response bodies 163–165
- describing with OpenAPI, resource paths
 - describing path 135
 - describing path with path parameters 135–136
 - initiating OpenAPI document 134–135
- description 475
 - element 478
 - field 192, 424–425, 468
 - fields 387
 - object 468
 - overriding when using \$ref 415–416
 - property 109, 140, 144, 159
 - rule 456
- design, web APIs
 - defined 2–6
 - interface for others 5–6
 - interface to implementation 4–5

- remote interface for applications 2
 - uses HTTP protocol 3–4
- design artifacts, mocking and prototyping during design 479–481
- design context
 - consumer and provider constraints 334–337
 - file management features 343–344
- design decision-making 395–410
- design-first approach 129
- design guidelines 402–403
 - automation, Spectral 437–439
 - benefits of 402
 - building 406–408
 - content of 403–405
 - governance and 403
 - when to use 403
- design guidelines, automating
 - checking element values 447–454
 - feedback when problems detected 454–457
 - returning problem-solving message 456
 - splitting rules due to severity or message concerns 457
 - stating importance or nature of problem with severity 455
- designing, APIs
 - adapting to context, provider-sourced events with webhooks 347–353
 - optimizing design 305–307
 - optimizing pagination 319–320
 - profiles needed to design web APIs 13
 - stakeholders influencing web APIs 13–14
 - step by step 15–16
 - when to design 10–12
 - who designs web APIs 12–14
- designing for, user needs and user-friendliness 308–310
- designing operation flows 240–246

- design of APIs
 - importance of 6–10
 - analogy to kitchen appliances 6–7
 - benefits of good design 9–10
 - poor design affecting developers and architecture 7–8
 - poor design affecting end-user and third-party experiences 9
 - poor design affecting security and infrastructure 8–9
- design questions, researching solutions to 399–402
- design reference kits 464–482
- Django 399
- document optimization, defining consistent request bodies 422–423
- documents array 339, 342
- do operations 92–94, 116
 - focusing on results 94
 - turning actions into business concepts 93–94
 - using action resources 93
- downloads 345–346
 - downloading files from another system 345
 - enabling partial downloads 344
 - enabling partial uploads 344
- DPO (Data Protection Officer) 276
- Dungeons and Dragons 233
- DX (developer experience) 177

E

- each 153
- EDA (event-driven architecture) 357
- efficiency
 - concerns about 305
 - design and 305
 - designing for user needs and user-friendliness 307–310
 - enabling caching and conditional readings 311–314
 - inefficient APIs 303–304
 - overview 303–305

- efficient API design, separate optimized APIs 325–327
- elements
 - checking values 447–454
 - deprecating 385
- encoding object 342
- enriching API design artifacts, examples 473–477
- enum array 473
- Error data model 405
- Error-Id 411
- Error-Id HTTP header 411
- Error JSON schema 165, 297
- Error model 113
- errors 294–297
 - avoiding disclosing implementation details on server errors 295–296
 - choosing HTTP statuses for 87–88
 - consumer errors 223–227
 - ensuring exhaustive error-handling 88
 - handling missing scopes or permissions 294–295
 - handling token-related errors 294
 - providing implementation details in response descriptions in OpenAPI 296
- Etag 313
- Etag header 283, 313
- Etag HTTP header 313
- example
 - field 475
 - key 474–475
- examples 473–477
 - adding examples of parameters, request and response bodies, and headers with OpenAPI 475
 - adding property examples with JSON Schema 474
 - authoring accurate and realistic 476
 - connecting to each other 477
 - sharing OpenAPI examples across operations 476–477
- exclusiveMaximum keyword 471
- exclusiveMinimum keyword 471

- execution mode, choosing with Prefer header 356
- Expect
 - 100-continue 344
 - 100-continue HTTP header 344
- Expect, 100-continue header 344
- extends keyword 457, 459
- extensibility, creating extensible API designs 381–384
- external API 5
- externalDocs object 468, 470
- external shared components 426–428

F

- failure paths 35–38
 - analyzing for each step 36–37
- falsy function 448
- features, standardized 230
- feedback, providing informative, problem-solving 224–225
- Fielding, Roy 94
- fields, enabling field selection 316
- file management features 343–344
- files 337–342
 - collecting in flow 338
 - describing mixed data and files with OpenAPI 341–342
 - describing with OpenAPI 341
 - retrieving with single call 340
 - sending with single call 338–339
- file-type library 341
- filtering lists 216–220
- flexible filters 217
- guessable filters that map returned data 216–217
- minimizing filters 218
- pagination 218–219
- q filter 217
- returning filter, sort, and pagination metadata 219–220
- sorting with helpful defaults 218
- formats, handling different data formats 221
- function rule 456

G

- GDPR (General Data Protection Regulation) 275
- Geometry object 181
- GeoJSON 181–183
- GET HTTP method 51–54, 72, 78, 127–128
- governance, design guidelines and 403
- granularity, request and response data 228
- GraphQL Schema 124
- greater-than-or-equal-to filters 400
- grouping data
 - with arrays 191
 - with objects 189–191
- gRPC (Google Remote Procedure Call) 124
- guessable filters that map returned data 216–217

H

- Header OpenAPI object 145, 162
 - reusing headers object 423–424
 - reusing response headers 423–424
- headers map in Response OpenAPI object 145, 162
- helpful defaults, sorting with 218
- HTTP compliance 208, 214, 282
- HTTP headers
 - enhancing response with rate-limiting headers 307
 - Link header, providing pagination, formats, and resources links with 261
- HTTP (Hypertext Transfer Protocol) 3–4
 - choosing input data locations in requests 80–84
 - choosing output locations in HTTP responses 90–92
 - ensuring configuration efficiency 306
 - interoperable API browsing with 259–263

HTTP (Hypertext Transfer Protocol) (*continued*)

- operations, successful responses 214–215
- overview of 51
- representing actions with HTTP methods 77–80
- representing operations with 71–73
- representing output types with HTTP statuses 84–90
- REST principles for API design 94–96
- statuses 84–90

HTTP methods

- consumer limitations 334–335
- correctly implementing 282
- DELETE 77–80, 282, 335, 338
- GET 51–54, 72, 77, 79
- modifying operations or their 371
- OPTIONS 259–260, 264
- PATCH 77–79, 81–82, 87, 208
- POST 51, 79, 87, 332, 348, 353
- PUT 77–79, 137, 208
- representing actions with 77–80

HTTP operations

- describing inputs 138–140
- describing operation output contents 143–145
- describing with OpenAPI 122–137
- modifying operations or their 371
- representing actions with 77–80

HTTP Server-Sent Events (SSE) 357

HTTP status codes

- 100 Continue 344
- 200 OK 71
- 201 Created 424
- 204 No Content 144
- 207 Multi-Status 323
- 400 Bad Request 224
- 404 Not Found 84, 144, 413
- 405 Method Not Allowed 371

- 412 Precondition Failed 284
- 413 Content Too Large 344
- 415 Unsupported Media Type 224
- 422 Unprocessable Content 224
- 429 Too Many Requests 372
- 503 Service Unavailable 336, 461
- choosing adequate 214
- modifying 371
- using adequate 224

hypermedia APIs, interoperable API browsing with 259–263

I

IAM (Identity and Access Management) API 233

Idempotency-Key 283

Idempotency-Key HTTP header 283

identifier models 114–115

identifiers

- defining naming pattern for 198–199
- using well-known or standard identifiers consistently 198

id property 198, 215, 229, 401, 415, 417, 462

IETF (Internet Engineering Task Force) 226

If-Match 284, 370

- z567dff 283
- z567dff HTTP header 283

If-Match header 370

If-Match HTTP header 284, 370

If-None-Match 314

If-None-Match header 314

If-None-Match HTTP header 314

If-Unmodified-Since 284

If-Unmodified-Since HTTP header 284

implementation, design guidelines 405

implicit flow 293

index-based pagination 320

inefficient APIs 303–304

info object 135

info property 443

inline schemas 167–169

- describing non-body request parameters with 161
- describing response headers with 162–163

in property of OpenAPI Parameter object 136, 139, 145, 444

input data

- choosing locations in HTTP requests 80–84
- data models 108–113
- determining 33–34
- modeling 113–116
- modifying 368–370

int32 format value for JSON Schema 157

int64 format value for JSON Schema 157

integer type of JSON Schema 156

internal API 5

interoperability

- atomic types and formats 186–189
- interoperable API browsing with HTTP and hypermedia APIs 259–263
- operations, successful responses 214–215
- standardization 197–199

interoperable APIs 17, 254–257

interoperable data

- consistency and standard 181

interoperable operations, adapting request and response data 220–222

invisible contract 372

J

jq command-line tool 479

JSON (JavaScript Object Notation) 103–104, 132

- using content negotiation to select hypermedia or plain JSON format 262

JSON Merge Patch 318–319, 337

JSON Patch 318–319

JSON pointer 163

JSON Schema 148–170,
471–473
 adding property examples
 with 474
 authoring data model in
 OpenAPI, adding com-
 plete resource data
 models to document
 152–153
 checking partial JSON
 schemas 451
 contrasting with
 OpenAPI 151
 describing complete resource
 data models with
 153–160
 describing data while design-
 ing it 151
 describing number or ele-
 ment size range 471–472
 describing operation body
 data 163–169
 describing value with pattern,
 enum, and default
 472–473
 enforcing expected error data
 with 297
 in OpenAPI, describing oper-
 ation non-body data
 161–163
 overview 149–151
 JSON Schema keywords
 allOf JSON Schema keyword
 418
 const JSON Schema keyword
 298, 449–450
 contains JSON Schema
 keyword 449–450
 default JSON Schema
 keyword 472
 enum JSON Schema
 keyword 472
 examples JSON Schema
 keyword 474
 exclusiveMaximum JSON
 Schema keyword 472
 exclusiveMinimum JSON
 Schema keyword 472
 items (array) JSON Schema
 keyword 158
 maxItems JSON Schema
 keyword 472

maxLength JSON Schema
 keyword 472
 minItems JSON Schema
 keyword 472
 minLength JSON Schema
 keyword 472
 pattern JSON Schema
 keyword 298, 472
 properties JSON Schema
 keyword 155–160
 required JSON Schema
 keyword 158–160
 type JSON Schema
 keyword 155–160

K

keys, performing basic checks
 on values and keys 447–448

L

layered API design 16–18
 layered system 94–95
 limit query parameter 422, 473
 Link 259–262
 Link header, providing pagina-
 tion, formats, and resources
 links with 261
 Link HTTP header 259–262
 linters 125
 linting 433
 linting rules, deciding what to
 verify 439–442
 lists
 embedding in resource
 model 193–194
 filtering, sorting, and
 paginating 216–220
 locale, adapting to 222
 Location 11, 90, 92–94, 163, 250
 Location header 90, 102, 111,
 145, 163, 214, 250, 316, 324,
 344–345, 351, 367, 370, 424
 Location HTTP header 11, 90,
 92–94, 163, 250
 location object 179, 183
 long operations 353–356
 avoiding polling with callback
 APIs 354
 choosing execution mode
 with Prefer header 356

describing callbacks with
 OpenAPI 354–356
 starting and monitoring
 status with polling
 353–354

M

machine-readable feedback
 225
 MADR (Markdown any decision
 records) 401
 magic identifiers 213
 MAJOR.MINOR.PATCH
 version 375
 maximum keyword 471
 media types 140
 application/cloudevents+json
 349
 application/hal+json 262
 application/json 140, 144,
 262, 315–316, 319
 application/octet-stream 340,
 342, 346
 application/vnd.bankingapi.
 complete+json 315
 application/vnd.bankingapi.
 summarized+json 315
 application/vnd.siren+json
 262
 application/xml 221, 224–225,
 336
 multipart/form-data 338–340
 text/csv 220–221, 224, 315
 message property of errors 113,
 230
 messages list, error 249
 metadata object 220
 minimal models 114–115
 minimizing filters 218
 minimum keyword 471
 missing scopes 294–295
 mixed responses 323–324
 mocking 479–481
 creating basic mock with
 OpenAPI 480
 creating functional API
 tests during design
 480–481
 favoring early prototype over
 complex mock during
 design 480

- modeling data 99–119
 - ensuring completeness and proper focus 117–118
 - input and output data modeling 113–116
 - overview of 101–105
 - theoretical resource data models 105–108
- models 113–116
- modifying APIs 361–388
 - assigning version to API 374–378
 - creating extensible API designs 381–384
 - describing design modifications with OpenAPI 384–387
 - designing modifications 363–365
 - identifying breaking changes and ensuring backward compatibility 365–373
 - identifying security-breaking changes and preventing breaches 373
 - overview of concerns 362–365
 - potential problems 363
 - versioning APIs 378–381

N

- names
 - of APIs 258–259
 - user-friendly 195–197
- naming
 - pattern for identifiers 198–199
 - structuring consistently 199
 - typing consistently 199
- nominal paths 31–35
- non-atomic parameter serialization 162
- non-body data 161–163
- non-body parameters 138–139
- non-body request parameters, describing with inline schemas 161
- non-HTTP-compliant web APIs 53
- number type 156

O

- oas Spectral rules 479
- oauth2 OpenAPI security scheme type 293
- object resource properties, choosing 106–107
- objects, describing in JSON schema
 - adding properties to 155
 - describing 154
 - describing properties of 157
 - sorting data in 192
- OOP (object-oriented programming) 52
- OpenAPI
 - adding complete resource data models to document 152–153
 - adding examples of parameters, request and response bodies, and headers with 475
 - authoring documents 129–132
 - contrasting with JSON Schema 151
 - describing callbacks with 354–356
 - describing design modifications with 384–387
 - describing files with 341
 - describing HTTP operations with 122–146
 - describing mixed data and files with 341–342
 - describing operation body data 163–169
 - describing operation inputs 138–140
 - describing operation output contents 143–145
 - describing operations 136–137
 - describing resource paths 133–136
 - describing scopes with 292–293
 - describing webhooks with 353
 - enhancing or adapting for code generation 479
 - initiating document 134–135
 - JSON Schema in 161–163, 170–171
 - linting document with Spectral CLI 438
 - optimizing documents, enhancing API design guidelines 428–429
 - providing implementation details in response descriptions 296
 - providing overview of API design with 468–471
 - shared components 441
 - sharing examples across operations 476–477
- OpenAPI documents
 - optimizing 413–428
 - targeting elements to check in 442–446
- openapi field 135
- OpenAPI objects and keywords
 - Callback OpenAPI object 355
 - components keywords 153, 293, 422–427
 - external components 426–427
- Content OpenAPI object 140, 143–144
- deprecated OpenAPI keyword 385–386
- Example OpenAPI object 475–477
- examples OpenAPI keyword 474
- externalDocs OpenAPI keyword 468, 470
- files, OpenAPI 341–342
 - describing with OpenAPI 341
 - mixing data and files 342
- Header OpenAPI object (response) 145, 163, 424
- reusable headers 424
- Info OpenAPI object 135
 - description keyword 469
- Operation OpenAPI object 137
- Parameter OpenAPI object 136, 139, 161–162, 419–422
- Path parameter 136
- Query parameter 139, 161–162
- Request header 136
- serialization 162

OpenAPI objects and keywords (continued)

- Path Item OpenAPI object 135
- Paths OpenAPI object 135
- readOnly OpenAPI keyword 417
- Reference Object (\$ref keyword) 164, 167, 298, 414–416
- Request Body OpenAPI object 140, 422–423
 - reusable request bodies 422–423
- Response OpenAPI object 141–142, 297–298, 425
 - reusable responses 245
- Schema OpenAPI object 150
 - example OpenAPI keyword 298, 474
 - reusable schemas 153, 164–169, 298, 413–416
- security OpenAPI keyword 294
- Security Scheme OpenAPI object 292–294
- tags OpenAPI keyword 386, 469–470
- webhooks OpenAPI keyword 352–353
- writeOnly OpenAPI keyword 416–418
- openapi ruleset 458
- OpenAPI Specification 124
- operation body data,
 - describing 163–169
 - deriving complete resource model to create other reusable models 165
 - mixing inline schema and reference 167–169
 - using references to resource models in request bodies 167
 - using references to resource models in response bodies 163–165
- operation flows 235–251
 - designing 240–246
 - designing flexible data-saving flows 246–250
 - modifying 372

- optimizing 238–239
 - user-friendly and interoperable 236–238
- operations 203–232
 - adapting request and response data 220–222
 - avoiding hiding multiple capabilities in single operation 227–228
 - consistency and standardization 228–230
 - designing 207–210
 - differentiating steps from 38–39
 - ensuring that implemented operations behave according to context 278–281
 - handling consumer errors 223–227
 - identifying unique and versatile 39–40
 - limiting access to with scopes 287
 - modifying 371
 - refining steps to identify 38–40
 - representing with HTTP 71–73
 - requesting easy-to-provide inputs 210–214
 - successful responses 214–215
 - user-friendly and interoperable 204–205
- optimizing OpenAPI documents
 - defining consistent data models 413–419
 - defining consistent parameters 419–422
 - defining consistent request bodies 422–423
 - defining consistent responses 423–425
 - ensuring cross-API consistency with external shared components 426–428
 - volume of data 314–319
- OPTIONS HTTP method 259–260, 264
 - listing resource operations with 260
- organizing data 189–192

- grouping data with arrays 191
- grouping data with objects 189–191
 - sorting data in arrays and objects 192
- output data, choosing locations in HTTP responses 90–92
- outputs, data models 108–113
- output types, representing with HTTP statuses 84–90
- overrides list, ignoring Spectral problems 459
- overriding, descriptions when using \$ref 415–416
- OWASP (Open Web Application Security Project) 273

P

- page size limits, optimizing 319
- pagination 216–220
 - flexible filters 217
 - guessable filters that map returned data 216–217
 - minimizing filters 218
 - optimizing 319–320
 - q filter 217
 - returning filter, sort, and pagination metadata 219–220
 - sorting with helpful defaults 218
- Parameter OpenAPI object 145, 421
- parameters in OpenAPI 128, 136, 138–139, 161
 - consistent, defining 419–422
- path-level 419–420
 - reusable groups of query parameters 421–422
 - reusing 420–421
- partial downloads and uploads 344
- partial updates 318, 337
- partner API 5
- PATCH HTTP method 77–79, 81–82, 87, 208, 370
 - requests 370
- path JSON pointer for JSON Patch 318
- path-level OpenAPI parameters 419–420

paths, representing resources
with 73–76

pattern, JSON Schema
keyword 472
Spectral function 448

permissions, handling missing
294–295

perspectives, focusing on
proper 41

PII (personally identifiable
information) 275

polling 347, 353, 356
avoiding with callback APIs
354
starting long operations and
monitoring status
with 353–354

Postel's law 223

POST HTTP method 51, 79,
87, 140, 208, 332, 347–348,
353, 367

Prefer 315, 356

Preference-Applied 315

Preference-Applied HTTP
header 315

Prefer header 315–316, 356

Prefer HTTP header 315, 356

presigned URL 345

principles and rules, design
guidelines 404

private API 5

Problem Details for HTTP
APIs 227, 296, 397, 399,
404

profiles 32

programming interface,
describing 16, 123–128
contrasting OpenAPI docu-
ment with API spread-
sheet 127–128
OpenAPI during design
125
OpenAPI Specification 124
while designing it 128
YAML format 126

properties, JSON schema's
adding to objects 155
array 158
atomic 155–157
indicating required 107
listing and modeling
efficiently 108

object 157
required 158–160

property-camel-case Spectral
rule 456

property key, JSON schema
103

property names and types,
choosing 107

property-names Spectral
rule 448

Protocol Buffers 124

prototyping 479–481
creating basic mock with
OpenAPI 480
creating functional API tests
during design 480–481
favoring early prototype over
complex mock during
design 480

provider 4

provider's perspective, avoiding
exposing 43–46
business logic 44–45
data organization 43
software architecture 45–46

public API 5

PUT HTTP method 77–79, 137,
208, 238, 322, 370

Q

q filter 217, 222

query-name-camel Spectral
rule 438

R

RAML (RESTful API Modeling
Language) 124

Range 219

Range HTTP header 219

RateLimit 307

RateLimit HTTP header 307

rate-limiting
enhancing response with
headers 307
limiting API usage with 306

RateLimit-Policy 307

RateLimit-Policy HTTP
header 307

RBAC (Role-Based Access
Control) 287

read actions or operations, rep-
resenting with HTTP
methods 78–79
choosing HTTP statuses
for 86
inputs and success outputs
109

reference kits 464–467
contents of 465–466
deploying APIs with 467
designing APIs with 466
developing APIs with 466
enriching 467
providing and consuming
APIs with 467
testing APIs with 466

references, OpenAPI \$ref
checking with Spectral 450
overriding descriptions when
using 415–416
to resource models in request
bodies 167
to resource models in
response bodies 163–165
using deep references in
OpenAPI 414–415

\$ref property 163–164, 167,
415–416

replacement models 115

replace operation 318

replaying requests,
preventing 283

representing operations with,
do operations 93–94

request bodies, in OpenAPI
140
consistent request bodies
422–423
using references to resource
models in 167

Request Body, OpenAPI
object 140

requestBody, OpenAPI
property 140

requestBody Open API
property 140

request data, granularity
227–228

Request-Unique-Id 283

Request-Unique-Id HTTP
header 283

required flag 109, 145

- required property (JSON Schema) 109, 136, 145, 160
- researching solutions to design questions 399–402
- searching and
 - considering 400
- using architectural decision record format 401–402
- where to research solutions to design questions 399–400
- resolved, flag of a Spectral rule 445
- Resource actions, identifying 61–66
 - dealing with contradictory successes and failures when listing outputs 65–66
 - dealing with operation's resource when listing action inputs 63–64
 - listing action inputs 62
 - listing action outputs 64–65
 - overview of 61–62
- resource identifiers 75, 81
 - choosing between resource identifiers and modifiers 83–84
 - choosing locations for 82
- resource-id-number Spectral rule 439
- ResourceLocation 424
- ResourceLocation HTTP header 424
- resource models
 - deriving complete resource model to create other reusable models 165
 - resource data models, describing complete with JSON Schema 153–160
 - selection 315–316
 - using references to in request bodies 167
 - using references to in response bodies 163–165
- resource modifiers 81
 - choosing between resource identifiers and 83–84
 - choosing locations for 83
- resource paths
 - combining meaningful with HTTP compliance 208
 - crafting short but accurate 209–210
 - creating predictable 208–209
 - describing with OpenAPI 133–136
 - modifying 370
- resource representations 81
 - choosing locations for 83
- resources
 - defined 58
 - embedding lists in resource model 193–194
 - modeling embedded 194
 - operation's 59
 - patterns and recipes 60–61
 - relations 60
 - representing with paths 73–76
 - toggleing return of updated or created 316
- Response, OpenAPI object 163
- response bodies, Open API
 - describing 143–144
 - using references to resource models in 163–165
- response data, granularity 227–228
- response headers, with OpenAI 145
- responses, OpenAPI object 141–144
 - consistent, defining 423–425
 - reusing response headers 423–424
 - reusing 424–425
 - without bodies, dealing with 144
- REST APIs
 - alternatives to 356–358
 - designing user-friendly, interoperable 381–382
 - identifying resource actions 61–66
 - identifying resources and relations 58–61
 - observing API Capabilities Canvas from 55–57
 - observing operations from REST angle, programming interface design 50–55
 - overview of 52
 - principles for API design 94–96
 - REST (Representational State Transfer) 95
- Retry 352
- Retry-After 336, 354
- Retry-After header 336
- Retry-After HTTP header 336, 354
- Retry HTTP header 352
- reusable groups of query parameters, in OpenAPI 421–422
- reusable response 425
- reusing parameters, in OpenAPI 420–421
- RFC and drafts (IETF)
 - 2119 404, 455
 - 6648 287
 - 6901 163
 - 6902 113, 318
 - 7240 315
 - 7396 113, 318
 - 7516 287
 - 7519 287
 - 7946 181
 - 8188 287
 - 8594 384
 - 9110 73, 93, 284, 314
 - 9111 313
 - 9335 442
 - 9421 287
- RateLimit header fields for HTTP 307
- The Idempotency-Key HTTP Header Field 283
- RS (resource server) 271
- rubber ducking 398
- rules, Spectral 458
 - automating design guidelines 454–457
 - message 456, 461
 - organizing 457–458
- rulesets, Spectral, importing and tweaking 458–459
- rules key, in Spectral rulesets 443

S

SAML (Security Assertion Markup Language) 287

Schema, OpenAPI object 153

schema, OpenAPI property 128, 136, 140, 144, 151–153, 161–162

schema, Spectral function 449–451, 457

schema-object-required, Spectral rule 449

schemas, JSON

- defining complete from summary 418
- mixing inline schema and reference 167–169
- optimizations 419
- reusing 413–414
- targeting part of with deep reference 414–415

scopes 287–292

- deciding which types to use 292
- describing with OpenAPI 292–293
- end-user- or consumer-based 291
- for read or write operations 291
- limiting access to operations with 287
- measuring importance of 288–289
- operation-based 289
- resource-, concept-, or use-case-based 290–291
- tweaking operation behavior with 292

scopes, OpenAPI property 293

search operations 242, 440–441

- choosing HTTP statuses for 87
- inputs and success outputs 109–110
- representing with HTTP methods 78–79

security 269–299

- design-related problems 272–273
- ensuring data integrity 281–284

- ensuring that implemented operations behave according to context 278–281

errors 294–297

exposing only necessary data and operations 275–278

overview of 270–273

protocol- or architecture-based security problems 284–287

scopes 287–292

- when and how to handle during design 273–275

security list, in OpenAPI Operation Object 293

- security property 353

security schemes, in OpenAPI 293

self relation 261

semantic-version, Spectral rule 443

sensitive operations and data 275

separate optimized APIs 325–327

server errors 295–296

signed URL 345

similarly named elements 116

SMEs (subject matter experts) 13, 32, 54, 125

sorting, filtering, and paginating lists 216–220

- flexible filters 217
- guessable filters that map returned data 216–217
- minimizing filters 218
- pagination 218–219
- q filter 217
- returning filter, sort, and pagination metadata 219–220
- sorting with helpful defaults 218

sorting data in array and objects 192

specification-first approach 129–131

Spectral 437–439

- editing rulesets 438–439
- linter 438

Spectral keywords

- aliases Spectral keyword 446
- documentationUrl Spectral keyword 441
- extends Spectral keyword 457–459
- field Spectral keyword 447–448
- @key 447
- function Spectral keyword 437
- defined 448
- pattern 447
- schema 448
- undefined 443
- given Spectral keyword 437, 443
- JSON Path 437, 442–444
- message Spectral keyword 454–457
- message placeholders 444, 456
- overrides Spectral keyword 458–459
- resolved Spectral keyword 445
- rules Spectral keyword 440, 459
- severity Spectral keyword 454–455
- then Spectral keyword 443–447

Spring Boot 399

SSE (HTTP Server-Sent Events) 357

standardization 197–199

- defining naming pattern for identifiers 198–199
- local, domain, or global 197–198
- naming, typing, and structuring consistently 199
- using well-known or standard identifiers consistently 198

statelessness 94–95

string, JSON schema type 156

style OpenAPI parameter serialization property 162

subschemas, defining in JSON Schema 414

success outcomes of use case's step, determining 33–34

summarized models 114–115
 summary property, OpenAPI
 operation 135, 137, 140
 Swagger Editor Next 131
 Swagger Specification 124

T

tags, OpenAPI 387, 469
 document 469
 Operation object 387, 469
 tags list 387
 testing
 functional API tests 480–481
 reference kits 466
 text/csv media type 221
 theoretical resource data
 models 105–108
 choosing object resource
 properties 106–107
 choosing property names and
 types 107
 determining resource
 structure 105
 indicating required
 properties 107
 listing and modeling proper-
 ties efficiently 108
 TLS (Transport Layer Security)
 351
 tokens, handling token-related
 security errors 294
 total replacement 318
 Transaction-Id 283
 Transaction-Id HTTP
 header 283
 type property, JSON
 Schema 160, 193, 367

U

UI (user interface), avoiding
 mapping consumers' 42
 undefined, Spectral
 function 442, 449
 uniform interface 94–95
 update operations
 choosing HTTP statuses for 87
 inputs and success
 outputs 112–113
 representing with HTTP
 methods 79

uploads and downloads 345–346
 URLs, ensuring data and URL
 compatibility 337
 use cases
 adding alternative branches
 on each 37–38
 analyzing alternative 38
 analyzing spotted elements
 34–35
 decomposing in steps 33
 listing 32
 spotting missing elements
 with sources and
 usages 34
 user-friendly, interoperable
 APIs 17
 API names 258–259
 clear purpose 254–255
 creating one or multiple
 APIs 255–257
 defined 254–255
 discovery and navigation 255
 interoperable API browsing
 with HTTP and hyperme-
 dia APIs 259–263
 user-friendly, interoperable
 data 175, 179–201
 atomic types and formats
 186–189
 consistency 180
 helping find and interpret
 information 179
 limiting consumers' work 180
 meeting user needs 179
 when and how to design
 181–183
 user-friendly, interoperable
 operations
 adapting request and
 response data 220–222
 filtering, sorting, and paginat-
 ing lists 216–220
 how to design 206–207
 successful responses 214–215
 when and how to design
 205–207
 when to take into
 consideration 206
 user-friendly names 195–197
 designing simple, clearly
 organized, concise
 names 195–196

learning by fixing non-user-
 friendly names 196–197
 when to design 195
 user needs and user-
 friendliness 307–310
 analyzing inefficient flow
 308–309
 lessons learned 308
 optimizing each operation
 309–310
 rethinking flow 310
 users, identifying 32
 UTC (Coordinated Universal
 Time) 223

V

validation, separating from
 completion 249
 Vary 313
 Vary header 313
 Vary HTTP header 313
 Version 377, 420
 Version header 419–421
 Version HTTP header 377, 420
 versioning APIs 378–381
 accumulating trade-offs or
 breaking regularly 381
 assigning version to API
 374–378
 avoiding sub-API-level
 versioning 377–378
 balancing effects and benefits
 of breaking changes 380
 checking whether consumers
 use what we break 379
 choosing API version
 identifier 375–376
 choosing how to represent
 API version in request
 377
 complying with API versioning
 policy 380
 determining whether it's pos-
 sible to expose multiple
 API versions 380
 differentiating interface and
 implementation
 versioning 374–375
 how API version can be
 represented in request
 376–377

- versioning APIs (*continued*)
 - indicating 384–385
 - listing consumers and their types 379
 - when to choose API
 - version scheme and representation 377
- volume of data, optimizing 314–319
 - centralizing redundant data in dedicated operations 317
 - considering partial update over total replacement 318
 - contrasting JSON Merge Patch and JSON Patch for array updates 318–319
 - enabling field selection 316
 - enabling resource model selection 315–316
 - toggling return of updated or created resources 316

W

- WADL (Web Application Description Language) 124
- Warhammer 233
- web APIs
 - defined 2–6
 - designing 10–16
 - interface for others 5–6
 - interface to implementation 4–5
 - remote interface for applications 2
 - uses HTTP protocol 3–4
- webhooks 347–353
 - choosing event data granularity 349–350
 - dealing with failures 351–352
 - defining expected behavior 351
 - describing with OpenAPI 353
 - designing operations 348
 - designing securely 350–351
 - overview of 347

- should be optional 348
- using standard event format 348–349

- WSDL (Web Services Description Language) 124

X

- X-HTTP-Method-Override 335
 - PATCH 335
 - PATCH HTTP header 335
- X-HTTP-Method-Override custom header 335
- X-HTTP-Method-Override HTTP header 335
- X-RateLimit 307
- X-RateLimit headers 307
- X-RateLimit HTTP header 307
- x-something properties 478

Y

- YAML (YAML Ain't Markup Language) 126
- yq command-line tool 479

The Design of Web APIs Second Edition

Arnaud Lauret • Foreword by Kin Lane

Web APIs are a way to connect your software with external applications. They unlock features of your site for other developers to use and should support good system performance and end-user experience. This book shows you how to design APIs your fellow developers will love.

The Design of Web APIs, Second Edition teaches you to design efficient and adaptable REST APIs. This revised and rewritten second edition contains the latest updates to the OpenAPI standard, along with insights you can apply to other API styles such as GraphQL. Learn vital skills for gathering requirements, creating easy-to-consume public and private web APIs, and handling non-backward compatible modifications and versioning.

What's Inside

- Design reusable, user-friendly and interoperable APIs
- Document your APIs with OpenAPI and JSON Schema
- Create secure and efficient APIs by design
- Streamline and standardize API design decisions

Written for developers with experience building and consuming APIs.

Arnaud Lauret runs the *API Handyman* blog and is a frequent speaker at API conferences. He currently works as an API Industry Researcher at Postman.

For print book owners, all digital formats are free:
<https://www.manning.com/freebook>

“Brings together excellent knowledge, experience, and practices. Two thumbs up!”

—Mike Amundsen
Amundsen.com, Inc

“Distills API design into a clear and understandable approach that will stand the test of time.”

—James Higginbotham
APICoach.io

“You need this book on your bookshelf! Clear, informative, and practical.”

—Lorna Mitchell
OpenAPI Initiative

“Design APIs with confidence! Arnaud’s approach is both easy to follow and instantly applicable.”

—Joyce Stack, Elsevier

 **FREE**
eBook
see first page

ISBN-13: 978-1-63343-814-9



9 781633 438149