FROM SCRATCH

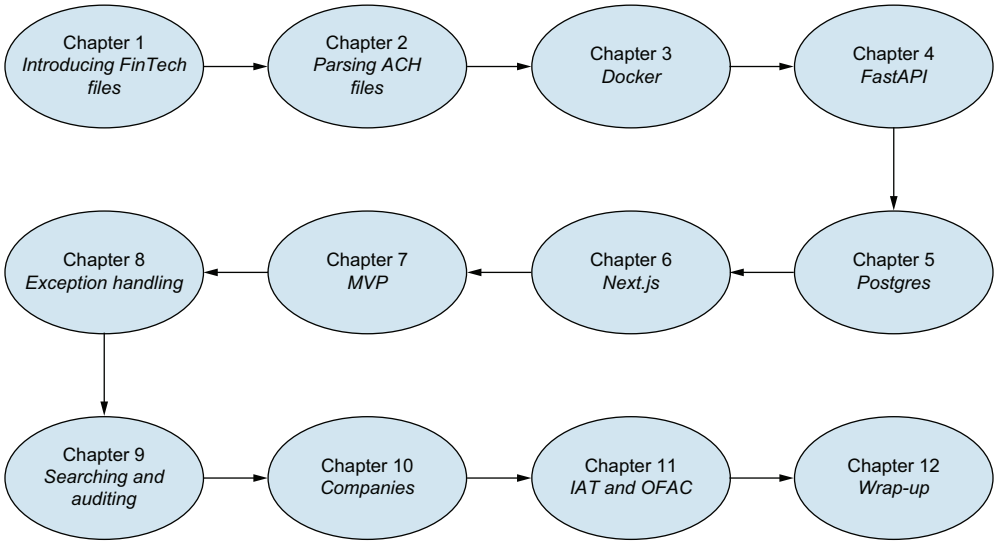
BUILD

Financial Software with Generative AI

Christopher Kardell
Mark Brouwer

MANNING

Building financial software with generative AI roadmap



Build Financial Software with Generative AI (From Scratch)

Christopher Kardell
Mark Brouwer

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 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 author and publisher have made every effort to ensure that the information in this book was correct at press time. The author 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:	Ian Hough
Technical editor:	Laurens Meulman
Review editor:	Radmila Ercegovac
Production editor:	Kathy Rossland
Copy editor:	Lana Todorovic-Arndt
Proofreader:	Olga Milanko
Technical proofreader:	Saranya Balasubramaniyan
Typesetter:	Tamara Švelić Sabljić
Cover designer:	Marija Tudor

ISBN 9781633436626
Printed in the United States of America

To our friends and families who supported us through the making of this book.
We appreciate your support and patience!

brief contents

PART 1 FINTECH AND THE AUTOMATED CLEARING HOUSE 1

1 ■ Exploring FinTech and generative AI 3
2 ■ Parsing payments 29

PART 2 A PATH TO THE MINIMUM VIABLE PRODUCT 63

3 ■ Getting started with Docker 65
4 ■ APIs: The piping between our components 94
5 ■ Storing our ACH files 127
6 ■ Taking the next step with Next.js 163
7 ■ Our minimum viable product 194

PART 3 BEYOND THE MINIMUM VIABLE PRODUCT 237

8 ■ Exceptions 239
9 ■ Searching and auditing 281
10 ■ Company information 311
11 ■ International ACH transactions and OFAC scanning 355
12 ■ Where to go from here 393

contents

preface xii
acknowledgments xiii
about this book xiv
about the authors xvii
about the cover illustration xviii

PART 1 FINTECH AND THE AUTOMATED CLEARING HOUSE.....1

1 ***Exploring FinTech and generative AI 3***

1.1 The Futuristic FinTech project 4

1.2 ACH dashboard 5

How important is the ACH? 6

1.3 SAFe Agile 7

1.4 The ACH dashboard 9

Tools of the trade 13

1.5 A first look at generative AI 13

A better rubber duck 14 ▪ Hello, Check 21! 14 ▪ Help! This code is not working! 18 ▪ Help! I forgot the syntax! 21 Did I just give away my data? 24 ▪ Privacy and security 26 Staying up to date 26

2 *Parsing payments 29*

2.1 Modernizing our legacy software 29

2.2 Understanding the ACH 31

2.3 Parsing an ACH file 32

Asking ChatGPT to parse an ACH file 34 ▪ *Parsing an ACH file with Copilot 38* ▪ *Generative AI: Trust but verify 40*

2.4 Automated testing 41

Testing the number of records read 42 ▪ *Parsing the records 42* *File header record (type 1) 45* ▪ *Batch header record (type 5) 48*

2.5 Entry detail record (type 6) 50

Addenda record (type 7) 51 ▪ *Batch control record (type 8) 52* *File trailer record (type 9) 53* ▪ *Passed! 54*

2.6 The not-so-happy path 54

File rejection 55 ▪ *Batch rejection 56* ▪ *Entry rejection 57*

2.7 Interpreting the code 57

Where's my switch statement? 57 ▪ *Type hints 58* *Secure coding 59*

PART 2 A PATH TO THE MINIMUM VIABLE PRODUCT 63

3 *Getting started with Docker 65*

3.1 Where to begin 66

3.2 Creating a docker-compose.yml file 68

Creating a CloudBeaver container 70 ▪ *Create a PostgreSQL container 72* ▪ *API container 74* ▪ *Web server container 76*

3.3 Connecting our containers 76

Connecting to the database 77 ▪ *Troubleshooting PostgreSQL and FastAPI 78*

3.4 Calling our API 79

Troubleshooting our web site 82

3.5 Container security 83

3.6 Optimizing Docker 85

General maintenance 86 ▪ *Optimizing image size 87* *Optimizing build time 88*

3.7 Removing the hardcoded username/password 89

3.8 Health checks 92

4 *APIs: The piping between our components 94*

4.1 API-driven design 94

4.2 Our APIs 96

4.3 Building our APIs with FastAPI 97

Retrieving files with GET /files 99 ▪ Adding a file with POST /files 102 ▪ Finishing the rest of the APIs 104

4.4 Adding our API to Docker 104

Preparing the project 105 ▪ Docker setup 105

4.5 Documenting our API 106

Swagger 107 ▪ Redoc 108 ▪ What is so great about API documentation? 108

4.6 Managing our APIs 109

Benefits of API management 109 ▪ Adding WSO2 API manager to our project 110 ▪ OpenAPI Version 111 ▪ Create an API in WSO2 112 ▪ Troubleshooting 113

4.7 Testing with Postman 114

4.8 Load testing with Locust 116

4.9 Improving our code 121

Using a prefix and versioning 121 ▪ Adding Pydantic 122 Validation with Pydantic 123 ▪ Troubleshooting Pydantic 125

5 *Storing our ACH files 127*

5.1 Designing our database 128

5.2 Using SQL directly 132

Adding records to the ach_files table 133

5.3 Storing the file header record 138

Using generative AI 138 ▪ Full example 139

5.4 Storing the rest of our ACH records 143

Storing ACH files challenge: Lessons learned 143

5.5 Storing exceptions 145

5.6 Uploading an ACH file 147

5.7 Storing records with integrity 152

5.8 Using Pydantic 153

5.9 Lessons learned 156

5.10 Coding changes 159

Creating Pydantic schema for the unparsed ACH records 159
Creating Pydantic schema for the parsed ACH records 159
Unit test changes 160

5.11 Design and different approaches 160

6 *Taking the next step with Next.js 163*

6.1 Getting started with Next.js 164

6.2 Creating an upload page 166

Testing our upload page 166

6.3 Refactoring our navigation 168

6.4 Converting recent order to recent ACH uploads 170

6.5 Mocking our ACH uploads data 175

Adding WireMock to Docker 176 ▪ Setting up mappings in WireMock 177 ▪ Accessing endpoints with axios 179

6.6 Expanding the UI with an ACH batch view 181

Defining our WireMock data for the ACH batch view 182
Updating the dashboard to take us to the batches screen 184
Displaying batch details 185

6.7 Creating a ScatterPlot component 186

Scatterplot component 187 ▪ Scatterplot custom tooltip 189
Scatterplot custom ticks 189 ▪ Using colors for the chart 190

6.8 Deploying the dashboard in Docker 192

7 *Our minimum viable product 194*

7.1 Which minimum are we talking about? 195

7.2 Preparing for the MVP 196

7.3 Building out the /files APIs 197

Testing the API 204

7.4 ACH batches APIs 207

Adding the /files/id/batches API 207 ▪ Supporting ACH batches in the dashboard 213 ▪ Uploading files 215

7.5 Putting the puzzle together 217
Cleaning up the configuration files 217 ▪ Ensuring APIs are accessible 220 ▪ Programming challenges 220 ▪ Exploring the dashboard 221

7.6 Gotta have more files 221

7.7 Adding a basic login page 224

7.8 Adding TLS to our dashboard 229

7.9 Testing with Playwright 231

7.10 MVP wrap-up 233
Troubleshooting the MVP 233

PART 3 BEYOND THE MINIMUM VIABLE PRODUCT..... 237

8 *Exceptions 239*

8.1 Planning enhancements 240

8.2 Exceptions and warnings 241

8.3 Creating ACH files with exceptions 244
Creating ACH files with file-level exceptions 244 ▪ Creating ACH files with batch-level exceptions 245 ▪ Creating ACH files with entry-level exceptions 246 ▪ Clean-up of our file creation syntax 246

8.4 Expanding the ACH processing for exceptions 247
ACH exceptions in the database 247 ▪ Recovery options 249

8.5 Updating the ACH parser 250
Creating unit tests for the exceptions 251 ▪ Supporting the exceptions in our parser 252 ▪ Validation in Pydantic 255

8.6 Updating the APIs 256
Retrieving an unparsed record 262

8.7 Insights into exceptions with the UI 263
Showing all the exceptions 264 ▪ A visual cue for exceptions 267

8.8 Additional UI changes 269
Creating an exceptions component 269 ▪ Batch statistics component 271 ▪ Adding the application 275 ▪ Making our UI more responsive 277

9 *Searching and auditing 281*

9.1 Sprint planning 282

9.2 Searching ACH transactions 283

BDD testing for searching 284 ▪ *Search API 286* ▪ *UI search page 291* ▪ *Integration testing with Playwright 295*

9.3 Auditing user interactions 297

Database design 297 ▪ *TDD testing for auditing 298* *Auditing logic 300*

9.4 Using middleware for our logging 301

9.5 Viewing the audit log 303

Creating the page 303 ▪ *Creating the page components 304* *Adding the APIs for the audit log 308*

10 *Company information 311*

10.1 Sprint planning 312

10.2 Company Information 313

Database design: Company information 313 ▪ *Using generative AI to generate mock data 317* ▪ *API design 319* *User interface 322*

10.3 Company dashboard 326

Navigation for the company dashboard 326 ▪ *Creating ACH files for our companies 327*

10.4 Company Information component 328

10.5 Recent Batches component 333

Implementing the recent batches API 333 ▪ *Displaying recent batches 336*

10.6 Expected Files component 337

Storing expected files 338 ▪ *Implementing the expected files API 339* ▪ *Displaying the expected files 343*

10.7 Company Limits component 344

Storing the company limits 345 ▪ *Implementing the company limits API 346* ▪ *Displaying the company limits 349*

11 *International ACH transactions and OFAC scanning 355*

11.1 Sprint planning 356

11.2 International ACH transactions 357

IAT batches: An overview 357

11.3 Creating IAT batches 359
Database changes 361

11.4 IAT record parsing 363

11.5 IAT file processing 367
Unit testing 368 ▪ Updating file processing 372

11.6 Effects on the dashboard 375
The get_batches method 375 ▪ Batch entries 376

11.7 OFAC scanning 378
Sanctioned individuals and countries 378 ▪ Scanning for individuals 379

11.8 Application programming interface 383
Unit test 384 ▪ Creating the endpoint 385 ▪ Finishing the API 385

11.9 User interface 387
The OFAC page 387 ▪ OFAC components 388
OFAC results 390

12 *Where to go from here 393*

12.1 Backend processing 394
Asynchronous processing 394 ▪ ORM frameworks 395
Application programming interfaces 395

12.2 Infrastructure 396
Docker 396 ▪ Multitenant environment 396 ▪ Continuous integration and delivery 397 ▪ Testing 397 ▪ Database 397

12.3 User interface 398
Charts and forms 398 ▪ Configurable dashboards 400
Mobile versionv400

12.4 Finance 400
Accounts and customers 400 ▪ ACH origination and returns 401 ▪ Positive Pay 402 ▪ Service charges 402

12.5 Final thoughts 402

index 404

preface

We have been fortunate to have successful and rewarding careers in financial technology (FinTech). Having spent a combined 50 years working in this industry—ranging from large Fortune 500 companies to small consulting companies with a few dozen employees—we have learned quite a few things. During this time, we have explored many different languages used in FinTech such as Python, Java, COBOL, Visual Basic .NET, and C, to name just a few. Our primary area of expertise is core banking development, which is a broad area that consists of everything needed to run a financial institution. We have explored tasks from handling the depositing of funds and transferring that money, to withdrawing it from an ATM and having that all reflected on bank statements.

One critical piece of core banking software is the ability to work with the Automated Clearing House (ACH), which facilitates payments between both businesses and individuals. From working with existing ACH systems to designing and developing new systems, we have had the opportunity to work with ACH throughout our careers. This includes both working with the backend for processing ACH and providing user interfaces for bank employees to work with it on the frontend.

In our tenure as developers, we have seen companies struggling to attract talented developers because, traditionally, finance is not an area where developers first set their sights when thinking about job opportunities. This book evolved from an idea to share our experiences in software development and our passion for FinTech. The project that you will be building in this book uses modern technology applied not only at FinTechs but anywhere software development projects may be found. It was important for us to introduce both tools and techniques that we use in our work on a daily basis. We also wanted readers to have an opportunity to build a project that is an integral part of the world of FinTech, reflecting the process of software development as we have practiced it.

acknowledgments

We would like to thank our colleagues who provided insight and advice throughout the development of the book. We especially want to thank Manning Publications and all the tech reviewers that helped get this book across the finish line.

Special thanks go to our technical editor Laurens Meulman, who is Director of AI Product Management at BNY Mellon, where he spearheads applied AI initiatives across the enterprise. With a background in international business studies and econometrics from Maastricht University, Laurens transitioned from his previous role as a data scientist to his current leadership position. Drawing from his expertise at the *Financial Times* and several financial institutions, he navigates the intersection of finance and technology, driving innovation and efficiency through AI solutions.

To all the reviewers—Adam Shannon, Aditya Pant, Aliaksandra Sankova, Amol Gote, Andrey Makarov, Antowan Batts, Arun Chandramouli, Bhavin Desai, Daniel McAlister, David Perkalis, Eleni Tsilgianni, Felipe Provezano Coutinho, Hariskumar Panakkal, Harsh Daiya, Hasan Banna, Ionuț Arțărișii, Jose Antonio Martinez, Jose Luis Manners, Kapil Patil, Laud Bentil, Lucian Torje, Luis Fernando Barrera, Manash Chakraborty, Manju Rupani, Manoj Kumar Vandanapu, Maxim Volgin, Michal Krokosz, Michael Schleichardt, Murugan Lakshmanan, Peter V. Henstock, Piyush Ranjan, Rajat Gupta, Rajeev Reddy, Ravi Malhotra, Saranya Balasubramaniyan, Saurabh Shukla, Siddharth Parakh, Simeon Leyzerzon, Sumit Pal, Todd Jobson, and Viktoria Dolzhenko—your suggestions helped make this a better book.

about this book

Build Financial Software with Generative AI (From Scratch) was written to introduce readers to various technologies and tools used within real-world FinTech companies. This book has two main objectives—to fire the imagination of both new and existing software developers in FinTech to the potential possibilities a career can take you; and to introduce a variety of topics in software development, technologies, and finance with the hope that you find at least one area you will want to explore further. We created a real-world project, using real-world techniques and technologies that we believe will benefit developers, regardless of their industry or experience.

Who should read this book

Build Financial Software with Generative AI (From Scratch) is for developers interested in full stack development with a FinTech spin. Both beginner and experienced developers should find something worthwhile as we touch on many topics. We try to present a full Agile development cycle and work on the project throughout the book in an Agile manner. While this book is intended for a wide audience, a basic understanding of programming in a high-level language such as Python or Java is assumed, as we do not cover basic programming constructs such as loops, conditional statements, and similar.

How this book is organized: A road map

This book has 12 chapters split into three parts. The first part discusses FinTech and generative AI and provides the background for our project—developing software for a fictional FinTech company.

- Chapter 1 introduces you to FinTech, setting up the background for why we are creating the ACH dashboard, and explores generative AI.

- Chapter 2 dives into parsing payments made via the ACH, providing an overview of the file structure and record types.

In the second part, we begin organizing the pieces of our project by exploring the technologies and building a minimum viable product (MVP).

- Chapter 3 explores the use of Docker and setting up the containers required for the ACH dashboard project.
- Chapter 4 reviews the use of the FastAPI framework for Python to build APIs and OpenAPI documentation.
- Chapter 5 discusses the database design in Postgres for storing ACH files loaded through the dashboard.
- Chapter 6 explores using Next.js to build sample components for the ACH dashboard.
- Chapter 7 puts all the pieces together into an MVP that you will be able to turn over to the stakeholders.

The third part takes the MVP we have built and starts a series of enhancements based on feedback from various stakeholders involved in the project.

- Chapter 8 enhances the ACH dashboard by expanding on the exceptions that are encountered and handled when processing an ACH file.
- Chapter 9 expands the functionality of the ACH dashboard by adding the ability to search and provide an audit trail.
- Chapter 10 updates the capabilities of the ACH dashboard by defining company information in the database and making it available through the dashboard.
- Chapter 11 explores international ACH transactions and OFAC scanning using fuzzy logic techniques available in Postgres.
- Chapter 12 wraps up the project by providing ideas for inspiring the reader to dive deeper into the topics covered.

The book is meant to be read without skipping, but depending on your familiarity with the technologies, you may choose to skip sections or entire chapters. For instance, if you have a solid understanding of the technologies used and just want to start enhancing the MVP, you could start with part 3 and just work through the enhancements.

About the code

This book makes heavy use of source code primarily in numbered listings throughout the chapters. The source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also in **`bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we have added line breaks and reworked indentation to accommodate the available page space in the book.

In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listing when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/build-financial-software-with-generative-ai. Source code for the examples in this book is available for download through GitHub from the publisher's website at www.manning.com/books/build-financial-software-with-generative-ai.

liveBook discussion forum

Purchase of *Build Financial Software with Generative AI (From Scratch)* 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 authors and other users. To access the forum, go to https://livebook.manning.com/book/build-financial-software-with-generative-ai/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 authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Chris Kardell is a senior software engineer at Digital Era Banking Services. With 20 years of experience working in the financial industry, from IT support at a small community bank to an IT Architect at a Fortune 500 company, he has held a variety of roles. He holds Master's degrees in both computer science and data analytics and enjoys all aspects of software development, as well as staying up to date on the latest technologies and trends.

Mark Brouwer has over 30 years of experience in the FinTech industry, specializing in large systems and complex project design. He has 15 years of experience as a Nacha Accredited ACH Professional with a deep knowledge of COBOL, Visual Basic .NET, and shell scripting. His knowledge allows him to design robust software that helps financial institutions serve their customers quickly and efficiently.

about the cover illustration

The illustration on the cover of *Build Financial Software with Generative AI (From Scratch)* is "Turc, comme il va d'ici à la Mecque," or "A Turkish man in the costume worn when traveling to Mecca," a part of the *Costumes Turcs* album from circa 1790.

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.

Part 1

FinTech and the Automated Clearing House

This part sets the stage for the project that we will be building throughout the book—developing software for a fictional FinTech company.

Chapter 1 gives a bit of background on FinTech, discussing the role of the Automated Clearing House (ACH). It also covers some foundational concepts necessary when working with generative AI tools.

Chapter 2 goes deeper into the ACH, exploring the file structure and record types that we will be dealing with throughout the book.

Once you have made it through the first two chapters, you should have a basic understanding of ACH to begin working on the ACH dashboard.

1 Exploring FinTech and generative AI

This chapter covers

- A brief introduction to FinTech
- What the ACH is and why it is important
- How to use full stack development to build an ACH dashboard
- The benefits and pitfalls of generative AI

The term "FinTech" became common in the last few decades as shorthand for financial technology. In fact, *financial technology* is a term that could be applied to inventions from long ago, such as the 19th-century cash register and telegraph, and even the earliest coin purses from circa 3,300 BCE. Today, FinTech helps empower crowdfunding opportunities through websites such as GoFundMe and Kickstarter. Gig service apps such as Uber and DoorDash rely on FinTech to facilitate fast payments. Payment services such as Venmo and Zelle can offer fast person-to-person (P2P) and business-to-business (B2B) payments thanks to FinTech. Cryptocurrencies like Bitcoin and Ethereum are certainly classified as FinTech, but even the more traditional banking, investing, and credit companies have FinTech at their core.

What is FinTech, then? Simply put, it is using technology to improve the handling of money. Today, FinTech refers to software that offers quicker, more convenient, and/or new ways to handle money.

As you can see, a FinTech developer has the opportunity to work on many kinds of software. It should not be hard to imagine that the career paths open up for full stack FinTech developers, or those who can work on all aspects from frontend to backend. We are fond of saying that a full stack developer gets to be curious about many different areas. As a full stack developer, you will have opportunities to work on the different pieces that compose an application—from the user interface (UI) that customers will see down to the backend processing that fulfills requests from the UI and everything in between.

Imagine receiving a reported problem about a customer interaction that is taking too long and ruining the user experience (UX). Where other developers will have to throw in the towel at one of their assigned boundaries, you will have the opportunity to trace that problem from the button click, through the application programming interface (API), to the server, and down to the specific Structured Query Language (SQL) statement to see how it can be addressed.

This book will teach you about full stack development using a real-world FinTech project. The book aims to provide a high-level understanding of several aspects of software development encountered at FinTech companies, both large and small. After working through the project presented in the following chapters, you should have a better understanding of the areas of software development that interest you most and which topics you may want to dive deeper into. In much the same way that FinTech and full stack provide us with many different opportunities, we hope to present the reader with similar opportunities for learning both FinTech and software development. The varied topics should fire your imagination and encourage you to dive deeper into one or more topics.

1.1 The Futuristic FinTech project

This book will take you on a journey as a new software developer with a fictional FinTech company known as Futuristic FinTech. This means that it will be beneficial for you to have a little software development experience in a modern programming language, as we do not aim to teach you programming from the ground up. Instead, you should have a basic understanding of loops, if-statements, and similar. As this book does not focus on a single language or technology, newer developers should find much to learn about, while more seasoned developers should find some new tools or techniques.

To help you put your learning into practice, this book will take on a typical project for a large company such as Futuristic FinTech. This project addresses the need to modernize one of Futuristic FinTech's numerous products. Products that need modernization are usually existing products that have been with the company for a long time. They are usually well established in their marketplace and used by many customers. The company may undertake modernizing these products for any number of reasons, such as

- The company owns a product due to an acquisition or merger that does not meet the company's existing design guidelines.
- A product has come under renewed scrutiny because of new compliance or regulatory changes.
- A product is being repositioned in its marketplace or marketed to a new audience.
- A product requires expensive hardware or additional software to maintain it, and modernizing it will provide a more cost-effective solution.

The project we are undertaking is the updating of a dashboard related to a payment system common for facilitating payment (such as paying a mortgage or car loan) and movement of funds (transferring money from a checking account to a higher-interest savings account at another institution) in the United States—the Automated Clearing House (ACH).

Our ACH dashboard project will require us to gain an understanding of ACH processing and full stack development as we tackle the aspects of database design, APIs, backend processing, and presenting everything within a UI. Along the way, we introduce different approaches and technology that we have worked with as FinTech developers ourselves. We will also explore how generative AI tools can make us more productive in our daily work, helping us to write, troubleshoot, and conduct many other tasks.

1.2 ACH dashboard

As mentioned, we will be building out an example payment system commonly used in US-based payments known as the ACH. However, the ACH system will often interact with equivalent systems in other countries such as

- Electronic Funds Transfer (EFT) in Canada
- Single Euro Payments Area (SEPA) in Europe
- Bankers Automated Clearing System (BACS) in the United Kingdom
- Bulk Electronic Clearing System (BECS) in Australia
- National Automated Clearing House (NACH) in India

ACH payments are a great way to get introduced to FinTech because they are a real-world example of how transactions flow between financial institutions. Once we have transactions flowing into a database, the world of FinTech will really open up to us. We will touch on many areas of software development and FinTech as we build out and then enhance our ACH dashboard, as shown in figure 1.1.

Knowing the pieces that feed into the dashboard helps see the big picture. We can also take a look at how these pieces may come into play when loading an ACH file into the dashboard. Figure 1.2 gives us a high-level overview of the sequence of events that our dashboard will eventually execute once it is built out. Being able to load ACH files is the first step in interacting with the dashboard.

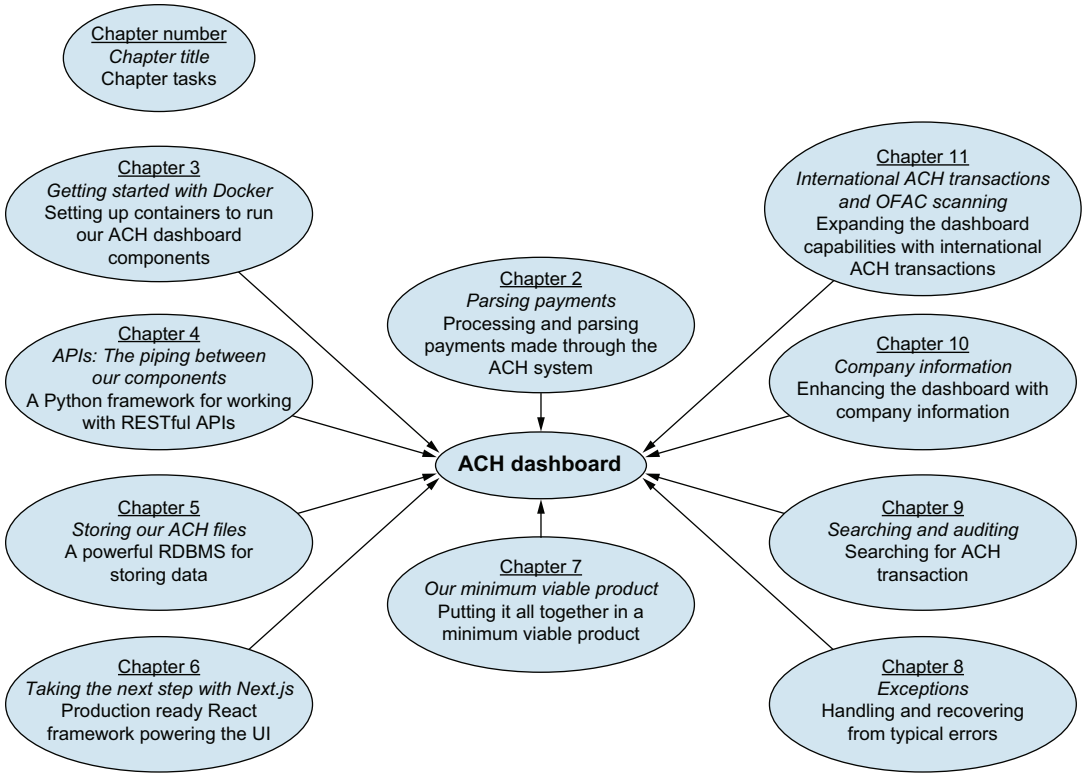


Figure 1.1 Building the ACH dashboard

Once we complete the ACH dashboard, we will take a look at all of the potential paths that are open to our project and hopefully find a fascinating direction to take the project in. While ACH likely affects us on a regular basis (is your payroll automatically deposited, or do you schedule credit card payments, mortgage payments, or other loans?), it is equally likely that you never heard of it before. Let us look at just how important the ACH is.

1.2.1 How important is the ACH?

Figure 1.3 shows the trends in noncash payments in trillions of dollars between 2000 and 2021 as gathered by the latest Federal Reserve Payments Study (https://mng.bz/5g2z). Notice how ACH transfers have been dramatically increasing, taking a larger part of business, especially since the decline of checks.

ACH is a dominant US standard, but there is a rise in the need for real-time payments. Earlier, we had mentioned P2P and B2B. There are other payment types as well, but the underlying idea is a quick real-time payment between two accounts. In response, ACH has developed the idea of same-day ACH, which can considerably speed up the time it takes to process an ACH payment, moving it from days to hours.

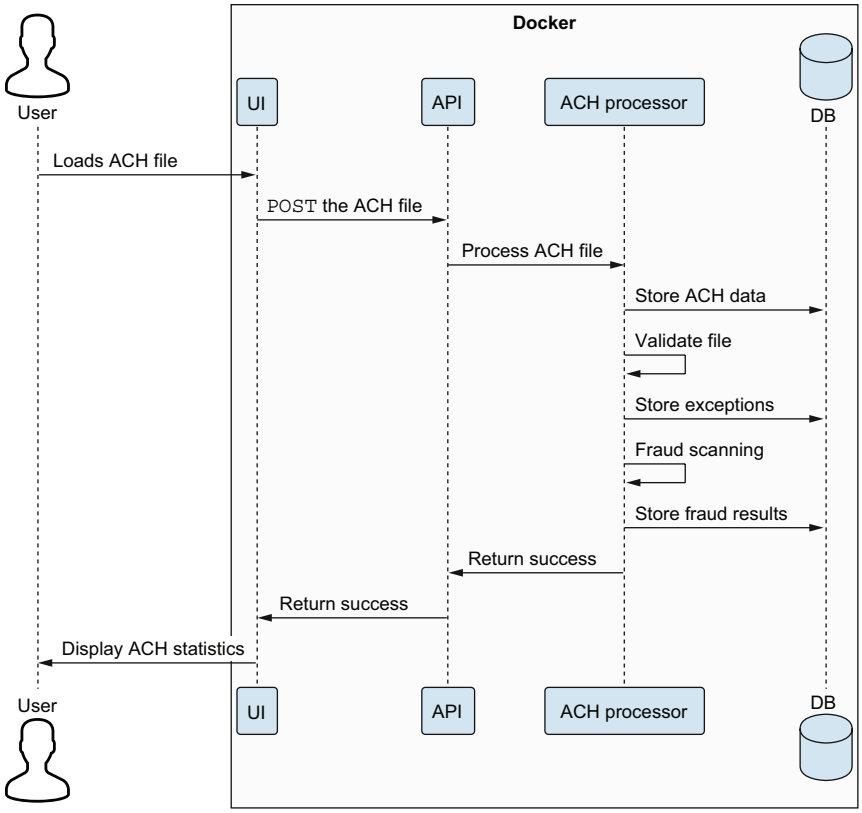


Figure 1.2 Loading a file through the ACH dashboard

Real-time payments and their providers are positioning this standard as the next revolution in payments, and they are likely to gain some traction in the future. Real-time processing is set to be responsible for $10 billion per year by 2026, still a long way to go to affect the trillions of dollars processed by ACH every year. In either case, the lessons learned from this book will give us a handle on payment processing regardless of the technology being used. Now that we have an understanding of ACH, let us take a look at how we are going to accomplish building out this project, which will be done using an Agile methodology known as SAFe.

1.3 SAFe Agile

According to the 17th Annual State of Agile Report, 71% of respondents use Agile in their software development lifecycle (SDLC) (see https://mng.bz/6e2y). This book draws on principles from SAFe Agile because we have direct experience with it at large Fortune 500 FinTech companies. Moreover, Scaled Agile (the creators of SAFe) states that over 20,000 organizations worldwide use the Scaled Agile Framework (SAFe).

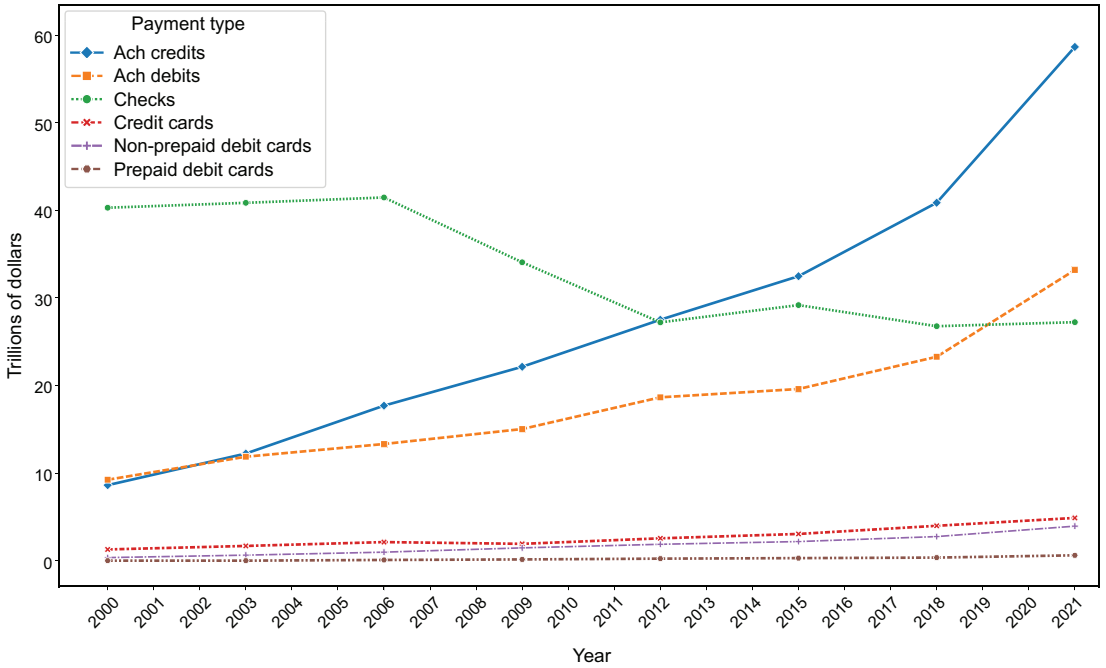


Figure 1.3 Year-over-year comparison of various payment methods

As mentioned, even large FinTech companies that can traditionally lag in the adoption of technology have made the switch to Agile. The large fictitious FinTech company we joined known as Futuristic FinTech is no exception and has adopted SAFe as their approach to development. In the following chapters, as we build out our ACH project, we'll try to adhere to some of the principles of the framework and Agile methodology in general. One of the key events in SAFe Agile is the Program Increment (PI) Planning Session, which helps plan out the work for the coming sprints.

> **DEFINITION** A program increment (PI) is a timebox (a set amount of time) that is anywhere from 8 to 12 weeks long. The PI usually consists of four iterations called "sprints"; that term is from Scrum and has been used by many different software tools (e.g., JIRA). A sprint is typically a two-week period but can be more or less, depending on what the teams decide.

As part of the PI planning, the division displays a product roadmap for the coming year that shows various commitments the business has made, expected regulatory changes, and company initiatives. During this PI planning session, work will be broken up into epics. These epics are larger projects or initiatives. For instance, some companies may call their release an epic (such as the Spring 2024 release), and projects for the release are broken down into user stories. Still others may create their individual projects as epics and break down tasks for that project as user stories. Throughout the book, we

try to work in short development cycles, and that is one of the main concepts of Agile as well. If we are breaking up the work into smaller manageable pieces, we should be okay, regardless of what we call those smaller chunks (of course, having some consistency helps with communication).

The Agile methodology

It should not come as a big surprise that every company is different. Many FinTech companies have their roots in Waterfall methodology but have moved to Agile recently. It is likely that we will encounter groups (divisions, business units, or teams) that follow Agile in different capacities—from the purists to those that have been dragged along kicking and screaming.

Our aim is not to teach Agile per se, but rather to roughly work within the bounds of an Agile framework as we work through our project. If you want to dive into Agile more deeply, we encourage you to take a look at *Becoming Agile* (2009, Manning) by Greg Smith and Ahmed Sidky, and the Agile Manifesto (https://agilemanifesto.org/) is a required reading as well. But don't worry: the manifesto is a quick read!

The PI planning is about coming together as a group, and you will interact with business analysts, QA, developers, tech writers, and so forth to come up with a plan for the coming work. Figure 1.4 shows a PI planning board using a template found in Miro, a popular collaboration tool for just this type of activity.

On the PI planning board, we should be able to find our team and assigned work. In our case, we have been assigned a project to create a new ACH dashboard as part of a companywide initiative to modernize legacy software. Typically, the PI planning session lasts for a day with the focus on prioritizing work, determining dependencies, and hammering out requirements that must be met. In addition, we identify risks using a ROAM (resolved, owned, accepted, mitigated) board and finally take a confidence vote where everyone uses a "fist-of-five" vote where they hold up their hand and vote (or via supported tools). If the average vote is a three or higher, the plan is committed, and we can get to work. If not, we review and rework the plan. Often, there are concerns about the amount of work, time to complete it, and any existing knowledge gaps. Working through these concerns is part of the process, and it can take some time. We are going to fast-forward through that and jump into the project that our team has been assigned since we can always find out more about SAFe and all its moving parts on their website (https://scaledagileframework.com/).

1.4 The ACH dashboard

Coming out of PI planning, we should have a promising idea of our project and the goals we are aiming for within the coming sprints. Our team has picked up the stories for modernizing the ACH dashboard. Modernizing legacy software can be tricky as not only must we deal with the technical challenges of design and development but also consider the effects of our changes on the customers as well. Some industries (and

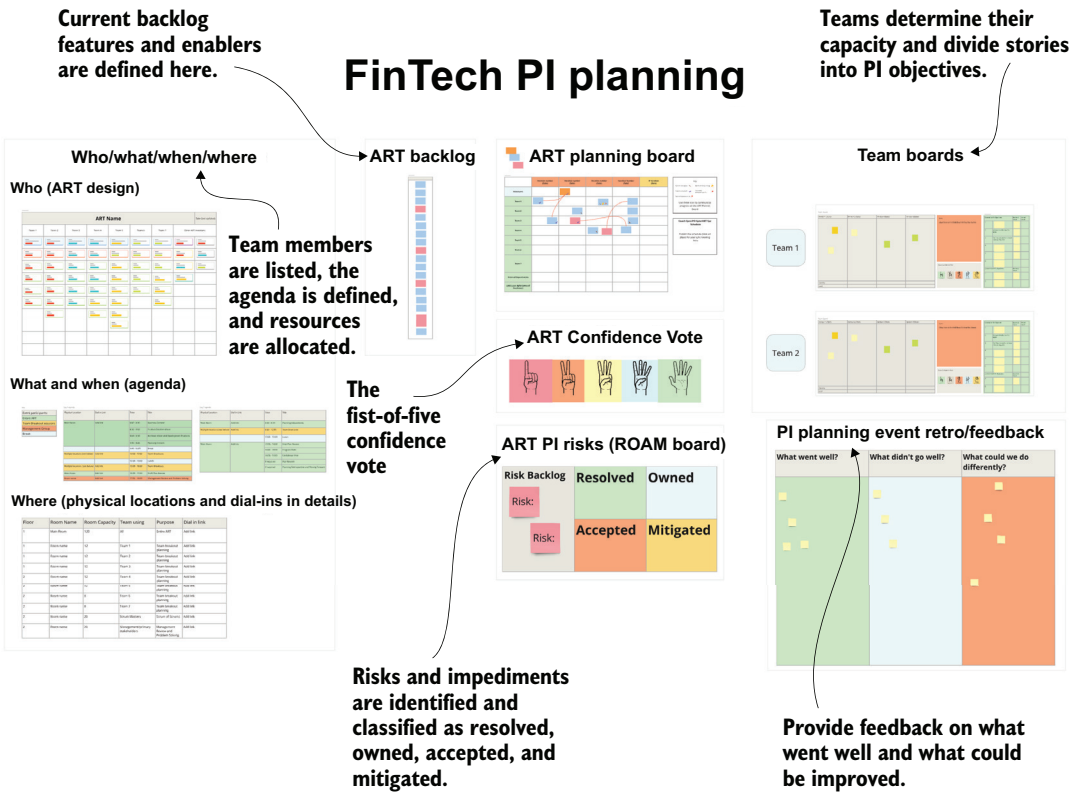


Figure 1.4 PI planning template in Miro

individuals) can be notoriously slow to change. In the case of individuals, they know the current system, processes, and its idiosyncrasies. They have learned how to navigate the system to get their work done, so why change something that works? These are reasonable questions that we should consider for any project.

In our case, the company has treated our ACH dashboard project, as a utility type task—a necessity for business but nothing more. However, since ACH is such a large part of how transactions are being processed by financial institutions, customers have begun asking for enhanced functionality for some time now. To maintain a competitive advantage, the business wants to modernize the dashboard. There is potential for better analytics of the transactions, fraud monitoring, as well as expanding the use of the dashboard to customers originating transactions. With these capabilities on the agenda, we need to look at providing a product that not only provides those capabilities but has a look and feel consistent with the rest of the products offered. Figure 1.5 shows an example of how the legacy interface currently looks.

While the interface is functional, there is room for improvement. The business has reported the limitations that the team should work to address.

The current limitations of the legacy interface are as follows:

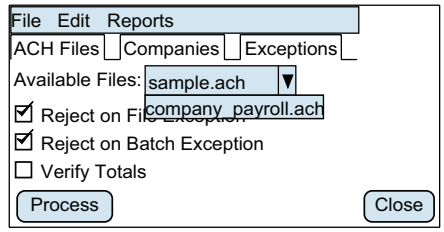


Figure 1.5 Legacy ACH Interface

- *Unable to upload files*—Operations must currently ensure the files are uploaded, and then the back office can select them from the drop-down. This creates an unnecessary delay in processing.
- *Rudimentary reporting*—Only file totals are reported. This method does not produce meaningful analytics. The system stores files in a legacy database that lends itself to reporting.
- *Access limitation*—The interface requires a connection to the backend server running on an IBM Power, while this system can run a webserver. Our SecOps team prefers to offload this task to a different server due to the extensive routing and firewall protection in place for this server.
- *Styling*—The new dashboard will eventually be marketed as a separate component in a wider system or to end-users that manage ACH files. The business wants existing style guidelines to be applied to the new project.

The business has envisioned a modern website (figure 1.6) that will address these concerns and has assigned our team to work on this project.

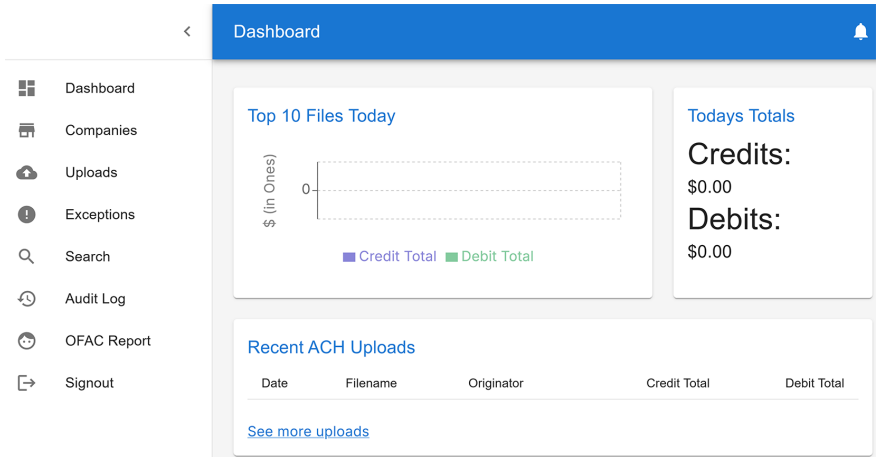


Figure 1.6 Modern ACH dashboard

The technical architects have also provided requirements from the technical side to support the company's modernization initiative. Figure 1.7 provides an overview of the ACH dashboard.

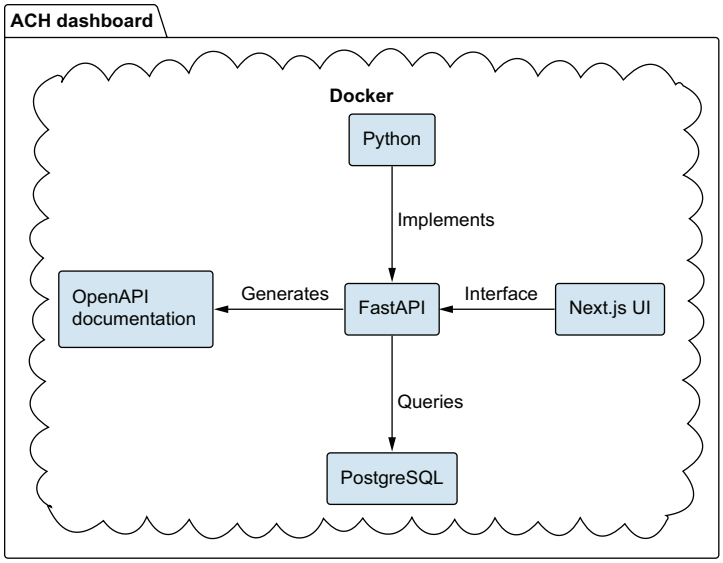


Figure 1.7 ACH dashboard initiative

The modernization initiative includes the following:

- *PostgreSQL*—Ensure a relational database is used and that ACH files are parsed into the database so that the Business Intelligence group can perform analytics on the data.
- *Next.js*—Use a production-ready React framework such as Next.js. Ensure that the interface can support visualizations.
- *Docker*—We want to ensure our application is split into components that are running in containers.
- *Python*—Currently, our legacy applications, such as the ACH processing system, are written in COBOL. New projects should use Python as the primary language.

It should be noted that we can substitute any of these components as we deem necessary (for the purposes of working on this project; on the job, chances are we will not have that luxury). That is something that we will bring up repeatedly in the following chapters. For instance, if you are more comfortable with MySQL or your company uses Java instead of Python and you would be better served by using those technologies, then, by all means, use them. You may have a tougher time reworking the examples from Python into Java (if you do not have experience in one or the other), but you will also learn a lot that way as well. Is there a technology you wish we had used instead? Feel free to reach out and let us know—perhaps we will tackle that next!

We would also like to note that the process of modernization is something that you will come across repeatedly in your career. It may be that all the Python/Java/Golang code in your company will need to be replaced by the next generation of languages at some point in your career. It may be that the company will hire a new chief technology

officer who is more experienced in another technology stack and wants to move in that direction. Regardless of whether the decision is good or bad, we may be forced to endure the transition. Now that we have a bit more understanding of the project, let us take a look at some of the tools we will be using.

1.4.1 Tools of the trade

As part of the ACH dashboard project, the team will be using the following tools. As in the previous section where suggested that readers may choose to replace Python with Java, or choose to use MySQL in place of Postgres— depending on their current technology stack—the same applies to these tools. If there are tools that are more suited to your current environment, please feel free to use those. Many of these tools will be discussed later, but we wanted to give you an idea of the toolset being used.

The tools selected by our team are

- *JetBrains PyCharm*—Either the community or professional edition. We use the Professional edition and run our APIs, UI, and Docker from within it.
- *JetBrains Webstorm*—An additional tool for our UI development in Next.js.
- *Visual Studio Code*—We use it for working with Docker and Plant UML, but it can also be in place of JetBrains PyCharm/Webstorm.
- *JetBrains Qodana*—Used for code review.
- *ChatGPT*—We chose it for coding assistance because there is a free version available for use to experiment with. Also, if our company cannot not provide an integrated experience, we can still use generative AI by accessing https://chat.openai.com/.
- *GitHub Copilot*—It provides AI coding assistance for various IDEs. Having a subscription to GitHub Copilot will let us get assistance in PyCharm, Webstorm, and Visual Studio Code, to name just a few.

We will also introduce various plugins that we use as well; however, we wanted to ensure that we set the playing field before we started to dive into the rest of the book. It is also possible to run the project using a single IDE. Personally, we tended to use JetBrains PyCharm Professional for all the development (Python, Next.js, and Docker) and that worked out great.

Before we dive into the project, let us take a closer look at generative AI. This book is not a comprehensive book on generative AI. For that, you may want to look at *Introduction to Generative AI* (2024, Manning) by Numa Dhamani and Maggie Engler or *Generative AI in Action* (2024, Manning) by Amit Bahree. Instead, we will briefly introduce generative AI and some important concepts when working with it. Then, as we work through the project, we will show some specific cases where we found generative AI helpful (and some where it was a hindrance).

1.5 A first look at generative AI

In their last companywide meeting, Futuristic FinTech's executive team expressed an interest in adopting generative AI tools to help drive efficiencies within the company.

Several initiatives have been started in different company divisions to evaluate different tools and ways to incorporate generative AI. The ACH dashboard project can also use generative AI with the modernization project. The company has asked that we gain a better understanding of the capabilities, limitations, and liabilities of generative AI. So, what is one way we can think of generative AI?

1.5.1 A better rubber duck

We can think of generative AI as the next iteration of the rubber duck. Rubber duck debugging was popularized by the book *The Pragmatic Programmer* by Andy Hunt and Dave Thomas (Addison-Wesley, 1999), and it provides a way of encouraging us to voice our thought process to a less intimidating audience than our boss or team lead! By speaking out loud to a rubber duck (or any inanimate object), we are often able to identify the problem before involving anyone else. We have often called over a co-worker only to point at the screen showing our captive audience how this piece of code could not possibly fail, only to see the immediate problem with our solid piece of code. Having a rubber duck to "bounce" ideas off is an invaluable resource (and quite cost-effective). If we are shy of speaking out loud (even when no one is around), some tools offer the ability to type out our thoughts. Slack has a channel that functions as a sounding board for our thoughts. Entering the channel will display "You can also talk to yourself here, but please bear in mind you'll have to supply both sides of the conversation." There is also a rubber duck chatbot service at https://rubberduckdebugging.com/cyberduck/ and multiple generative AI rubber duck sites such as https://rubberducktherapy.app/.

We can look at generative AI as the evolution of that rubber duck. Generative AI allows our rubber duck to provide ideas and feedback instead of just a sounding board. We will still encourage the use of a rubber duck as well because it provides insight into our thought process and a cheap therapist when our day is long, and our code is uncooperative.

Generative AI is a subset of machine learning that powers tools such as ChatGPT and GitHub Copilot. Generative AI can generate new content and is based on the use of large language models (LLMs). We see generative AI being quickly adopted into workflows in a variety of industries and roles, not just in technical areas but also in areas such as HR and marketing. Generative AI provides us with greater efficiency, which in turn saves time, and we all know time is money.

In the next section, we put a FinTech spin on the old classic "Hello, World!" program with generative AI.

1.5.2 Hello, Check 21!

"Well, where to begin?" we may think when we first see the ChatGPT prompt. We may want to get started with diving right in and seeing the kind of coding chops ChatGPT has, but let us first take a look at some other possibilities with ChatGPT. FinTech is filled with regulations that we need to be aware of, so we first ask it to "Please summarize the Check 21 regulation." After reading the response, we find it to be a little dry as we are software developers, not business analysts. So, while we may need the details

later on, we just want to get a general impression. We can try refining our prompt: "Summarize the benefits of Check 21 for the average bank customer." This gives us something that may be a little more aligned with what we wanted to understand from the beginning. We could refine our prompt further: "Tell me about Check 21 in a couple of sentences," and we again get a suitable response. One interesting thing to ask is "If we had to describe Check 21 in a single word, what would it be?" The response was "Digitalization," which is actually a pretty good way to describe it.

Of course, as software developers, we came here to code! Let us assume that we do not have any Python coding experience, but maybe we have knowledge of other languages and software development in general. We would like to see if ChatGPT can get us going with Python.

Let us take the traditional "Hello, World!" program, but put a FinTech twist on it by instead printing "Current Balance: $1000.00." With ChatGPT, we say, "In Python, show me the code to print a line that says 'Current Balance: $1000.00'" and we get the associated code of print ("Current Balance: $1000.00"). It is important to note that the updates to ChatGPT and the fact that ChatGPT is not deterministic may cause the responses that we receive to vary. We may find that we enter the same prompt multiple times and receive different responses each time. So, if you are following along, it is okay if some of the responses vary slightly.

As you can see, with Python, all we have to do is print a single line; we do not have the overhead of importing libraries or setting up main functions that we may find in Java or C. So, we decided to expand on that program a little bit by telling ChatGPT, "Expand the program to allow the user to enter the current balance amount that is then printed." That also works as expected, giving us a simple program shown in the following listing.

Listing 1.1 ChatGPT-generated code

```
# Ask the user to enter the current balance, note that we expect the user
# to enter a numeric value otherwise we will receive a message similar to
# ValueError: could not convert string to float
current_balance = input("Enter current balance: ")

# Print the entered balance in the desired format
print(f"Current Balance: ${float(current_balance):,.2f}")
```

Accepts input from the keyboard and stores it in current_balance

Prints a line saying Current Balance and formats the current_balance accordingly

Looking at the code, we decide that it may be more useful if we were to break this out into a function. We then ask the system to "Update the sample code to call a function that prints the `current_balance`."

Listing 1.2 ChatGPT updated code with a function

```
def print_balance(current_balance):
```

Defines a function that takes a parameter of the current balance

```
    """
    Prints the current balance in a formatted string.

    Parameters:
    - current_balance: The current balance to be printed.
    """
    print(f"Current Balance: ${float(current_balance):,.2f}")

# Ask the user to enter the current balance
user_input = input("Enter current balance: ")

# Call the function to print the balance
print_balance(user_input)
```

Documentation of the function and its parameters

The print statement has been moved to the function.

Calling the function with the user input

The code is looking better, and it is nice that we received a commented code for the function using Google Style docstrings. However, we believe the code would be more maintainable if it were to include type hints, as mentioned at https://peps.python.org/pep-0484/. So, we then request that the system includes the type hints as well by entering "Please add type hints to the above code."

Listing 1.3 Code expanded with type hints

```
def print_balance(current_balance: str) -> None:
    """
    Prints the current balance in a formatted string.

    Parameters:
    - current_balance (str): The current balance to be printed, as a string.
    """
    print(f"Current Balance: ${float(current_balance):,.2f}")

# Ask the user to enter the current balance
user_input = input("Enter current balance: ")

# Call the function to print the balance
print_balance(user_input)
```

We now show that current_balance is a string and the function does not return anything.

Now, we really like the simple program we have written so far. However, when looking at the return value for `print_balance`, we see it is returning `None`. This makes sense since we are only calling a print statement within the code. However, to make the code even more maintainable, we can just return the formatted string and let the caller decide how they want to handle it. After all, this function should not decide how to handle the output as the caller may want to write it to a file or return it in an API call. We believe it is a good habit not to assume that the return value will always be handled the same way (in this case, printed). Often, we have had to rework large swaths of code because of an assumption that it would always be printed (in one of our cases, printed to an open network connection). The resulting business logic was so intertwined and the deadline so tight that we ended up copying the existing code and ripping out the print statements, so we could return what we needed. This resulted in technical debt

(work that needed to be done in the future to address the duplicated code). Therefore, we want to return just the formatted string to get us into the habit of not assuming our function should be doing something specific with the output. So, we then tell ChatGPT to return a string instead of printing it, "Update the code so that `print_balance` will return the string and then print it."

Listing 1.4 Returning a string

```
def print_balance(current_balance: str) -> str:
    """
    Returns the current balance in a formatted string.

    Parameters:
    - current_balance (str): The current balance to be formatted, as a
     string.

    Returns:
    - str: The formatted current balance.
    """
    return f"Current Balance: ${float(current_balance):,.2f}"

# Ask the user to enter the current balance
user_input = input("Enter current balance: ")

# Call the function to get the formatted balance and then print it
formatted_balance = print_balance(user_input)
print(formatted_balance)
```

The function has been updated to show it returns a string.

Documentation has likewise been updated.

Returns the string instead of printing it

Stores the balance in a variable called user_input, which is of type string

Stores the formatted string in a variable called formatted_balance, which is of type string

Prints the formatted string

It is very interesting that we are able to create a working program with just our software development knowledge and not necessarily any knowledge of Python (maybe we would need to look up typing for Python). While this shows that ChatGPT can be very helpful, it will only be as helpful as the information that we put into the prompt.

Prompt engineering and problem formulation

Generative AI has led to a lot of buzz in the areas of prompt engineering and problem formulation. Somewhat ironically, these are not only subjects for generative AI but probably areas we should have been working on in our daily lives anyway! Prompt engineering is referred to as being able to craft meaningful input (prompts) for the LLM to decipher. After all, the model will need to be able to parse the prompt and form a response, so the input does matter because garbage in results in garbage out more often than not. Thus, prompt engineering focuses on

- Clear and specific prompts
- Keeping prompts concise
- Correct grammar and spelling
- Using examples when appropriate

(continued)

There is a notion that prompt engineering is not that important as models are continuously improving and becoming increasingly better at handling prompts. However, being able to clearly articulate our perspective and keeping it concise is a skill that most of us could benefit from. Have you ever just stopped reading a lengthy email? Yup, us too. While prompt engineering may be something that falls out of favor, there are many different approaches, some of which may currently be necessary depending on the problem you are working with. We recommend checking out various techniques defined at https://www.promptingguide.ai/techniques or *Generative AI in Action* by Amit Bahree (Manning, 2024).

Problem formulation refers to the ability to clearly define our problem. This is another skill that is obviously applicable to daily life, not just to our new generative AI overlords. Would we rather work with someone who asks for our help by saying, "This program is broken, and I sure didn't do anything wrong!" or the co-worker who says, "I tried to speed up execution of this program for the DevOps group as part of ticket 123, and now the formatting is off"?

While having generative AI write our code can be helpful, as new software developers, we will be fielding a lot of support-type work where we need to troubleshoot problems. Let us look at how generative AI may help us with debugging.

1.5.3 Help! This code is not working!

Another benefit of generative AI is the ability to help us identify problems with code. This can be valuable for keeping our sanity or making us look like a hero to our co-workers. For example, let us say we have code that should keep a list of the last transactions a customer made so that we can easily retrieve them in our code. Probably not an unreasonable feature for most financial software. In the following listing, we defined a simple function that keeps a list of transactions.

Listing 1.5 Simple function to store transactions

```
def last_transactions(user_transactions: list,
➥transaction: str, max: int = 3) -> list:
    """
    Adds a transaction to the list of user transactions
 ➥and returns the updated list.
    :param max_length: Maximum transactions to keep
    :param user_transactions: List of current transactions
    :param transaction: New transaction
    :return: Updated list of transactions
    """
    if len(user_transactions) + 1 >= max:
        user_transactions.pop(0)

    user_transactions.append(transaction)

    return user_transactions
```

Defines a function that takes a list of transactions, a new transaction, and the maximum number of transactions to keep; this defaults to three if nothing is passed.

If the length of stored transactions exceeds our maximum, then get rid of the first one.

Stores the new transaction

Returns the updated list of transactions

It's reasonably straightforward—we pass in a list of transactions and the transaction we want to add to the list along with a maximum size to keep the list at. If we exceed the length, we get rid of the first item using the method with a parameter of zero to remove the first item from the list. When we write a program that uses this function, we see that it does not behave quite as expected.

Listing 1.6 Using our function

```
from source.last_transactions import last_transactions

transactions: list[str] = []

transactions = last_transactions(transactions, "deposit $100")
transactions = last_transactions(transactions, "deposit $200")
transactions = last_transactions(transactions, "deposit $300")
transactions = last_transactions(transactions, "deposit $400")

print(transactions)
```

Imports our function

Defines an empty list containing strings

Adds some transactions

Prints the results

The output from listing 1.6 is `['deposit $300', 'deposit $400']`, so we only have two items, not three as we would have expected the default to be. So, either the logic to default the parameter to `3` is not working, or the logic to pop transactions is not working. Another possibility is that something is happening that we are not anticipating. Keep that in mind because, on occasion, we will be staring at a part of the code we think is the problem, but we can't figure it out. We may spend hours struggling only to find that both the problem and answer lie in another part of the code. We have some unit tests that go along with this problem as well, which are shown in the next listing.

Listing 1.7 Defining a unit test for our function

```
def test_first_transaction_is_dropped():
    expected_value = 3
    transactions: list[str] = []

    transactions = last_transactions(transactions, "deposit $100», 3)
    transactions = last_transactions(transactions, "deposit $200", 3)
    transactions = last_transactions(transactions, "deposit $300", 3)
    transactions = last_transactions(transactions, "deposit $400", 3)

    assert (
        len(transactions) == expected_value
    ), f"Expected {expected_value}, but got
➥ {len(transactions)} transactions"

    assert (
➥     transactions[0] == "deposit $200"
    ), f"Expected {expected_value}, but got
➥ {len(transactions)} transactions"
```

Defines our expected number of transactions

An assert test to ensure we have the expected number of transactions

We expect the first value passed should have been dropped, and so we have "deposit $200".

So, if we wanted to take a break and look at fixing this for ourselves before moving on, we certainly could do that now. Then we can come back and see what ChatGPT gives us.

Welcome back! Hopefully, we were able to give this a try to play around with the code a little bit, but let us jump right back in and enter this into ChatGPT. As we saw, the code from listing 1.5 has a logic flaw: we expect to see three transactions listed but only see two. We simply ask ChatGPT, "Why doesn't this code work?" which we did on purpose just to disregard some of the prompt engineering and problem formulation that we discussed earlier. We wanted to see what type of response we would recieve, and it was a great response as ChatGPT identified two problems that it did not like with the code:

The code you've provided is aimed at adding a new transaction to the user's transaction list while ensuring that the length of the list doesn't exceed the given "max." If adding a new transaction would exceed the "max," the oldest transaction is removed.

However, there are a couple of problems:

1. The name "max" is a built-in Python function that returns the largest of the input values. It would be better to avoid using it as a variable name. Let's change it to something like "max_length."
2. The condition for popping a transaction is "if len(user_transactions) + 1 >= max:". This means if you have "max_length = 3" and the current length of "user_transactions" is 2, it will remove the first transaction before adding the new one, thus always keeping the length as 2. This is incorrect. The correct condition should be "if len(user_transactions) + 1 > max_length:".

As we can see, a couple of problems were pointed out: first, we used the name "max" in our code. Our IDE flagged this as well since `max` is a built-in Python function. As we saw earlier with this name shadowing, the code will still work but can become a bit more convoluted when trying to follow it. Second, the other suggestion was to update the if statement in the code to be `if len(user_transactions) + 1 > max_length:`. With either of these solutions in place, we can rerun the code or rerun the tests to confirm that the code is working better.

One thing we always find interesting is the number of solutions to any problem; sometimes we solve something only to go back later and say, "What were we thinking!?" We thought that ChatGPT would have just dropped the `+ 1` and kept the existing logic. If we found that puzzling, then we might ask ChatGPT about it. We followed up with "What if we were to change the line to `if len(user_transactions) >= max_length:`?" and ChatGPT gave us some additional information but also confirmed that "this change will indeed make the code work correctly for the intended purpose!" We have the code and `pytests` confirming both ways.

We cannot really provide insight into why ChatGPT chose one way over another, but we take this as confirmation that, again, generative AI is a tool that can be used but should not necessarily be taken as the end all be all. We tried to follow up with "Is there a performance benefit to one way versus the other?" and ChatGPT gave a reasonable

answer. One thing we found interesting was that it started putting `if len(user_transactions) >= max_length:` as its suggested change and favored it because it was intuitive and provided better readability. Either way, we hope that you will also see the usefulness in being able to get some feedback from generative AI tools to help us determine code flaws and potential enhancements.

Sometimes it is not necessarily a code problem that we have but more of a "I am sure that you need to use an equal sign to assign a variable, but it is not working." Generative AI can be of huge help when you know what you want to do but cannot quite remember how to do it in a particular language or environment. We look at how generative AI can help in the next section.

1.5.4 Help! I forgot the syntax!

Another benefit of generative AI is just getting started with the syntax. Sometimes, as a full stack developer, constantly switching from one language to another can make us dizzy. We may find ourselves knowing what we want to accomplish but not being quite sure how to get there. As we saw with the first example, we can have our generative AI tools get started for us as long as we know how we want to go about the process. For instance, in our first example, we were able to define a function, pass parameters, and use type hints. If we are not aware of these constructs or are unwilling to learn about them, then it is unlikely that generative AI will be of much use to us.

Let us try another example from full stack development. Previously, we were working with Python, but let's switch gears and jump to the UI with HTML5 and D3.js. We like to use VS Code with the Live Server plugin when working with HTML and scripting. Everyone has varying levels of experience with HTML5 and D3.js, from never having heard of them to an expert. However, even if our experience is having seen a bar graph or used one on a website, we can work our way through this without too many problems. We can ask ChatGPT to "create a simple d3.js bar chart in an html5 page" and end up with something respectable but, by no means, award winning (figure 1.8).

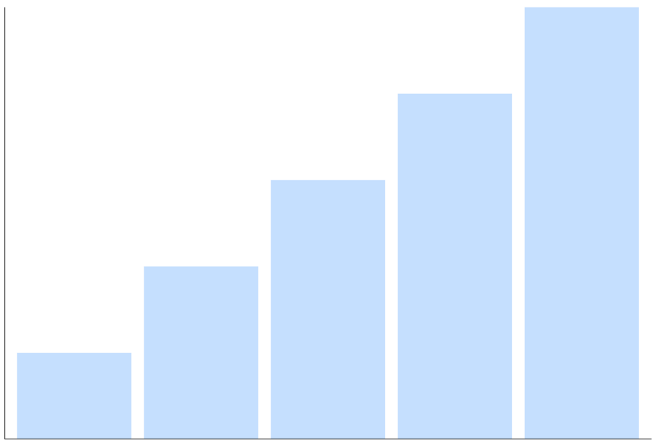


Figure 1.8 Simple bar chart

The code shows the data being used:

```
const data = [
    { name: 'A', value: 10 },
    { name: 'B', value: 20 },
    { name: 'C', value: 30 },
    { name: 'D', value: 40 },
    { name: 'E', value: 50 },
];
```

That graph looks okay, but we want to add some labels to the axis. That should be something we have come across when seeing visualizations before and should not be that far of a jump for us to consider adding to the visualization. We can simply enter, "Can you update the code to add axis labels?" at the prompt, and we will see that labels were successfully added. Even though we may not be sure how to write the code, we know what we want the code to do, and that is enough to get us going in the right direction for this example. See the output with axis labels in figure 1.9.

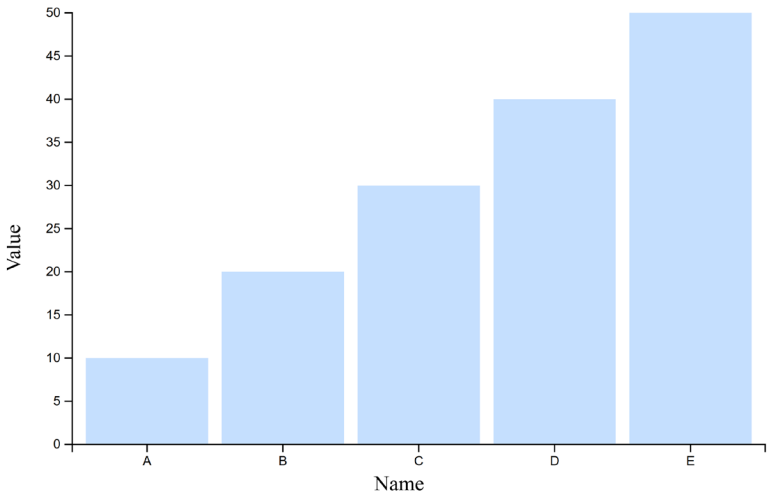


Figure 1.9 Bar chart with axis and labels

This is looking pretty good now. We may not win any contests, but it certainly helps get the point across with our visualization. Now, we can see about adding a little interactivity when we hover the mouse over a bar. We enter "Can the code be updated to change the bar color and display the value for the bar when I hover with the mouse?" on the prompt, and we get an interactive graph. The bar changes color from blue to orange when we hover along with the value being displayed near the mouse. Playing around with this, we can see it works reasonably well, but one thing we are not happy with is

that the tooltip with the value appears where the mouse enters the bar but doesn't move with the mouse as it moves within the bar itself. We could fix this by updating the code to update the position when the mouse moves, or maybe we should just display the value above the bar when we hover, or perhaps just display the value above the bar all the time. Feel free to play around with some of these or implement other approaches!

We asked for the tooltip to be displayed above the bar. Figure 10.1 shows the final shot of what we ended up with.

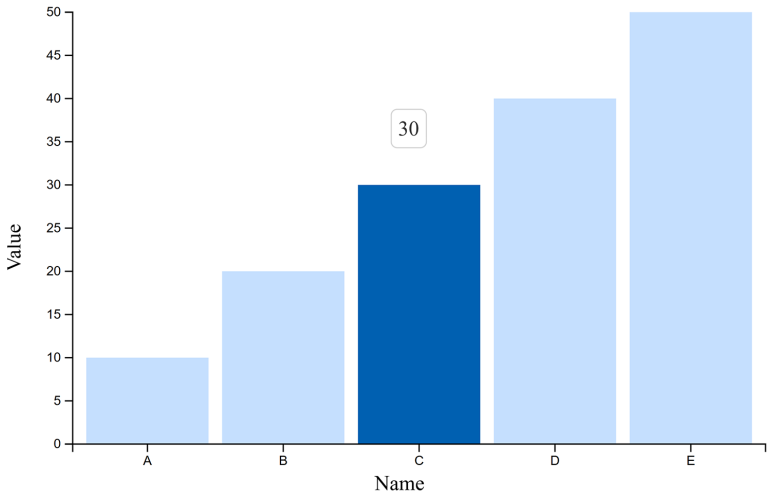


Figure 1.10 Highlighting and hovering in our bar chart

The label does not appear quite centered, and when it is displayed over the largest bar, it goes off-screen. We could probably coerce ChatGPT into getting it just the way we want, or we could dive into the code and try to work through it. Either way, we should have been able to come up with a decent bar graph even if we have not used D3.js or are not familiar with HTML. These code snippets can also act as training tools for us to play around with the code to use and learn from. Maybe we aren't sure how to use the `append` or `on` methods from d3.js, reviewing these samples or the generated code is a great way to see them in action.

Verification

Since we just worked through some examples that we may not have been very familiar with, it is probably a good time to go over verification. We should never have tools such as ChatGPT write us a bunch of code that we could never understand. It can be beneficial to see how things are done, and we can even use it to generate code

(continued)

for ourselves (see the following sections for some additional insights on security/privacy/ownership concerns). But remember that sticking that code in and checking it means that we will own it. There have been many times where we have been asked about a code change that we made a decade ago because we were the last person to change that module.

Here is a Perl one-liner generated from ChatGPT that validates an email address is formatted correctly. Feel free to insert it into your existing software solution—just be ready to own it in the middle of the night when the support staff calls you to say that the customer email import process is broken!

```
perl -e 'my $email = "user@example.com"; if ($email =~ /^[a-zA-Z0-9._%+-
]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/) { print "$email is a valid email
address.\n"; } else { print "$email is an invalid email address.\n"; }'
```

If you were unfamiliar with regular expressions and Perl, how likely would you be to put that in your code?

Another feature of tools such as ChatGPT is the ability to ask for references. Perhaps, after seeing the regular expression to validate an email, you decide that you may want to validate the results by doing a bit more research. By asking ChatGPT to provide sources for the code, it will refer you to the Perl documentation (`perldoc`). It will also refer to general regular expression knowledge, as well as the official standard for email addresses (RFC-5322). This can give us a good starting point when we want to learn a bit more about code we do not understand.

We see how powerful and helpful generative AI is, but in the next section we'll take a look at a potential pitfall when using it.

1.5.5 Did I just give away my data?

The next example of the benefits of generative AI is the ability to have it take input and process it for us. For example, let us say that a banker provides us with a report file, and they want some reporting done with it. They provide us with a comma-separated value (CSV) file and want us to total it up. Why they cannot do that in Excel is beyond us, but we figure it may be a good chance for us to get some additional Python practice and maybe try generative AI. We are provided some data formatted as

```
"ssn", "account", "balance"
"000-65-4320", 1, "$1.00"
"000-65-4321", 1, "$2.00"
```

When we paste that data into the prompt, ChatGPT responds with "It looks like you've provided a list of Social Security Numbers (SSNs) paired with account numbers and corresponding balances. How can I assist you with this data?" which is a very nice

response. Compare that response to one we may receive from your co-worker if you just threw a similar printout in front of them.

So, at the prompt, we enter "Given this sample file, write a python program to print a report with totals" and paste in our data below the prompt. It prints the program from the following listing.

Listing 1.8 Sample report program

```
data = """
"ssn", "account", "balance"
"000-65-4320", 1, "$1.00"
"000-65-4321", 1, "$2.00"
"000-65-4322", 1, "$3.00"
"000-65-4323", 1, "$4.00"
"000-65-4324", 1, "$5.00"
"000-65-4325", 1, "$6.00"
"000-65-4326", 1, "$7.00"
"000-65-4327", 1, "$8.00"
"000-65-4328", 1, "$9.00"
"000-65-4329", 1, "$10.00"
"""

lines = data.strip().split("\n")
total = 0.0

# Skip the header row
for line in lines[1:]:
    balance_str = line.split(",")[2].strip().replace("$", "")
    total += float(balance_str)

print(f"Total Balance: ${total:.2f}")
```

The code looks reasonable, so we go ahead and run it. However, the code runs into a value error: `ValueError: could not convert string to float: '"1.00"'`. So, the code produced from ChatGPT contains an error. We can try to fix this on our own, or we can let ChatGPT know that it has a problem. Feel free to review and troubleshoot the error to see whether it can be fixed before moving on!

The problem turned out to be the quotes in the `balance_str`. To correct it, we can add an additional `replace` to handle the double quotes, so the line becomes `balance_str = line.split(",")[2].strip().replace('$', '').replace('"', '')`.

While this is potentially a very helpful way to use generative AI, there are some potential drawbacks. One is that we have just entered nonpublic information (NPI) into ChatGPT. If the tool we are using ingests the data for training, then potentially, we have exposed customer information to an insecure source, as it may show up in responses to other users. We must be very careful about the data that we enter into any tool, not just when using generative AI, as we want to be sure that the information is stored securely. In our case, any SSN starting with `000` is invalid, so we are not using real data.

Ownership

We covered the potential concerns about entering secured data into generative AI, and we should also be aware of entering proprietary information into it as well. If our proprietary code is being used to train the system, it may not be proprietary for very long. Always look at the Terms of Service (ToS) and End-User License Agreement (EULA).

Another concern may be that of ownership of responses: if we ask the model to generate some code that goes into a product which we are selling, then who owns that code? Do we have to pay royalties to them? These types of questions/concerns are what we should keep in mind as we make the transition of student or developing for personal use versus developing in a corporate environment. Corporations can be very prickly about releasing sensitive data and rightfully so. Their business may depend on trade secrets, and there could be substantial fines for mishandling sensitive data.

Concerns about ownership lead us to similar concerns about privacy and security, which we discuss in the next section.

1.5.6 Privacy and security

Other areas of generative AI that we need to be aware of—which will be a pervasive topic throughout our careers—are the areas of privacy and security. It may be possible that feeding our code into a generative AI tool is helping to train it, and our proprietary code could end up getting output as an answer to someone else's prompt. Tools such as GitHub Copilot specifically state that they do not store or share our code. In that same aspect, we will need to be careful about sensitive information such as nonpublic information (NPI) or personally identifiable information (PII) that we input into the model as well.

Maybe we want the generative AI tool to summarize the quarterly earnings report for each business unit, so we can create a compelling infographic and think that generative AI could make that helpful. Well now, we have just potentially shared corporate/trade secrets with the model! So, some care and thought should be given to what we enter the system. If using generative AI to troubleshoot problems as we did in the previous section, it may be beneficial to create a smaller code snippet that is more generalized but still suffers the same problem we are troubleshooting. Who knows, we may just figure out the problem in the process!

The Open Worldwide Application Security Project (OWASP) has specific guidance on the adoption of generative AI at https://genai.owasp.org/. It is interesting to review the OWASP Top 10 for LLMs and generative AI and see what type of exploits we may encounter. Knowing about the potential dangers will hopefully help us avoid them.

1.5.7 Staying up to date

One last concern might be how recent the data in the model is. If the model is based on older data, we may not get the results we intend. In the majority of cases, we have

seen when asking tools such as ChatGPT for the latest version of Java or NPM, it is aware enough to qualify its response. In other cases, our question may indicate that it is a potential version problem. For instance, when asking ChatGPT "How do I connect to PostgreSQL in Python?" it provides a reasonable example of using `psycopg2`. However, there is a newer rewrite of `psycopg2` called `psycopg3`. If we try to refine our example and ask ChatGPT to use `psycopg3` instead of `psycopg2`, it again says that its data may be out of date:

It seems like there might be a typo in your request. As of my last knowledge update in September 2021, there is no library called `psycopg3`.

Of course, if we were not aware of `psycopg2` or given instructions to use `psycopg3`, we may be thrown off course a little bit but not to worry. We might think that if there was confusion about whether our assignment was `psycopg2` or `psycopg3`, we could easily ask our co-workers. However, if we are new to the job and thrown into the deep end of the pool, we may be reluctant to ask our new mentor (who is undoubtedly very busy) to clarify such a simple oversight.

It is interesting to see that we can ask ChatGPT to treat `psycopg3` as a drop-in replacement for `psycopg2` by entering the following at the prompt: "`psycopg3` is a rewrite of `psycopg2`. Can you simply replace `psycopg2` with `psycopg3` in the given example?"

Then, ChatGPT will again generate a reasonable response:

I apologize for the confusion earlier. Since I don't have information about `psycopg3` as of my last update in September 2021, I'll assume its usage is similar to `psycopg2`. If `psycopg3` is indeed a rewrite of `psycopg2`, the usage should be quite similar.

This is yet another aspect of verification that we must be aware of—if the tools are not up to date and giving us suggestions, there could be vulnerabilities to the particular version or various improvements that we might be missing out on. Again, generative AI is giving us a good starting point, but it is still up to us to do some due diligence to ensure everything is on the up and up.

Summary

- FinTech encompasses a wide range of financial technologies, historically from cash registers to modern-day applications, including crowdfunding, gig service apps, P2P payments, and cryptocurrencies.
- The book targets full stack developers in FinTech, covering end-to-end application development, including UI, backend processing, and troubleshooting through real-world examples.
- The SAFe Agile framework is widely used in FinTech companies and is significant in program increment planning in managing software development projects. Legacy systems are being modernized using new tools and frameworks, such as PostgreSQL, Next.js, Docker, and Python. These changes have a significant effect

on both users and businesses. Generative AI is seen as an evolution of the rubber duck debugging method, providing active feedback and ideas.

- Generative AI uses LLMs and is integrated into various industries for increased efficiency.
- Generative AI assists in coding, debugging, syntax recall, and generating code snippets across programming languages.
- Readers should be cautious about relying on AI for up-to-date information due to potentially outdated training data.

2 Parsing payments

This chapter covers

- Working with generative AI to parse an ACH file
- Unit testing in Python
- Agile concepts

This chapter explains the fundamentals of files formatted by an Automated Clearing House (ACH). You will work on parsing an initial sample file to better understand some of the record layouts used, as well as the software development practice of unit testing. You will also expand your knowledge of generative AI and see how to apply it to a real-world problem.

2.1 Modernizing our legacy software

Now that we have gone through PI planning and have been assigned our project, we will begin working on our first spike. In SAFe, spikes are a type of story, often referred to as an *enabler story*. A variety of enabler story categories may be used (e.g., exploration, architectural, infrastructure, and compliance). In our case, our story may be classified as an exploration type or a research spike. However, the details are not as important as our activities because the particulars may vary between Agile

frameworks. Essentially, this story aims to provide the team with some experience working in Python and learning about the ACH file layout so that we can understand prospective solutions better.

The stories assigned to the team in this sprint can shed light on the business needs driving the modernization effort. The current ACH system runs on the mainframe system, and it is highly coupled with the existing architecture. This coupling prevents the system from being maintained easily. The business has been keeping metrics on various key performance indicators (KPIs):

- *Release cadence*—Shows how often the development teams release new features and updates
- *Defect backlog*—Shows the number of defects identified during development and release
- *Escaped defects*—Shows the number of defects identified after the software has been released
- *Customer satisfaction*—Indicates how happy the customer is with our product/service

These metrics have shown that there has been a steady increase in the time it takes to make regulatory enhancements and general bug fixes as the system's code base has grown in both size and complexity.

Currently, we do not have any knowledge of ACH files, and understanding and navigating the COBOL code will take time; moreover, we may need to request access. The team will have to gain insight into the COBOL code so that they can better assist in converting functionality from one area to another. In the interim, the team is supposed to tackle the project from the ground up. The team also has first access to generative AI tools to help evaluate their use since the company is looking into using Gen AI to improve productivity. Our manager believes that the team is in a unique position to evaluate the tools to help us with this effort.

The company gave us access to both premium ChatGPT and GitHub Copilot membership. They have also asked that we test out the free version of ChatGPT because, if the free version is comparable to the premium one, they would much rather save some money. In addition, our manager has reminded us not to paste any proprietary or confidential information into ChatGPT because that would be a potential violation of the nondisclosure agreement (NDA) we have in place. This means we cannot paste the COBOL code and convert it to Python or ask ChatGPT to interpret existing code. Instead, we should focus on isolating, sanitizing, and generalizing code samples before plugging them into ChatGPT.

Our plan is to do a little research on what exactly an ACH is and what type of format we'll be working with. We are familiar with JSON as we have done a fair amount of web-based work before. Either way, we are excited to get moving on to this project: getting into some code and working on a project is why we wanted this job in the first place, so let's begin!

2.2 Understanding the ACH

The ACH network allows banks and credit unions to transfer money without establishing a separate relationship. Given that there are over 10,000 banks and credit unions in the United States, that is quite an undertaking! So, how does it work?

A common type of ACH transaction is known as Prearranged Payment and Deposit (PPD). It is often used for direct deposit of paychecks (and recurring bill payments, gym memberships, and social security payments, to name a few). Let us say our employer has an account at bank X. Every two weeks or twice a month, they need to pay us, but we use a different bank, bank Y. To get the money to you, a few days before payday, our employer or their payroll processor will create a file containing the information needed to execute the payment. It includes the amount of our take-home pay, our account number, and a *routing transit number*. A routing transit number is a unique number that identifies a bank—in this case, the bank of the end recipient of the transaction: bank Y. The payroll processor will transmit the file containing the employee wages info to bank X. Bank X accepts the file and combines it with other files they receive. Next, it sends one file to either the Federal Reserve or a private company called The Clearing House. These two entities are known as *ACH operators* or *clearing houses*. ACH operators accept files from thousands of banks, sort the transactions based on their routing transit numbers, and create new files for each bank receiving transactions. These files are then transmitted to the banks, which then receive and post the transactions.

ACH files use a format defined by Nacha, the organization that sets standards for the ACH network. Figure 2.1 shows the ACH processing flow.

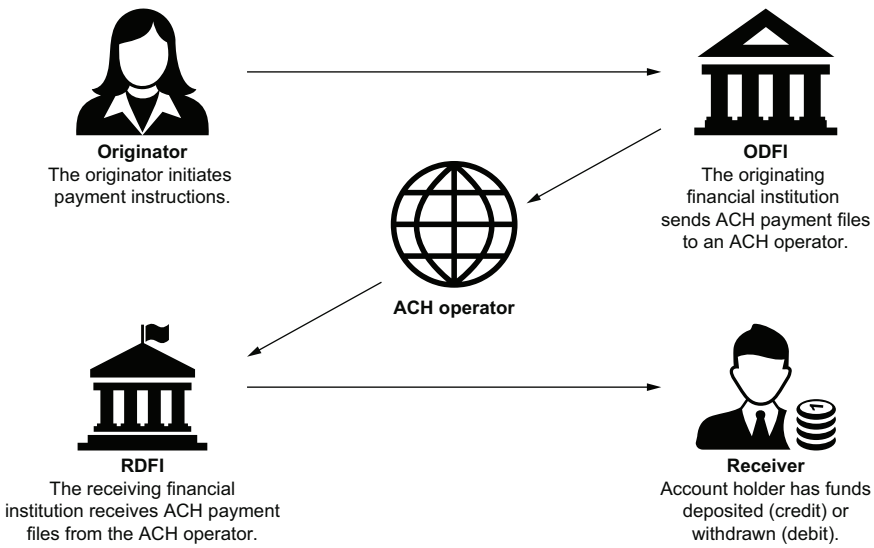


Figure 2.1 ACH processing flow

2.3 Parsing an ACH file

So, we have been assigned a story to parse an ACH file and store it in the database. The story has subtasks for parsing each of the record formats found in an ACH file, as well as storing the results in a database.

The ACH file standard was created in the 1970s and has undergone various updates and expansions since then. This is great because it means that plenty of information is available on the standard. We start by doing some research on ACH, finding that an ACH file is a fixed-width ASCII file with each line being 94 characters long. These lines are known as records, and each record consists of fields that are at fixed positions. A fixed-width file with records at fixed positions should mean parsing is a relatively straightforward task.

Digging a little bit further, we see that there are six types of ACH records that may be present in a file. It seems there are a few types of header records, and each has a trailer record known as a control record that wraps up the data. We also find an overview of the record types and file structure (see figure 2.2). Further details on the records and their corresponding fields (such as the position, whether it is required, etc.) are available at https://achdevguide.nacha.org/ach-file-details.

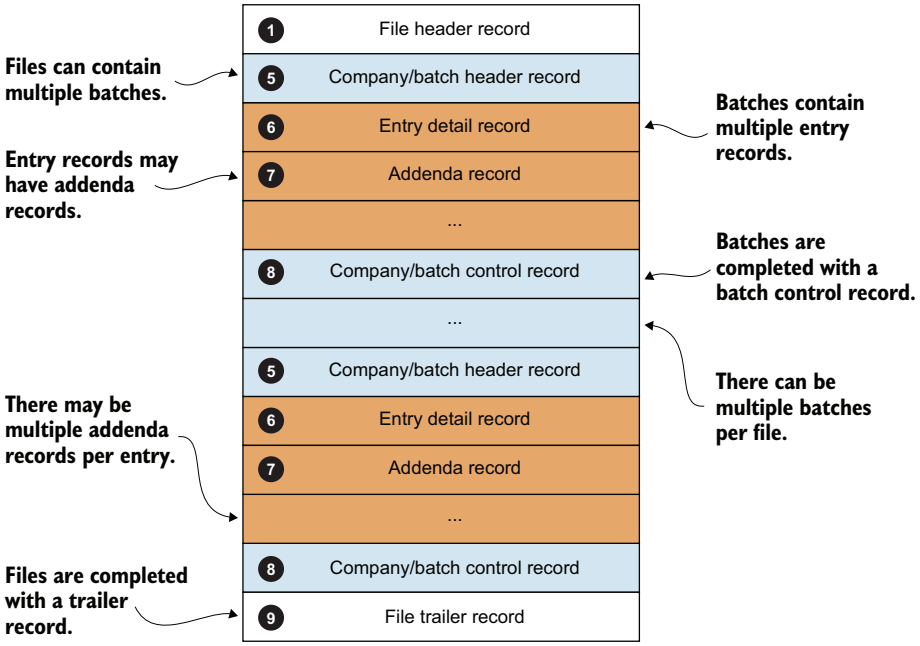


Figure 2.2 ACH file layout

Figure 2.2 gives us an idea of the file structure; however, let's also look at a sample file that we will be working with so we can get a better idea of what a file may look like. Figure 2.3 may look a bit daunting, but once we break down the records into their fields, it is relatively straightforward.

```
101 990000013 691000134040220083OA094101DEST NAME              ORIGIN NAME            XXXXXXXX
5200Company name    DiscretionaryData   1234567890ARCComp desc 021623230216047106100001000001
627267084131123              000000010101223344       xxxxxxxxxxxxxxxxxxxxxxLC0061000010001001
6272670841311234             0000000102027766554433221An ARC                LC0061000010001002
6262670841310000000000001234500000010111223344       xxxxxxxxxxxxxxxxxxxxxxLC1061000010001003
799R01021302641003335      06310027RETURN OF ARC TRANSACTION                   061000010001003
626267084131123456           000000010288776655443322 1xxxx                LC1061000010001004
799R01021302641003335      06310027RETURN OF ARC TRANSACTION                   061000010001004
820000000600330033000000000004060000000000001234567890                        061000010000001
5200CCD coname                          1234567890CCDDesc      021623230217048106100001000002
62226708413234               0000000201222222222222222TESTS CCD1 TRANSACTIONS 1061000010002001
705Addenda stuff                                                                00010002001
820000000600247524750000000004040000000000201CCDCompany                       061000010000002
5200CTXCompname                         1234567890CTXCompDesc  021623230216047106100001000003
6272670841345                0000000301               0003xxxxxxxxxxxxxxx  LC1061000010003001
705Test 1                                                                       00010003001
705Test 2                                                                       00020003001
705Test 3                                                                       00030003001
6222670841313456             0000000302               0002xxxx               1061000010003002
705Test addenda 1                                                               00010003002
705Test addenda 2                                                               00020003002
820000000800165016500000000003010000000003021234567890                        061000010000003
5200Comp--name052308Discretionary---Data1234567890MTEComp--desc021623230216047106100001000005
6272670841315678             0000000501INDIVIDUAL---IDINDIVIDUAL--------NAMEdd1061000010005001
702TxnDescIDCTermIDSerial0216235959Terminal Location          Term City      FL061000010005001
62226708413156789            0000000502XXXXXXXXXXXXXXXYYYYYYYYYYYYYYYYYYYYYDD1061000010005002
702MTETXN    TERMID2000020216235959Galleria                   Miami          FL061000010005002
820000000400165016500000000004010000000004021234567890                        061000010000005
90000100000100000007401980198000000000078250001146114 80
```

Figure 2.3 Example ACH file

Based on the structure and file sample, we know that we will have to parse the following record types:

- *Type 1*—A file header record with a single record per file
- *Type 5*—A batch header record with multiple batches per file
- *Type 6*—Entry records, with multiple entries per batch
- *Type 7*—Addenda records with zero or multiple records per entry record
- *Type 8*—A batch control record that encloses the entry and addenda records
- *Type 9*—A file trailer record that encloses all the batch records

While we have not gotten access to the legacy COBOL code, a team member found a confluence (a wiki provided by Atlassian) site with the flow chart that provides some insight into the processing of an ACH file (figure 2.4).

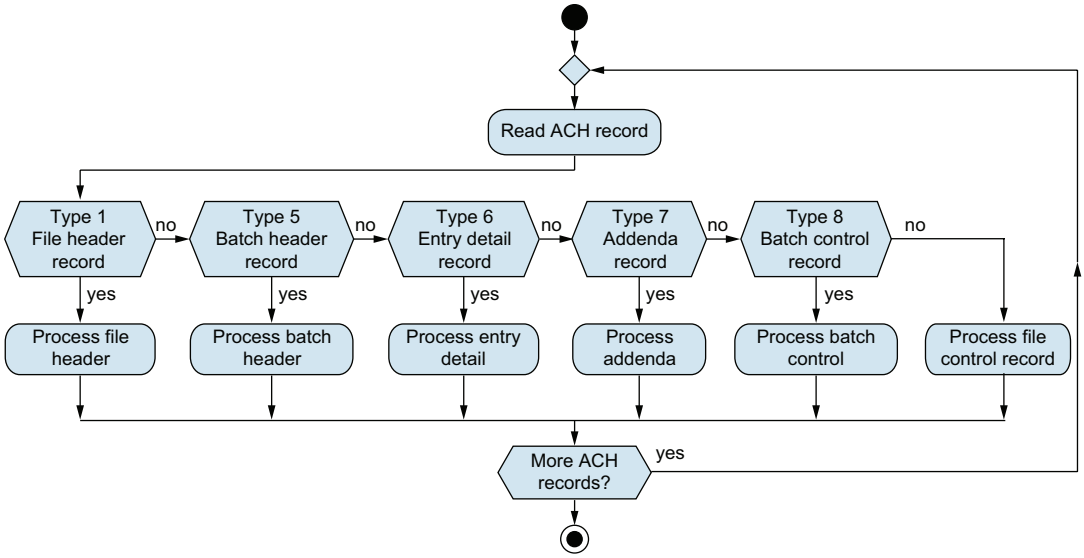


Figure 2.4 ACH processing flow

Your first task is to check your favorite search engine to see what is available. You may find some existing packages (`pyNacha`, `py-nacha`, and `pynacha`). Those appear to be various projects exploring how to create and parse ACH files. Further digging also shows some of these projects came from `carta-ach`, which again came from `python-ach`. These projects have been forked many times, but most appear to have not been updated for some time. As they are not actively maintained, it may not be wise to base your project on them. However, they appear to have a permissive MIT license (more on licensing concerns later), so we could possibly fork a project to get started. We can also lean on some of our generative AI tools for some help as well.

2.3.1 Asking ChatGPT to parse an ACH file

We can start by asking ChatGPT the question, "Can you write a Python program to parse an ACH file?" When we try this in ChatGPT 3.5, it gives us the basics of an `ACH` class that can parse a file, as shown in the next listing.

Listing 2.1 ACH parser from ChatGPT 3.5

```
class ACHParser:
    def __init__(self, ach_file_path):
        self.ach_file_path = ach_file_path
        self.records = []

    def parse(self):
```

Defines a class named ACHParser

The __init__ function is used as a constructor for our class.

```
        with open(self.ach_file_path, 'r') as file:
            lines = file.readlines()

        record_type = None
        current_record = {}

        for line in lines:
            record_code = line[0]
            if record_code == '1':
                record_type = 'header'
            elif record_code == '6':
                record_type = 'batch'
            elif record_code == '7':
                record_type = 'entry'
            elif record_code == '9':
                record_type = 'footer'

            if record_type:
                current_record[record_type] = line.strip()
                if record_type == 'footer':
                    self.records.append(current_record)
                    current_record = {}
                    record_type = None

    def get_records(self):
        return self.records


if __name__ == "__main__":
    ach_file_path = "example.ach"
    parser = ACHParser(ach_file_path)
    parser.parse()

    records = parser.get_records()
    for record in records:
        print(record)
```

Reads the entire file

Determines the first character and stores it in record_code

Performs processing based on the record_code; notice there is no record_code 5 or 8.

The program does not look too bad; however, despite our current limited knowledge of ACH, we can see that it has some problems:

- Missing a record type code of `5`.
- Missing a record type code of `8`.
- The `record_type` lables are incorrect, as `6` is listed as a `batch` instead of an `entry`, and `7` is listed as `entry` instead of `addenda`.

We try switching over to the updated ChatGPT 4 and ask it the same question. We are presented with the code shown in the following listing. Again, keep in mind that the nondeterministic nature of LLMs may lead to different results.

Listing 2.2 ACH parser from ChatGPT 4

```
def parse_ach(filename):
    records = []
```

An empty array to store ACH records into

```
with open(filename, 'r') as file:
    lines = file.readlines()

    for line in lines:
        record_type = line[0]

        if record_type == '1':  # File Header Record
            record = {
                'record_type': 'File Header',
                'priority_code': line[1:3],
                'immediate_destination': line[3:13],
                'immediate_origin': line[13:23],
                # ... additional fields
            }

        elif record_type == '5':  # Batch Header Record
            record = {
                'record_type': 'Batch Header',
                'service_class_code': line[1:4],
                'company_name': line[4:20],
                # ... additional fields
            }

        elif record_type == '6':  # Entry Detail Record
            record = {
                'record_type': 'Entry Detail',
                'transaction_code': line[1:3],
                'receiving_dfi_id': line[3:11],
                'account_number': line[12:29],
                'amount': line[29:39],
                # ... additional fields
            }

        elif record_type == '7':  # Addenda Record
            record = {
                'record_type': 'Addenda',
                # ... additional fields based on the
                # specific addenda type
            }

        elif record_type == '8':  # Batch Control Record
            record = {
                'record_type': 'Batch Control',
                # ... additional fields
            }

        elif record_type == '9':  # File Control Record
            record = {
                'record_type': 'File Control',
                # ... additional fields
            }

        else:
            print(f"Unknown record type: {record_type} in line: {line}")
            continue
```

Each record_type is stored as a dictionary; some of the fields were provided by ChatGPT.

ChatGPT 4 did generate a record type 8 this time.

```
            records.append(record)

    return records

# Test the function with an ACH file
parsed_records = parse_ach('path_to_ach_file.txt')
for record in parsed_records:
    print(record)
```

ACH records are added to our array.

The records are returned.

The returned records are printed out using a for-each loop.

This program seems to be a little more built out. We see the batch control record (record type 8) this time, and some of the lines have a sample parsing done as well. This seems to be a good start; however, we want to keep the program testable. If we separate the parsing logic into separate functions, we can pass both well-formatted records and invalid data to the function to test it. Let's see if we can get ChatGPT to do that somewhat tedious process.

We simply ask ChatGPT:

CK Can you update the above program so that the parsing of each record type is a separate function?

The program is updated to break each of the record types into its own function. The relevant changes are shown in the following listing. Notice that we have been provided with parser functions (`parse_file_header`, `parse_batch_header`, etc.) and that these have been stubbed out by ChatGPT (meaning we will have to provide implementation details to parse the actual data).

Listing 2.3 Parsing with separate functions

```
def parse_ach(filename):
    records = []

    parser_functions = {
        '1': parse_file_header,
        '5': parse_batch_header,
        '6': parse_entry_detail,
        '7': parse_addenda,
        '8': parse_batch_control,
        '9': parse_file_control
    }

    with open(filename, 'r') as file:
        lines = file.readlines()

        for line in lines:
            record_type = line[0]
            parser = parser_functions.get(record_type)

            if parser:
                records.append(parser(line))
            else:
```

Defines needed parsing functions in a dictionary

Retrieves the appropriate function based on the first character we stored in record_type

Calls the parse function with the appropriate data

```
                print(f"Unknown record type:
    ➥{record_type} in line: {line}")
                continue

    return records
…
def parse_file_control(line):
    return {
        'record_type': 'File Control',
        # ... additional fields
    }
…
```

Calls the parse function with the appropriate data

One of the sample stubs for the parser_functions

In the tests/data folder, there is also a sample.ach file (as we saw in figure 2.2) that we can plug into these sample programs to examine how they process the file and play around with the results. Now it would be a good time to take a break and run the sample ACH file through these programs to see how they work. This is helpful if we do not quite understand ACH file processing yet or if we are unfamiliar with some of the constructs used in the Python program.

It is a powerful tool to help us get started. Many developers have templates, shells, or skeleton programs they use to get started with particular tasks. Or they may choose a particular program to copy and paste. Of course, the problem with copying/pasting is that you may inevitably miss something and introduce a compile-time problem or bug that you then need to troubleshoot. The same thing can happen when using a template to start your program.

These templates or code produced from generative AI can be useful, especially if we are willing to look at the code and learn from it. Maybe we will learn a new way to do something or maybe we will find a new technique to apply. Still, maybe we find a bug in the produced code. The point is that it can be a helpful tool as long as we use it and do not take the output for granted; the code produced must be treated as our own code, meaning it should be tested and verified through our unit testing and quality assurance (QA) process.

2.3.2 Parsing an ACH file with Copilot

After seeing what ChatGPT has to offer us, let's use Copilot to see whether it can get us closer. We first enter the following into an empty Python script:

CK `# Function to parse an ACH file`

It does an okay job, but not quite what we wanted, so we try again with an updated prompt:

CK
```
# Write a class to parse an ACH file for each NACHA record type and
return a list of transactions.
```

As we are typing, the "for each NACHA record type and return a list of transactions" is automatically populated. We then enter an expanded comment:

CK
```
# Write a class to parse an ACH file with separate functions for
each NACHA record type and return a list of transactions.
```

This prompt gives us the following output, with the full listing available in copilot_ach_parse_with_functions.py.

Listing 2.4 Copilot ACH parser

```
    def parse_ach_file(self):
        with open(self.ach_file, 'r') as f:
            for line in f:
                if line.startswith('1'): self.transactions
                        .append(
➥ self.parse_ach_batch_header(line))
                elif line.startswith('5'):
                    self.transactions
                      ➥ .append(
self.parse_ach_batch_control(line))
                elif line.startswith('6'):
                    self.transactions
                       .append(self.parse_ach_entry_detail(line))
                elif line.startswith('7'):
                    self.transactions
                       .append(self.parse_ach_addenda_record(line))
                elif line.startswith('8'):
                    self.transactions
                      ➥ .append(
self.parse_ach_file_control(line))
                else:
                    raise ValueError(
➥'Invalid ACH record type')
```

Incorrectly calls a parse_ach_batch_ header routine when this is the file header record

The batch header record is being parsed incorrectly as a batch control record.

The batch control record is listed incorrectly as file control.

Missing an elif statement for file control records (type 9)

A ValueError is raised when we do not have a valid ACH record type.

We can see that it created a class and functions to parse individual record types. However, we also see multiple problems:

- Record type 1 is calling a parse routine for `parse_ach_batch_header`, whereas it should be parsed as a file header.
- Record type 5 is calling a parse routine for `parse_ach_batch_control`, whereas it should have been parsed as a batch header.
- Record type 8 is calling a parse routine for `parse_ach_file_control`, whereas it should have been parsed as a batch control.
- The file trailer record (record type 9) is missing.

So, while this code also gives us another viable template, it does not necessarily produce something right out of the box to use. One helpful thing that it does provide is code to raise a `ValueError` with `invalid ACH record type`, so we would certainly run into an error right away when we try to load a test ACH file as the missing type 9 should cause an error on any well-formatted ACH file.

2.3.3 Generative AI: Trust but verify

Certainly, both ChatGPT and Copilot are able to get us started with meaningful templates we can expand on later. However, they both have some problems generating the code, which highlights the importance of verification mentioned earlier. We must understand the produced code and not just take for granted that it is correct. This is especially important when we expect code to apply business rules and logic to the functions we want to generate. For instance, when using generative AI to assist in creating purely functional code—such as a button to submit a form or read a file—it will be immediately obvious if there is a syntax error or if it does not perform according to specifications, as it will not compile. However, if it is missing a record type or has some other problem, it could introduce bugs that are harder to find, especially if we don't understand those business rules. We make this point multiple times only to drive home the importance of generative AI being a powerful tool but still only a tool. Think of the process of building a chair. We can build a chair with just a hand saw or we can have a shop full of the latest and greatest tools, but if we do not know how to use them, we will likely end up sitting on the floor.

So, how do we go about verifying the code that generative AI produces for us? By using extreme programming (XP) development practices such as test-driven development (TDD). We previously mentioned the need for reliable high-quality software in FinTech.

TDD boils down to the concept of writing tests before code by letting tests drive our development. One of the benefits is that we have highly testable code. We will use the TDD principles throughout this book and show various ways to automate our tests. Regardless of whether we decide to adopt a test-driven approach to our development practice, there are real benefits to thinking about how our code is tested. Questions such as the following are always considered when using a TDD approach:

- Is this code that needs to be tested, or will we need to regression-test this code?
- How do I test this code?
- How will others understand this code?

The answers to these questions are usually found through unit tests, and when thinking about unit tests, we should be thinking about TDD.

We can also make use of our Integrated Development Environment (IDE) and any available plugins it may come with. These tools may include customizable formatting options and default highlighting of syntax problems. As soon as our development team grows beyond just one person, there are real benefits to enforcing some standards in our code.

Our company will likely provide us with the tools to use. These may help identify problems with generated code right away. Tools such as Snyk which can scan our code for vulnerabilities are discussed in chapter 3. At the very least, we want to ensure that the code meets our company's policies and standards.

Reduce cognitive load with formatting and linters

Cognitive load can be interpreted as the amount of information we need to keep in our brains while coding. There are many ways to reduce cognitive load within your code. Providing consistency within the code helps reduce cognitive load. It also helps with scanning tools, onboarding, and avoiding errors.

Developers have enough to worry about even without the concerns regarding code formatting, comments, and similar. If we believe the number of spaces (or tabs) to indent a line would not get anyone riled up, we are in for a rude awakening. Seemingly trivial formatting decisions or coding practices may cause heated debates. Hopefully, many problems have been put to rest by the adoption of opinionated formatters, such as black when using Python and linters when using SonarLint.

We may find that legacy software in FinTech written in languages such as COBOL or RPG was constrained by line lengths of 80 or 132 characters and, in some flavors, required starting code in a specific column (specifically COBOL). Similar requirements are also present in RPG and other early languages. While modern languages have pretty much done away with those types of restrictions, developers quickly learned that maybe there was too much freedom. Today, many programming languages have various formatters that enforce some structure into our code, often forcing your code to adhere to a standard developed for the language. Whether it is Perl (perltidy), Go (gofmt), or JavaScript (Prettier), or Python (black), formatters are another tool we should investigate as soon as we start learning a new language.

Linters play a similar role by ensuring our code stays clean and does not fall into common language pitfalls such as identifying unused imports, unused variables, invalid comparisons, and problematic casting, to name a few. SonarLint is a popular linter available for many languages and IDEs. It also provides integration into SonarQube, a product that helps identify and manage problems. Other linters are also available, such as ESLint when working with JavaScript or TypeScript.

Both formatters and linters can be built into your IDE or are available through plugins within the IDE. So, it is not a big inconvenience to start using them. However, ensure that you are using something approved by the team and that it is configured correctly. These tools can also be helpful in making you a better programmer overall, as they often tell us why this is the best practice and how to avoid it. We often find ourselves reading why SonarLint has flagged something or looking up additional examples or further information on best practices because the linters flagged something.

2.4 Automated testing

Regardless of whether we are coding the initial script to parse the ACH file by hand or getting started with a generative AI tool, we want to ensure that we are using unit tests. We prefer a TDD approach and will often find that we favor it. However, we do not have to strictly subscribe to that approach to receive the benefits. We are looking for short development cycles, mixing testing and coding. We should have a pretty good idea at this point that parsing an ACH file properly is going to be somewhat complex task. However, reading a file and ensuring that we have all the lines is not an

insurmountable task, and we should be able to achieve it easily enough. So, why not start with a unit test that ensures we received all the lines from the file?

A continuing theme is the need for high-quality software and verifying our results, especially in the context of generated code. This section explores the use of unit testing with `pytest` to help validate both generated code and the code we built ourselves. We also discuss the need for smaller and faster feedback cycles while coding, and we will try to apply that to our code here as well.

Even before we try to write tests for parsing individual fields from the record types, we may want to ask ourselves questions such as

- Can I read a file?
- How many records are in the file?
- Can I parse a record of type 1? How about a record of type 5?

We start small with the initial steps and then go deeper into the actual functionality of our code. In the following sections, we will start with a well-formatted sample file and work our way through it by creating tests. Each section will illustrate the creation of a small unit test to validate our well-formatted file.

2.4.1 Testing the number of records read

We will test the number of records read from our ACH file before we start addressing any problems that we may have noticed in our generated code. This testing covers the first two bullet points we listed at the beginning of this section. Verifying the number of records may seem trivial, but it helps put us in the mindset of testing our code and helps verify that we are indeed accomplishing this critical task. Our unit test needs to verify the number of records from the file. We can get the record out by opening the file in most editors, or we can employ a command line to get the number of lines by using `wc -l` in Unix or a `Get-Content` and `Measure-Object` in PowerShell. If we are unsure how to get the number of lines from a file, we may want to ask generative AI if it has any ideas and maybe even learn a new trick or two!

Depending on how we got here, our test may be failing because of one of those missing record types or because of some other error such as an incorrect file path, permissions, or any number of other reasons. Now, as an exercise, it would be good to work on getting this first test up and running before we continue with the code. Otherwise, we can just continue by using the sample code provided on GitHub.

2.4.2 Parsing the records

You can find information about the ACH layout at https://achdevguide.nacha.org/ach-file-overview. Let's recall some of the main aspects of the file format:

- It is a fixed-width ASCII file.
- Records are 94 characters in length.
- Each line is known as a record.

- Each record contains fields that are at fixed positions.
- Order of records and fields matters.

This format makes processing and validating files more interesting. We do not have an XML Schema Definition that we can utilize. Nor do we have the freedom of formatting that XML and JSON provide.

Although it has constraints compared to other file formats, it was in use before the other formats were born. It would be an interesting challenge for us to ensure we process the file correctly and handle some of the tasks in daily ACH processing. We create a new Python project in our IDE and use the response from ChatGPT 4 as our starting point.

Listing 2.5 Starting point for our ACH parser

```
def parse_file_header(line):
    return {
        'record_type': 'File Header',
        'priority_code': line[1:3],
        'immediate_destination': line[3:13],
        'immediate_origin': line[13:23],
        # ... additional fields
    }

def parse_batch_header(line):
    return {
        'record_type': 'Batch Header',
        'service_class_code': line[1:4],
        'company_name': line[4:20],
        # ... additional fields
    }

def parse_entry_detail(line):
    return {
        'record_type': 'Entry Detail',
        'transaction_code': line[1:3],
        'receiving_dfi_id': line[3:11],
        'account_number': line[12:29],
        'amount': line[29:39],
        # ... additional fields
    }

def parse_addenda(line):
    return {
        'record_type': 'Addenda',
        # ... additional fields based on the specific addenda type
    }

def parse_batch_control(line):
    return {
        'record_type': 'Batch Control',
        # ... additional fields
    }
```

```
def parse_file_control(line):
    return {
        'record_type': 'File Control',
        # ... additional fields
    }

def parse_ach(filename):
    records = []

    parser_functions = {
        '1': parse_file_header,
        '5': parse_batch_header,
        '6': parse_entry_detail,
        '7': parse_addenda,
        '8': parse_batch_control,
        '9': parse_file_control
    }

    with open(filename, 'r') as file:
        lines = file.readlines()

        for line in lines:
            record_type = line[0]
            parser = parser_functions.get(record_type)

            if parser:
                records.append(parser(line))
            else:
                print(f"Unknown record type: {record_type} in line: {line}")
                continue

    return records


# Test the function with an ACH file
parsed_records = parse_ach('path_to_ach_file.txt')
for record in parsed_records:
    print(record)
```

We created a basic folder structure for the project consisting of

- `ach_processor`—Where our Python module lives
- `docs`—Where any necessary documentation lives
- `tests`—Where our unit tests live
- `venv`—Our virtual environment (to keep our project and dependencies separated)
- `README.md`—A markdown document to get into more details about the project and structure
- `requirements.txt`—A list of required Python dependencies that can be used by a CI/CD pipeline to build the project

Listing 2.6 Project folder structure

```
ACH_Parser/
├─ ach_processor/
├─ docs/
├─ tests/
├─ venv/
├─ README.md
├─ requirements.txt
```

After setting up the project, we run the `black` command to format the source code. With PyCharm, the IDE was smart enough to see the package installed and prompted us to set up Black from within the IDE since `black` was supported out of the box.

With that accomplished, we can begin working on parsing the ACH records. At this point, with our exploratory spike, we are only looking to simply parse the records. Input validation is an important aspect of processing any type of data as it is defensive coding.

Defensive coding

Defensive coding is a proactive approach to dealing with unanticipated input, errors, and general misuse that occur when users get their hands on your software. For instance, if you ask your user to enter a number between 1 and 5, they are likely to enter anything but the numbers 1, 2, 3, 4, or 5. You can expect them to enter a, b, %, 1231482, nonprintable characters, and a wide range of other inputs!

The practices of input validation, error handling, fail-safe defaults, output sanitization, logging/monitoring, and static analysis are some aspects of defensive coding. As we move on with the project, we will keep these principles in mind.

2.4.3 File header record (type 1)

A file header record contains important information about the institution the file came from and what it is destined for. Some file details, such as creation date and time, can also help determine whether the file has been loaded before, although we will rely on hashing the contents of the file as well.

A powerful feature of ChatGPT is the ability to remember conversations; if we are signed into ChatGPT, we can go in and ask it to expand the parsing of the `parse_file_header` routine. This will give us another good starting point, and then we can even go in and ask it to create a unit test for the header record as well.

However, when we tried this approach, we ran into a few problems such as the file creation date being eight positions instead of six (it used a four-digit year). The record itself was not 94 bytes long, which also gave an error during parsing. We used the sample.ach file and that header record as an expected result and then ran the `pytest` against that.

Let's look at the unit test and break it down (listing 2.7). We start by defining the function name, and by convention, the name begins with `test_`, which helps identify the function as something for `pytest` to pick up and run.

Next, we have our sample line that is going to be passed to the parse routine. We also use `expected_result`, which is defined as a dictionary. We could also embed this directly into our `assert` statement, but for clarity, it is often easier to break it. By using a dictionary, we also employ our IDE. For instance, PyCharm provides a nice comparison window if this test fails, where we can see where the difference is. We then define the parser and call the routine with our `sample_header`.

Finally, we have an `assert`, which is where the actual check is done. If the result is not the same as the `expected_result`, we will see the error message. We can include as many `assert` statements as necessary. For instance, we may assert the response of an HTTP call was successful prior to checking fields that we expected to be returned.

The subsequent parsing of other records will also follow this code pattern, so whether we code it by hand, copy/paste, or use generative AI, we should be able to come up with similar tests for the other record formats.

Listing 2.7 `pytest` for parsing a file header record

```
def test_parse_file_header():
    sample_header = "101 267084131 6910
➥001340402200830A094101DEST NAME
              ➥ORIGIN NAME            XXXXXXXX"

    expected_result = {
        "record_type_code": "1",
        "priority_code": "01",
        …
        "reference_code": "XXXXXXXX",
    }

    parser = AchFileProcessor()
    result = parser._parse_file_header(sample_header)
    assert result == expected_result,
➥ f"Expected {expected_result}, but got {result}"
```

Defines a record that we want to test the parser against

Defines a dictionary that contains the parsed fields

Creates an instance of our AchFileProcessor class

Calls the function to parse the file header and return a result. This method is meant to be private, but for ease of testing, we are accessing it directly.

Compares the returned result to the expected result; this causes pytest to throw an error if they do not match.

We could certainly have told ChatGPT to create unit tests for us as well. As mentioned earlier, we are looking to "trust but verify" when it comes to using these tools. It may be possible to generate both the code and the tests, but given that we are also trying to understand ACH better, that may be counterproductive. For instance, if the tool generated a bad function for us to parse the file, it may also be likely to generate incorrect unit tests. We may end up with unit tests that appear to pass and validate the code only

to find they are both wrong. When generating the code, we can understand it better by writing the unit tests ourselves (or generate the unit tests and write the code).

Still, there are benefits to seeing what may be generated when asking for unit tests. We prompt ChatGPT with

CK Given the python function below to parse a Nacha ACH File Header, please provide unit tests for it.

We are presented with the following.

Listing 2.8 ChatGPT generated `pytests`

```
def test_parse_file_header_with_incorrect_length(self):
line = "101094000123"  # Truncated line
# Assuming your method raises a ValueError for incorrect input
with pytest.raises(ValueError):
   self.parser._parse_file_header(line)

def test_parse_file_header_with_invalid_record_type(self):
line = "201 0940001234567890123200602161645A094101
➥ Your Bank          Your Company       12345678"
# Assuming your parser validates the record type
# and raises an exception for invalid types
with pytest.raises(ValueError):
   self.parser._parse_file_header(line)

def test_parse_file_header_with_extra_whitespace(self):
line = "1  094000123   4567890123 200602 1616A 09410
➥ Your Bank          Your Company       12345678  "
# Assuming your parser can handle extra whitespaces gracefully
   expected = {
   "record_type_code": "1",
   "priority_code": " 0",
      "immediate_destination": "94000123",
   "immediate_origin": "4567890123",
   "file_creation_date": "20060",
   "file_creation_time": "2161",
   "file_id_modifier": "6",
   "record_size": "A 0",
   "blocking_factor": "94",
   "format_code": "1",
      "immediate_destination_name": "Your Bank",
      "immediate_origin_name": "Your Company",
      "reference_code": "12345678",
   }
   # This test might fail based on the specific handling
   # of spaces in your implementation
   assert self.parser._parse_file_header(line) == expected

# Add more tests as needed for your implementation specifics
```

pytest.raises can be used to validate that specific errors are thrown, which is great for testing edge cases and exception conditions.

Extra white space caused invalid values to be used—most obvious was the record_size switching from 094 to "A 0".

We can see that these unit tests cover incorrect lengths of lines, invalid type, and extra whitespace. The invalid type and incorrect length lines may be useful tests, but we are

unsure what happened with the extra whitespace test. It looks as if it was meant to add trailing space, but it also added spaces within the record itself. Since this is a fixed-length record, this will obviously lead to validation errors later. Still, it did provide us with some direction for additional tests when we get to that point. In addition, we saw how we may validate expected errors using the `pytest.raises` syntax.

Of course, right now, we are following the happy path, not trying to focus too much on the input validation. We simply want to see whether we can get an ACH file parsed at this stage.

2.4.4 Batch header record (type 5)

As the name implies, batch header records indicate the start of a batch for a particular company. There can be multiple batches per file and all the entry and addenda records that are contained within the batch belong to that company.

For the parsing of the batch header record, we can return to the IDE and let Copilot get us started, and then also ask it to help us define a `pytest` as well. We use the following prompt:

CK # Define a test function to test parsing an ACH batch header record using `AchFileProcessor._parse_batch_header()`.

We can review that in `test_parsing_file_header.py`.

While the initial function to parse the batch header looks reasonable, the `pytest` itself needs more work in terms of the `sample_batch_header` that gets passed to the `_parse_batch_header`. The line is 181 characters long, which is more than the fixed 94 bytes required by the format. It looks as if the name of each field was put into the position of the test record. Also, on further inspection, we find that it does not include all the fields in the record format. However, when we began typing the name of the field, we were impressed to see the field was populated and used the data from the sample, although with mixed results.

Still, it is a very impressive result and more than enough to get us started and complete a passing test using the sample.ach file we have been working with. After working on the parsing routine and the expected result, we were able to pass another test.

Hopefully, you were able to parse this record and add a unit test on your own. If not, no worries—there are still more types ahead and plenty of chances to give it a try! Even though it might be jumping the gun, we also want to look at behavior-driven development (BDD) and show a few sample tests that could be used if we were looking into that type of approach.

What is behavior-driven development?

Behavior-driven development is an approach to developing software where we create tests around the behavior of the software, and then use them as part of the acceptance criteria for determining whether a project is complete. What makes this type of testing unique is that all stakeholders work closely together to develop these

scenarios. Formalizing these user requirements helps ensure that we build a project that meets users' expectations.

Each BDD test is described by a series of `given/when/then` statements that directly relate to desired functionality. We can run these tests from our IDE just like we did with our unit tests. The difference is just in the approach to designing the tests and the level at which they execute.

As we progress through the project, we will look to expand our BDD testing; for now, we create a very simple test that confirms we have successfully parsed the record type field. While we have a unit test that confirms this already, it will make for a simple introduction to setting up a BDD-style test.

We first define a feature file, which identifies the feature that we are testing along with various test scenarios. The feature file is written in a human-readable language so that anyone can make sense of it. The `batch_header.feature` is included in the code and shown in the next listing.

Listing 2.9 Parsing an ACH batch header

```
Feature: Parsing a Nacha Batch Header

  Scenario: We have a record type of 5
    When We parse the batch header
➥"5200Company name    DiscretionaryData    Company
 IDARCComp desc 021623230216047106100001000000l"
    ➥ Then The record type should be 5
```

We define the feature and then a simple scenario of "We have a record type 5." The scenario can be named anything, but obviously, we want to convey what this test is going to do. The `when` and `then` statements are where the real work happens—in our example, *when* we are parsing the header record, and *then* we want to ensure we have a record type of 5.

Once we have that information, our next step is to wire the feature file together with what are known as *step definitions*. This is how we translate the human-readable text into something that we can execute with `pytest`. For brevity, we will show a sample of the "then" step definition. The rest of the code is available in `test_batch_header.py` for you to browse.

Listing 2.10 Behavior-driven development testing in Python

```
@then(parsers.parse("the record type should be {type_5}"))
def then_record_type_code_should_be(record, type_5):
    assert (
        record["record_type_code"] == type_5
    ), f"Expected {type_5}, but got {record['record_type_code']}"
```

We decorate the function with `@then` and parse the text string. Within that string, we have `{type_5}`. This is a dynamic value that will be pulled in from the feature file. So, while we used "the record type should be 5," the 5 becomes a parameter to the function, and we could easily create other scenarios where we test other values. We then define the function, passing it the value we parse, a record (which is a Python fixture—more on that later), and an `assert` statement that we have seen before. This can then be run along with any other test, and we are done coding when this test passes. Of course, we already coded this, but in later chapters, we will work to define these scenarios beforehand, when we work on features. We could potentially come up with additional scenarios for each field or simply expand the "then" portion of our test to include multiple `then` statements for each field.

2.5 Entry detail record (type 6)

The entry detail records contain individual transaction data, including the account number and amount to credit or debit the account. Keep in mind that the parsing of the records can vary slightly, based on the type of records being processed. The batch header record contains the type of entry records present in the batch, and this is known as the standard entry class (SEC) code.

The parsing of an entry detail record proved more challenging for ChatGPT and in our formulation of the prompts for it. Initially, we tried the expanded prompt

CK Fully define the `_parse_entry_detail` method and provide a `pytest` to validate it.

While ChatpGPT did provide a method and `pytest`, it failed to meaningfully parse the record. We tried again with

CK Please the full layout for the NACHA Type 6 record.

Then ChatGPT started calling the type 6 record an addenda record and parsing it with the fields associated with that record. We realized that there are different types of entry details records (CCD, CTX, PPD, etc.), so we tried to redirect with

CK Please use the full layout for the NACHA Type 6 CCD record.

While the system correctly identified the type as a cash concentration disbursement, it still referred to the entry as an addenda record.

Using Copilot produced better results, populating the correct field names as we typed and allowing us to quickly create a template that took just a little updating to make the test pass. However, the underlying theme we have seen so far is that while both tools are powerful, they need us to validate the results and not plug them in blindly. Thus, we can see that knowing the domain is important if we want to be able to validate our results.

2.5.1 Addenda record (type 7)

The addenda record contains additional payment-related information that applies to the entry detail record. There may be multiple addenda records per entry detail record.

When Copilot was prompted with

CK Define a function to parse a Nacha Addenda Record.

it produced several code suggestions. The following listing shows the closest valid suggestion. It uses all the required fields, but the offsets are not quite right.

Listing 2.11 Parsing function for an addenda record

```
def parseAddendaRecord(record):
    # Create a dictionary to hold the parsed record
    parsedRecord = {
        'recordType': record[0:1],
        'typeCode': record[1:2],
        'paymentRelatedInformation': record[2:80],
        'addendaSequenceNumber': record[80:87],
        'entryDetailSequenceNumber': record[87:94]
    }

    return parsedRecord
```

In addition to the general parsing being incorrect, we would prefer to use snake case for the dictionary keys. We can try refining the prompt again by defining a function to parse a Nacha addenda record using a dictionary and snake case for the keys. This provides the keys with the formatting we prefer, but we still need to update the offsets.

Listing 2.12 Parsing function for an addenda record using the snake case

```
def parseAddendaRecordSnake(record):
    # Create a dictionary to hold the parsed record
    parsedRecord = {
        'record_type': record[0:1],
        'type_code': record[1:2],
        'payment_related_information': record[2:80],
        'addenda_sequence_number': record[80:87],
        'entry_detail_sequence_number': record[87:94]
    }

    return parsedRecord
```

To get the correct parsing, we need to adjust `type_code`, `payment_related_information`, and `addenda_sequence_number`. The following listing shows the updated return statement that could be used.

Listing 2.13 Updated return statement with corrected fields

```
…
return {
   "record_type_code": record[0],
   "addenda_type_code": record[1:3],
   "payment_related_information": record[3:83],
   "addenda_sequence_number": record[83:87],
   "entry_detail_sequence_number": record[87:94],
}
…
```

2.5.2 Batch control record (type 8)

The batch control record is the trailer record for each batch and is a required record. We use fields from the record, such as total debit/credit amount and the record count, to validate that we received the correct batch contents. The following listing shows that Copilot took a different approach to parsing this record.

Listing 2.14 Copilot parsing a batch control record

```
def parse_ach_batch_control_record(record):
# Define a dictionary to hold the parsed data
    parsed_record = {}

    # Define the fields in the record
    fields = [
        "record_type",
        "service_class_code",
        "entry_count",
        "entry_hash",
        "total_debit_entry_dollar_amount",
        "total_credit_entry_dollar_amount",
        "company_identification",
        "message_authentication_code",
        "reserved",
        "originating_dfi_identification",
        "batch_number"
    ]

    # Define the field lengths
    field_lengths = [
        1, # record_type
        3, # service_class_code
        6, # entry_count
        10, # entry_hash
        12, # total_debit_entry_dollar_amount
        12, # total_credit_entry_dollar_amount
        10, # company_identification
        19, # message_authentication_code
        6, # reserved
        8, # originating_dfi_identification
```

```
        7 # batch_number
    ]

    # Define the field start positions
    field_start_positions = [0]
    for field_length in field_lengths:
        field_start_positions.append(field_start_positions[-1] + field_length)

    # Parse the fields
    for i in range(len(fields)):
        field = fields[i]
        start_position = field_start_positions[i]
        end_position = field_start_positions[i + 1]
        parsed_record[field] = record[start_position:end_position]

    # Return the parsed record
    return parsed_record
```

This code looks a bit of overengineered because the nature of this widely used fixed-position file format means that the field positions will not be changing. As a matter of personal preference, we like to see the actual offsets being used, as shown listing 2.12. The offsets will also make our job easier when dealing with errors in parsing a record. Having the field as `'entry_detail_sequence_number': record[87:94]` means we know where the field `entry_detail_sequence_number` begins and ends. We are also willing to allow these magic numbers to exist in the code because they are limited to this specific area and not sprinkled throughout the code. Of course, we could also create variables named `BEGIN_ENTRY_DETAIL_SEQUENCE_NUMBER_POS` and `END_ENTRY_DETAIL_SEQUENCE_NUMBER_POS` and use them if we find a compelling reason. Now, let us take a look at parsing the file trailer record.

2.5.3 File trailer record (type 9)

The final record in an ACH file is the file trailer record (also known as a file control record). The file trailer record provides fields such as batch count and entry/addenda count we use to validate the file was received correctly. Note that the format required the number of records to be a multiple of 10. So, you may find files or software that will pad created ACH files out with records that consist of all 9s. However, most software does not require this to be done anymore.

In the following listing, we are back to Copilot parsing the records as we would expect. The fields were all cast integers.

Listing 2.15 Parsing the file trailer record

```
def parse_ach_trailer_record(record):
    return {
        "record_type": record[0],
        "batch_count": int(record[1:7]),
        "block_count": int(record[7:13]),
        "entry_count": int(record[13:21]),
```

```
        "entry_hash": int(record[21:31]),
        "total_debit_amount": int(record[31:43]),
        "total_credit_amount": int(record[43:55]),
    }
```

While this code parses the records correctly, we are missing the last field, which is marked as reserved. So, while it may not be necessary at this time, we may still want to consider including it just for the sake of completeness.

We may also want to create a sample BDD test because, conceivably, subject matter experts (SMEs) may provide us with specific use cases for parsing this record (or any of these records). A BDD style test may look like the following listing.

Listing 2.16 BDD style test for type 9—file trailer record

```
Feature: Parsing a Nacha File Trailer

  Scenario: We have a record type of 9
     Given the File Trailer Record "9000010000010000000740198019800000000007825000114611480"
     When We parse the file trailer
     Then the record type should be 9
     And the batch count should be 10
     And the block count should be 1
     And the entry count should be 74
     And the entry hash should be 0198019800
     And the total debit amount should be 78.25
     And the total credit amount should be 1146114.80
```

2.5.4 Passed!

Wow, we made it! That was a whirlwind of coding and testing. Even with generative AI helping out, that was a lot to take in. Let us now recap what we have just accomplished. We started using small development cycles to build unit tests to validate the parsing of the various ACH record types. It is important to understand that we started by breaking the program created by generative AI into functions (we had asked ChatGPT to do this for us before we started). Having generative AI create functions allowed us to create unit tests more easily for each record type. Otherwise, we would have had to figure out a way to determine if the records were parsed correctly and check them after the file was loaded.

Each time, we started by creating a unit test that should fail and then coded just enough to make it pass. At this point, we should be relatively familiar with the process and ready to apply the same concept to other parts of our project.

2.6 The not-so-happy path

Congratulations! We just finished parsing what is very likely our first ACH file. We concentrated on what is sometimes referred to as "the happy path." This is where

everything goes as expected, without throwing any kind of errors. We used a single well-formatted ACH file throughout our parsing to illustrate the process.

We should now also consider the not-so-happy path, which is probably what we are more likely to encounter in our day-to-day coding. Our not-so-happy path will cover some of the problems that can occur when loading a file. There are scenarios where the file or a batch may be rejected or where an entry can trigger a rejection. We will examine exceptions and handling/recovering from problems with ACH files further in chapter 9. For now, we just want to touch on some of the possibilities.

We will identify some of the possible rejection scenarios and provide sample ACH files and `pytests` to code for them. When a file is rejected, we often must go back to the originator to request a new one. If any transactions have posted, they may need to be reversed as part of the rejection process. For now, we are focusing more on identifying bad files than recovering from them. Of course, we will provide a finished example that addresses the `pytests`, if you want to skip ahead. Obviously, we would encourage you to work through these scenarios as the particular scenarios we have chosen do not require extensive knowledge of ACH processing.

We will also be adding further validation later, as the project expands. For now, we are just expanding our proof of concept.

2.6.1 File rejection

ACH files can be rejected when formatted incorrectly. Remember that the order of the records is important. An ACH file consists of batches, and each batch contains entry and addenda records. Both a batch and the entire file have trailer records. All the records should be 94 characters long. So, for the first scenario, we want to tackle dealing with a file where records may not be the correct length.

Why would a file be produced with records shorter than 94 bytes? Before SFTP (Secure File Transfer Protocol) became common, we would encounter this when ftp was set to truncate trailing spaces. While ftp is not as prevalent as before, it could still be used internally to transfer files, so the original use case may be valid. In addition, files may be routed and retransmitted or even created/updated on someone's computer. As banks merge and acquire each other, these file problems may still pop up. Since the Nacha spec requires 94-byte records, we will seek to enforce that. We have included ACH_Parser_v3 containing empty unit tests to help us validate our changes.

For each of these challenges, we expect the parser to accumulate a list of errors. This list should hopefully keep the parsing code relatively simple because we are not worried about input errors at this point. This approach also has the added benefit of being able to validate the expected results in our unit tests. If we know our file has two short lines, we should also expect two of those error messages. For now, we will only log an error message if it is a string, but you may expand your project or choose another approach.

Code challenge; benefit 1

We started with a relatively simple program to parse our ACH file. Going back and handling these changes will help us gather some real-world experience because we are likely going to be maintaining and updating existing code more than writing code from the ground up.

The relatively simple change to check for a record length forces us to deal with exceptions and think about how we might want to identify, format, and store them. It also presents us with some choices in how we code this: Do we continue to store this unparsed record along with the other records? If not, where does it go, and how does a user know which record had an error?

Seemingly mundane choices can sometimes have a significant influence down the road. So, this is just something to keep in mind and watch for as we work through the program.

2.6.2 Batch rejection

Individual batches may also be rejected from a file, and one reason is that the trace numbers are not ascending. A trace number appears on each entry detail record (type 6). As the name implies, these trace numbers provide a way to identify an ACH transaction within a batch. The first eight digits of a trace number are the routing number of the Originating Depository Financial Institution (ODFI)—in other words, what bank or financial institution the transaction came from—and the last seven digits of the trace number are in ascending order (but not necessarily sequential).

A unique trace number provides a means to reconcile and report on these ACH transactions and the ability to trace a transaction through the payment system for both regulatory and compliance reasons. We can now expand the ACH parser to consider these trace numbers. Again ACH_Parser_v3 has some unit tests that we can use to verify whether our program parsed the file correctly.

Code challenge; benefit 2

This challenge gets a little more complicated in how we will choose where and when to handle these exceptions. We started off with our functions being named `_parse_entry_detail`. If we choose to update the code to handle verification in that parse routine, will we update that function name since it is no longer just parsing? Will one routine call another, or will we call two routines or just use one routine? Do we parse the record first and then check the trace number, or do we check it before parsing the whole record?

Sometimes, we want to make changes with surgical precision for both time and complexity reasons. Other times, we may opt for a shotgun approach when there are multiple things that must be accomplished. Either way, you should still be working in short cycles, making sure to test often.

2.6.3 Entry rejection

An entry may be rejected if the addenda flag is not consistent with the existence of an addenda record. The entry detail record contains a flag indicating whether the next record is an addenda record. Addenda records include additional information for ACH transactions and are sometimes required for specific SEC codes. With only 94 bytes to work with, it is sometimes necessary to have addenda records to pass additional information about the transaction along.

Code challenge; benefit 3

Coding this challenge requires us to look ahead or look behind as we need to be able to determine whether we expect an addenda record. This goes back to making decisions and then deal with them later.

For instance, the existing code had read all the lines and then used a `for-each` loop to iterate through them. Having the `for-each` loop is handy, but now we potentially need a way to index our list of lines. If we need to conserve memory because we expect large ACH files, we may not want to read all the lines in, and therefore, we would have to take another approach to finding an addenda record.

This is all part of the software development process. Thinking about and planning our changes will help us address some of these challenges and hopefully future-proof our code.

When we come across a bad design choice that we or a co-worker have made, learn from it. Even if it is annoying and makes you rework a lot of code, you can take something away from the experience!

2.7 Interpreting the code

Depending on your experience with Python, you may have some questions about the code that was produced as part of this example. The following sections examine the code in more detail to provide additional insight into the Python code we have written, covering the Python `switch` statement, type hints, and secure coding.

2.7.1 Where's my switch statement?

There are six record types for ACH that we need to parse. If you have some programming experience, you have likely come across the `if/if else/else` construct for flow control. In Python, we would see this as

```
if condition1:
elif condition2:
else:
```

At some point, we probably also coded rather large `if` statements with multiple conditions before someone introduced you to a `switch/case` statement. Early in Python (circa 2006), support for `switch/case` statements was debated and finally rejected.

However, as of Python 3.10, there is support for the match statement (https://peps.python.org/pep-0634/), which provides the means to create a `switch` statement, as shown in the next listing.

Listing 2.17 Creating a `switch` statement using `match`

```
parsed_record = match record_type:
     case '1':
         self._parse_file_header(line)
…
     case '7':
            self._parse_addenda(line)
…
     case _:
            print(f"Unknown record type: {record_type} in line: {line}")
            continue
```

Uses the match keyword to create cases for the record_type field

Each condition has a case statement.

The default case is identified with an underscore.

With millions of lines of Python code written before the `match` construct became available, generative AI is likely to show you the `if/elif` or another common practice of creating a Dictionary to store the choices as was done in the sample program.

Listing 2.18 Using a Dictionary

```
parser_functions = {
            "1": self._parse_file_header,
            "5": self._parse_batch_header,
            "6": self._parse_entry_detail,
            "7": self._parse_addenda,
            "8": self._parse_batch_control,
            "9": self._parse_file_control,
        }
…
                parser = parser_functions.get(record_type)
…
                    self.records.append(parser(line))
```

Defines a Dictionary of functions that can be called

Retrieves the function from the Dictionary based on the record_type

Calls the parser for the given line and appends the results to the records variable

This code declares the functions for each type and calls the appropriate function. One thing that would make us favor another approach is that we cannot see whether the parameters should be passed to the function. We may not even be able to tell if they are functions. One way to handle this is using type hints.

2.7.2 Type hints

Type hints have been around since Python 3.5 and were defined in https://peps.python.org/pep-0484. While Python uses duck typing, the addition of type hints can

help make the code more maintainable. In our opinion, it goes back to having to choose between freedom to code and the code longevity. When writing Python code, it is great not to be constrained by static typing; instead, we can focus more on the code. However, as more Python code comes into existence, or when we must start looking at other people's code, we want some of those guides. So, while Python does not enforce type hints at runtime, there are tools that can be used to do static type checking based on these type hints, as well as just the benefits of documentation.

We also wonder whether the popularity of Python and the migration of developers to it from other languages have also brought some of their baggage. We believe these additions to the language are important, and as they have gone through the Python Enhancement Proposal (PEP) process, the community agrees. Of course, our backgrounds are in statically typed languages, so these make sense. If we came from LISP or Scheme, would we want to see a bunch of parentheses added to the language?

Our original problem with the dictionary approach to flow control was that we did not have a way to tell what parameters could be passed to the functions we were calling. We can update the `parser_functions` to use type-hinting:

```
parser_functions: Dict[str, Callable[[str], Dict[str, str]]] = {
```

The variable `parser_functions` is now defined as a Dictionary that contains a string (`str`) for the record type and a `Callable` that represents the function being called. The `Callable` takes a list of parameters, and in this case, we see it takes one string that is denoted by `[str]` and that it returns a dictionary of strings, which is our parsed record.

After seeing the type hint for this, maybe we decide to rewrite our code to take advantage of one of those other constructs because that seems confusing! By embracing the use of type hints, we could then move to static type checkers such as mypy, Pyright, and Pyre. In our opinion, static type checking is a must-have for enterprise applications where there are large teams or the project is long-lived (and we assume they will be). Other developers diving into the code base for the first time or after a hiatus on the project will find them immensely helpful. Of course, Python is going to remain a dynamically typed language (see https://mng.bz/oKEM), but the proliferation of tools and their adoption should show that there is at least some benefit to type checking.

2.7.3 Secure coding

There has been concern that generative AI may not be producing code that is secure or code that is even good. This is not only true for generative AI but also for humans: we all make mistakes. Going back to our tool analogy, we need to remember that generative AI is just another tool we should be using, which is why we advocate the use of formatting and linters to help identify problems within our code.

Corporations use a number of tools available on the market to scan their code looking for flaws and security holes. These tools include Fortify on Demand, Veracode, Snyk, and Amazon's CodeWhisperer. The primary goal of this software is to look for

insecure coding patterns. They often use the OWASP 10, SANS Top 25, and other lists as guides to the problems they identify.

Many times, these are incorporated into a CI/CD pipeline; however, we also have options for handling them inside our IDE as well. Since The Fantastic Fintech Company uses JetBrains, we can take advantage of some of their included features. For instance, in our requirements.txt for our Python project, PyCharm will detect vulnerabilities in our dependencies (see figure 2.5). We can see the Common Vulnerabilities and Exposures (CVE) ID and a short description of the vulnerability.

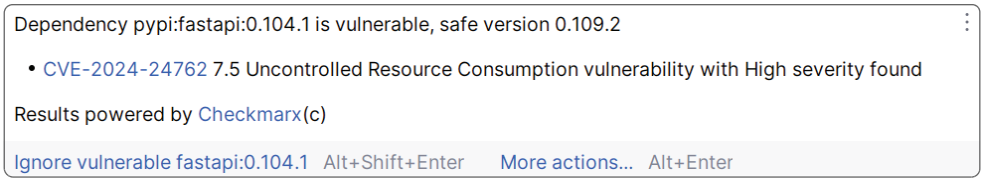


Figure 2.5 An example dependency

By clicking More actions, we can simply choose to update to a newer version, as shown in figure 2.6. Of course, this action would require further application testing, but it is certainly nice to be alerted to potential problems before our code even makes it to the CI/CD pipeline.

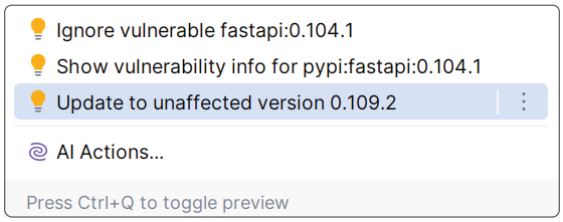


Figure 2.6 Options to handle vulnerability

Chances are most companies have either adopted a static analysis tool or are currently evaluating them. Having the ability to allow developers to view and troubleshoot problems directly in the IDE is another way to boost productivity.

Summary

- We explored how to incorporate generative AI into our development workflow, learning what to ask (prompt engineering) and how to clarify our intent (problem formulation).
- We focused on modernizing an ACH system running on legacy mainframe architecture using metrics such as release cadence and defect backlog.
- We utilized ChatGPT and GitHub Copilot to understand ACH file layouts better while ensuring that no confidential data is exposed.
- ACH network facilitates seamless money transfers between banks, and it is commonly used for transactions such as payroll and bill payment.
- ACH files are fixed-width ASCII files with specific record types that require careful parsing of file header, batch header, entry details, addenda, batch control, and file trailer records.
- ACH parsing can be validated using unit tests with TDD to obtain reliable, high-quality software.
- Linters and formatters such as SonarLint and black can be used to enforce coding standards and reduce cognitive load.
- Scenarios in which ACH files may have errors can be handled using robust code that accounts for unexpected inputs and edge cases.
- Type hints can be implemented in Python for enhanced code clarity and maintainability, which helps easier collaboration and onboarding.
- Security concerns can be addressed by using static analysis tools to identify coding vulnerabilities and ensure compliance with security standards.
- This chapter reiterated the importance of robust testing practices, iterative development, and early detection of potential problems using IDE plugins.

Part 2

A path to the minimum viable product

Now that you understand the layout of an ACH file, it is time to start building a functional ACH dashboard. If it sounds daunting, don't worry! We will first build various components and then pull them all together into a minimum viable product (MVP).

Chapter 3 begins by setting the environment using Docker. Here we explore the basics of Docker and how to use it when building the project.

Chapter 4 will allow you to get used to working with APIs in Python by utilizing the FastAPI framework. Building basic RESTful APIs to support the dashboard and exploring documentation using OpenAPI will keep you busy.

Chapter 5 discusses database design for storing ACH files inside a Postgres relational database. You will have the opportunity to design a database and see some of the benefits and pitfalls of different designs.

Chapter 6 explores creating a user interface with Next.js, taking us from the backend to the frontend.

Chapter 7 gathers all the pieces created in the previous chapters and puts them together in an MVP that you will be able to turn over to the stakeholders.

At the end of this part, you should have an ACH dashboard that you can be proud of. This is really the starting point of the project. After the project goes live, you will begin receiving various enhancement requests from the stakeholders.

3 Getting started with Docker

This chapter covers

- Creating containers for our project components using Docker and Docker Compose
- Optimizing Docker containers and images for enhanced security and maintenance
- Implementing general Docker maintenance to ensure our containers and images do not eat up disk space
- Using health checks to ensure the system is up and running

In the last chapter (or sprint, if we are thinking in Agile terms), we created a functional (albeit basic) ACH parser. We also experimented with generative AI tools to help us work faster and more efficiently. We have made some progress exploring tools and getting our bearing with ACH processing. Enjoy that feeling because there will surely be days when we log off the computer feeling completely drained.

At this point, we have been given project requirements and a general framework of what a final project should contain. Of course, there are different approaches that

we might take when working on a project, each having its pros and cons. While we may end up favoring one approach over others, there are always factors to consider when tackling a project, such as

- How does the size and complexity of the project compare to our team's experience and knowledge?
- Have the project requirements been well defined, or is the business/customer still feeling the project out? How do the requirements compare to the time allotted for the project?
- What is our risk tolerance for items such as bugs, delays, new technologies, and similar?
- Do we have existing infrastructure for the project, such as testing environments and automated builds? If not, will we need to factor in building that infrastructure, or will that come later?

While the previous points may seem more pertinent when having more of a leadership role (technical lead, architect, etc.), it is beneficial to have everyone thinking about them because different perspectives are always useful. We just need to be careful that our feedback comes across as constructive and not as an attack on our co-workers (and we should be careful about when it is obvious any feedback will be ignored). Let us get back to our project at hand.

3.1 Where to begin

First, you may not have any choice in what you are able to work on. The business needs may dictate that you work on these pieces in this order. However, assuming you have a choice on where to start, what would you pick? You could start with a piece that you are comfortable with—perhaps you are a strong Python programmer, so you want to finish working on the ACH parser.

Or, maybe you should pick the thing you know the least. Maybe you have never worked with Next.js before, and you want to ensure you have enough time to devote to that part because you feel the other pieces won't take much time.

We like to get started with the whole project! Well, not the whole project, but we like to see if we can get all the pieces talking to each other in some form. Even if it is the equivalent of a bunch of "Hello, World!" programs and components, at least we have the pieces together, and then we can pick them up, start building them, and let them evolve. It is really up to you—you could just as easily jump around to other chapters and get the various components up and running. This chapter focuses on getting a Docker setup and then putting our basic project components in place. Subsequent chapters will build on these components, but for now, we want to create an environment that looks similar to the one in figure 3.1.

As you can see, this environment will provide us with a UI container running Node and Next.js, an API container powered by FastAPI, and a database container running PostgreSQL. Running these inside Docker also allows the introduction of other

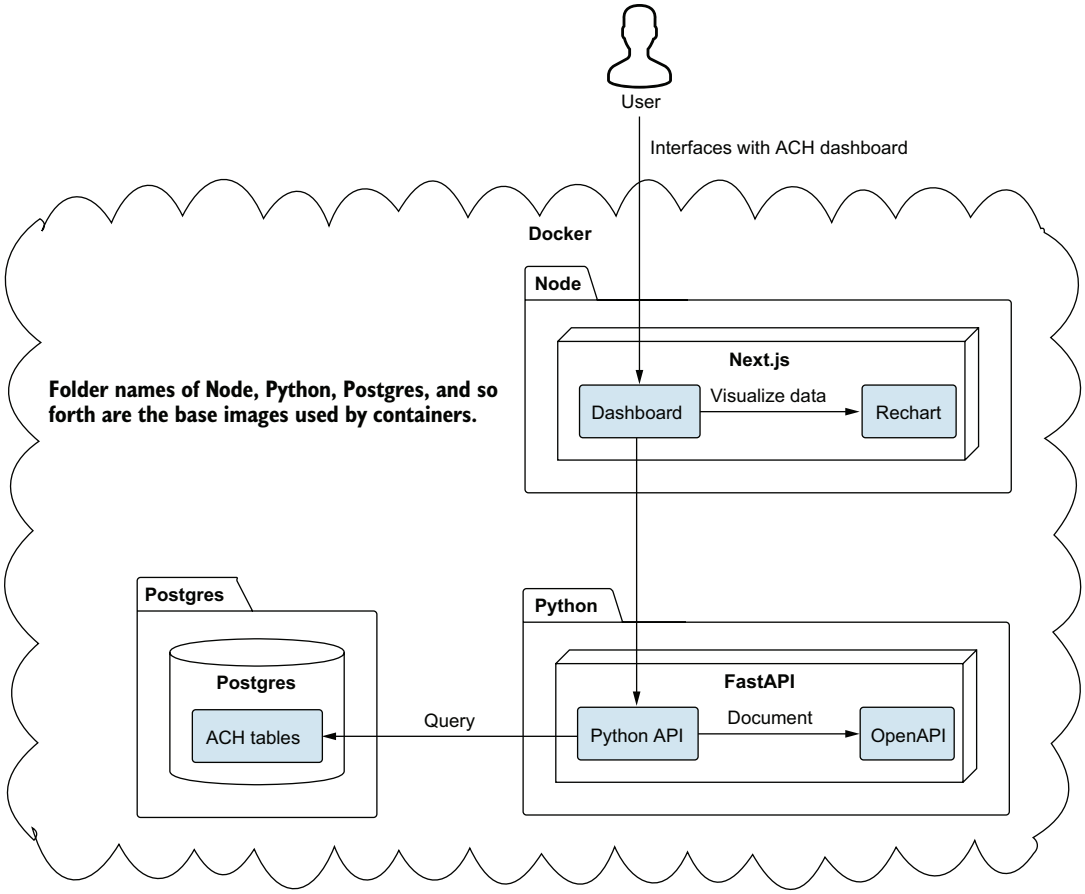


Figure 3.1 An overview of our ACH system

components that we will explore in other chapters. While this project takes a more service-oriented approach to architecture, this general approach of using Docker to break our software down paves the way for other architecture patterns. The goal is not to make you an expert in Docker (check out *Docker in Action* [2019, Manning] by Jeff Nickoloff and Stephen Kuenzli if you want to become an expert) but rather to show you how to use it to explore different technologies and software without having to install and configure it on a host machine. It really helps you stay more organized.

Is the monolith dead?

Since we are using containers, and you may have heard about Microservices and how they are the latest and greatest, you might be tempted to run back to your company looking for any monolithic software and start insisting that they are doing it wrong.

(continued)

We would advise against that. Just because something is new does not necessarily mean it is better. Monolithic architecture can have its drawbacks, but there is no silver bullet when it comes to technology. Change for the sake of change is not only unnecessary—it can be detrimental to the software.

Therefore, we suggest that if it is not broken, do not fix it. If there is monolithic software running at your company, it may continue to run for long after you have left the company. Only after identifying problems with the architecture should you look for alternatives.

Monolithic architecture can also be very useful in developing proof of concepts, as you can bundle and deploy everything in one easy package. We believe everything has its time and place, so while we encourage forward thinking and engineering, remember that part of Agile is looking to deliver the right product at the right time. It just may not be the right time to explore moving to containers.

3.2 Creating a docker-compose.yml file

If you have never worked with Docker before, you are in for a real treat. Docker allows us to create and distribute various containers. A container is sort of a mini-VM that you build in a declarative fashion and can be pulled in by others. If you are a Java programmer, think of your pom.xml and Maven, where you can basically describe your project and provide that file to someone else, so they can use it to have Maven pull the necessary files. Containers are similar, except they work on a larger scale. In figure 3.1, we showed a solution that has multiple containers running Nginx, FastAPI, and PostgreSQL. When working with a multicontainer application, we want to be able to manage everything from a centralized location, and that is where Docker Compose comes into play. We can declare a docker-compose.yml file and configure all the applications, containers, and services in one location.

If you are new to Docker, you might try to use Copilot, but our attempt did not yield any meaningful results. We tried to use the following command with Copilot:

CK
```
# Create a sample docker compose layout for Nginx, CloudBeaver,
PostgreSQL, and FastAPI
```

The results of that prompt are shown in the following listing. Unfortunately, it did not produce any actual executable code.

Listing 3.1 Failed attempt at a Docker setup

```
# Create a sample docker compose layout for
➥ nginx, cloudbeaver, postgres, and fastapi
# This is a sample docker compose file for a fastapi project
# It is not meant to be used in production
# It is meant to be used as a starting point for a fastapi project
```

We had much better results asking ChatGPT directly, as shown in the next listing, as it provided an actual compose file.

Listing 3.2 docker-compose.yml generated by ChatGPT

```
version: '3.7'

services:
  nginx:
    image: nginx:latest
    container_name: nginx_container
    ports:
      - "80:80"
    volumes:
      - ./nginx:/etc/nginx/conf.d
    depends_on:
      - web

  web:
    build:
      context: .
      dockerfile: Dockerfile.fastapi
    container_name: fastapi_container
    volumes:
      - ./app:/app
    expose:
      - "8000"
    depends_on:
      - db

  db:
    image: postgres:latest
    container_name: postgres_container
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: sample_db
    volumes:
      - postgres_data:/var/lib/postgresql/data

  cloudbeaver:
    image: cloudbeaver:latest
    container_name: cloudbeaver_container
    ports:
      - "8080:8080"
    depends_on:
      - db

volumes:
  postgres_data:
```

The version tag is obsolete, but even 3.7 was outdated when ChatGPT provided this answer; this is an example of the need to verify responses.

ChatGPT gives us a nonstandard name of Dockerfile.fastapi; instead, we should use Dockerfile with no extension or fastapi.Dockerfile (see https://mng.bz/xKoB); this is another example of the need to verify responses.

These environment variables allow configuration for the Postgres database.

These were good starts, but we need to back up a step so that we can break the code down and get something even shorter. Because we relied too heavily on generative AI, we started larger than we normally would, and that is a good lesson for us to take away. We

spoke about the need for quick feedback loops and short cycles. It is never a good idea to change too much code in one session because when verifying our changes, we are unsure what actually addressed our problem. Therefore, asking generative AI to build four or five containers without a good understanding of what needs to happen was a bad idea.

Now that we have gotten a dose of reality and taken a step back, let us pick a single container to get up and running.

3.2.1 Creating a CloudBeaver container

We will start by building out a CloudBeaver container. CloudBeaver is a database management tool that supports multiple databases through a simple web-based user interface. There is no shortage of database management interfaces. You may also want to check out pgAdmin (https://www.pgadmin.org/) since we are working with a Postgres database or DataGrip by JetBrains (https://www.jetbrains.com/datagrip/) since we'll be using other JetBrains IDEs in this project. We do not have any vested interest in any particular product but are hoping to share what we have seen in use in FinTech.

We are installing CloudBeaver because we want another way to view and manage our database other than just using code. Your company probably has a preferred tool, and you can feel free to use it. However, CloudBeaver is simple to get up and running.

So, why are we setting up a database management tool before our database? That is a good question. Because CloudBeaver is a web interface, we will be able to confirm the container is up and running simply by opening a browser and navigating to a URL. So, CloudBeaver provides a quick and simple way to get started and verify our progress. Another option is to use our IDE, as IDEs often provide integration (or a plugin) to browse the database. Of course, using our IDE gets away from Docker, and we are looking to explore different containers with Docker.

With that in mind, let's get started by prompting ChatGPT:

CK Create just a basic Cloudbeaver service in Docker Compose.

We receive the following output.

Listing 3.3 Sample CloudBeaver container output from ChatGPT

```
version: '3.1'
services:
  cloudbeaver:
    image: dbeaver/cloudbeaver:latest
    container_name: cloudbeaver
    ports:
      - "8080:8978"
```

The version tag is obsolete, but even 3.1 was outdated when ChatGPT provided this answer; this is an example of the need to verify ChatGPT responses.

The output looks plausible, but we see the version number is `3.1`. The latest Docker compose version was `3.8`, but now the `version` tag has been marked as obsolete by the project. While we can still specify it for the time being, using it will show a warning

message. If it were ever to become an error, that would certainly pose a problem. So, we tell ChatGPT:

CK Update the basic setup to remove the Docker Compose version as it is now obsolete.

With that, our file is updated. The ability to update output from our conversation is a powerful aspect of generative AI that we should keep in mind. Of course, we could have simply removed it from the produced file; however, we hope that our updated prompt will feed back into the system, and we will eventually stop getting the `version` tag as part of generated files.

We can copy and paste the output from listing 3.3 into a docker-compose.yml file, navigate to the folder, and then issue the command `docker-compose up`. On our machine, it failed to do this because of the following error:

```
Bind for 0.0.0.0:8080 failed: port is already allocated
```

This error is pretty straightforward, but remembering what you have running that might be using the port is not always straightforward. We could also update our docker -compose file to use a different port, but at this point, we can find the offending process on Windows PowerShell by using `netstat -aon | findstr "8080"` and then `tasklist | findstr PID` where PID are the process IDs that we just came across (which are on the right-hand side). Once you do that, you can either determine whether you want to shut down/stop whatever program you have running or adjust the port being used. In our case, another Docker container was being used, so it was safe to shut down. At this point, running `docker-compose up` works, and we can navigate to http://localhost:8080/ and see a web interface shown in figure 3.2.

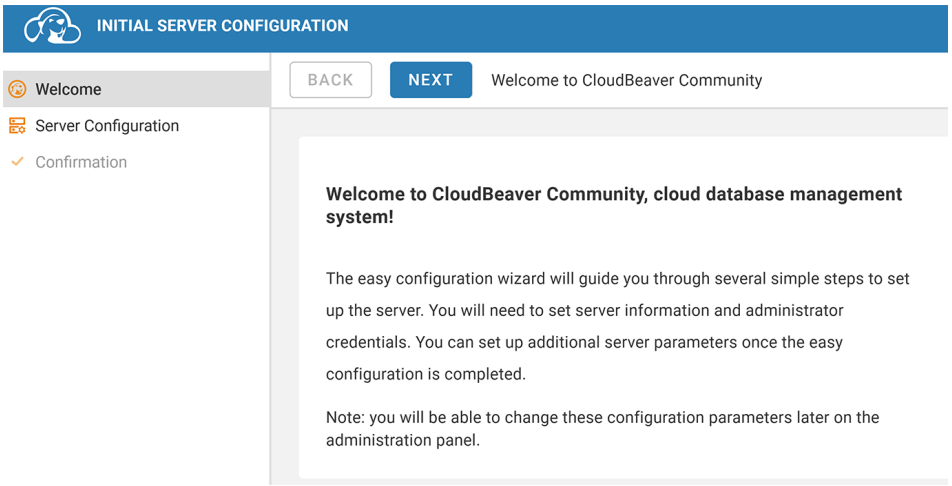


Figure 3.2 CloudBeaver welcome screen

3.2.2 Create a PostgreSQL container

Things will get a little more interesting now because we get to add a database container. Eventually, the database will store our ACH files and associated data, but for now, we will keep things simple. We can also populate the database with data when the container starts, so that we can use CloudBeaver to connect to the database and view our data. We can ask ChatGPT to simply

CK Update the docker-compose file to also include a `postgres` service.

Again, that created a helpful starting point, shown in the next listing.

Listing 3.4 Multiple containers generated by ChatGPT

```
version: '3.8'
services:
  cloudbeaver:
    image: dbeaver/cloudbeaver:latest
    container_name: cloudbeaver
    depends_on:
      - postgres
    ports:
      - "8080:8978"
    environment:
      - CB_DATABASE_URL=jdbc:postgresql://postgres:5432/postgres
      - CB_DATABASE_USERNAME=postgres
      - CB_DATABASE_PASSWORD=postgrespwd

  postgres:
    image: postgres:latest
    container_name: postgres
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgrespwd
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  postgres_data
```

However, let's take a step back and build a simpler file, which will serve us better at this time because we are still trying to adhere to the principle of YAGNI ("You aren't going to need it") as much as possible. Some purists might argue that even creating these Docker containers violates that principle. We do not strictly need any of this for our project at this point—we may not even need a database. Perhaps, we could get away with just writing to a file or keeping things in memory! That could certainly be true, but we could also look at this as the existing code because our application is built on

PostgreSQL and so that is going to be a requirement regardless of whether we really need it.

What is YAGNI?

YAGNI is an acronym for "You aren't going to need it." It is a principle of extreme programming and ties back to our Agile principles, such as minimum viable product (MVP), by trying to enforce the idea of avoiding unnecessary work and complexity. Sometimes, when coding, it is hard to resist adding that extra function that you're pretty sure you will need later, so you might as well code it now. Then, that code never gets used but is flagged by static code analysis for fixes and requires changes when you update some object.

So, if nothing else, just keep the principle in mind to avoid that extra complexity.

Let's go back to building our container. We add a simple container with the following (you can view the whole file in the provided v2/docker folder on GitHub for this chapter):

```
Postgres:
  image: postgres:latest
  environment:
    POSTGRES_USER: admin
    POSTGRES_PASSWORD: secret
```

With this, we can navigate to CloudBeaver with http://localhost:8080 and set up a connection; however, we do not have any data to view. We want to be able to load some data when the container is built, so we will look into doing that now. Note that we removed the networking and volume fields from the generated Dockerfile because we are going to rely on the networking that is set up by Docker. By default, all the containers defined with our docker-compose file under services will be able to communicate with each other. Also, we are not interested in persisting data in this project (at least for now), so we removed the volumes tag. This gives us a simple starting point.

To add some data, we will create a SQL script that executes when the container is first built. We start off by creating a `health_check` table. This table could be used to determine whether our application is in a valid state. For us, it is more of a "Hello, World!" scenario than anything meaningful.

Health checks

Because our application will be spread across a number of containers, it is often useful to create a health check. It is a common pattern to follow when using microservices. Even though we have started by initially creating a database table named `health_check`, this is just the beginning.

(continued)

Docker allows us to create a health check for the container within our setup, which we will explore later. This test can determine whether the container is "unhealthy." When using a container orchestration tool such as Kubernetes, these health status checks can trigger alerts to admins, as well as automatically restart containers.

Tools such as Kubernetes also have expanded these checks to differentiate between the app being unavailable because it is initializing and because something crashed. These are referred to as "readiness" and "liveness," respectively.

Now that we have a little more robust container, we will be working with four `docker-compose` commands:

- `docker-compose build`—We use this command when we need to build the container as specified in the Dockerfile. Any updates to files/scripts that we are including in the container will require us to issue this command.
- `docker-compose up`—This command is used when we need to start our containers.
- `docker-compose stop`—We use this command when we want to stop our container and leave the data in place. While important, we will typically use the `down` argument because we will rely on our initialization scripts to populate the database and start with a fresh database each time.
- `docker-compose down`—This command is used when we need to stop and remove the containers. We use it to make changes to any of our initialization scripts because the initialization will not occur when the database exists.

We'll require these commands as we make changes to our Dockerfiles. Usually, for development, we just chain them together as `docker-compose down && docker-compose build && docker-compose up`. You can even slim the command down to `docker-compose down && docker-compose up --build`.

Initially creating our containers may take a bit of time, but the process will be significantly faster when making small incremental changes. We should now be able to view our table and its data through CloudBeaver.

3.2.3 API container

Our next container expands on our container experience by building a container for our APIs. Soon, we will begin to build out specific APIs for dealing with ACH files, but for now, we will keep things very basic. We want to make use of the FastAPI framework for running our APIs in Python.

We will create a Python container from the standard Docker Python image (an image in Docker is a standard read-only template used to create a container) and install FastAPI and some dependencies on top. In our previous containers, we worked directly in the docker-compose file to create our containers. With this container, we will create

a Dockerfile that will be referenced by the docker-compose file. To store the dependencies and configuration of the image, we need to introduce a few new commands:

- `FROM`—Specifies a base image to start building the image from.
- `COPY`—Copies directories/files from the local machine to the image. In the upcoming section, we will create a requirements.txt that will need to be copied to the image that contains FastAPI, among other things.
- `RUN`—Executes a command in the image during the build process. We will execute `pip` to install the required Python packages.
- `CMD`—Specifies a default command to run when the container starts. We will use this command to run Uvicorn, which is an Asynchronous Server Gateway Interface (ASGI) that we need to run FastAPI.

As mentioned, we are using the `FROM` command to declare a base image to build from. Docker will default to the latest if you do not specify anything, but you can also specify a particular image tag. Specifying a tag can be useful for various reasons, especially if you want to lock the image to a particular version. We will use the `latest` for our images because we consider this a development project that would not be immediately pushed to production.

What do we mean by a development project? We just mean that in a production environment, more care is needed when changing the software version. You may have deprecated features, and there may be bugs or vulnerabilities that you should be aware of. These are all considerations that you need to weigh before changing the version. In this instance, we pick the latest version of Python when building our container. In a production environment, you would want to control that upgrade by running it through your test suite and ensuring no incompatibilities exist.

These concerns about the containers' version also extend to the application itself. This container installs FastAPI on top of the Python container, so we have a requirements.txt file to take care of that. We have chosen to just install the latest packages for this example; however, we encourage you to look at locking in a particular version and other requirements.

Python requirements.txt

Our example uses the package name without any type of versioning. However, there are better safety measures that you can take advantage of, such as

- *Specifying the exact version*—`package == 1.0.0`
- *A version range*—`package>=1.0.0,<2.0.0`
- *Exclude a version*—`package!=1.0.5`
- *A version combined with a Python version*—`package==1.1.0; python_version < '3.10'`

You can also install from version control systems, URLs, and local files.

(continued)

Finally, you can create a requirements.txt for your project by using `pip freeze > requirements.txt`.

You should be able to use the standard `docker-compose build` and `docker-compose up` commands to build and bring the containers online. Depending on the ports that you used (`8000` for us), you should be able to navigate to the APIs with http://localhost:8000/ and then see the output `{"message":"Hello World!"}`, which is still pretty cool although not doing much.

3.2.4 Web server container

The final piece of our container puzzle that we are looking into is a web interface. The plan is to provide a dashboard with which we can upload and view ACH files. Again, for now, we are just going to build a simple page to ensure everything is working. For this container, we will be building our application on top of Nginx. While a large number of existing infrastructures run Apache for a web Server, Nginx has had a slightly larger market share for the past few years.

This container builds on some of the concepts we explored before, such as building a container using a base image. In addition, you will likely want to manage the ports from the docker-compose file. Previously, we specified ports such as `8000:8000`, which gave both the host and container ports, respectively. By default, Nginx will be listening on `port 80`, and while we could update its configuration file to define a different port to listen on, we can take this opportunity to introduce configuring the ports in Docker. Suppose we want to be able to keep the default port for Nginx and just give a different port to the web browser. For instance, let us say we want to be able to navigate to http://localhost:3000/hello.html to access our sample web page. We can do that by specifying `3000:80` and then starting our containers.

At this point, we have set all the containers we are going to need so far, but we haven't ensured they are talking to each other.

3.3 Connecting our containers

Now, let's go back to getting our containers to communicate. We would like to make sure that our containers can interact. Our use case is simple to start with, but it lays the groundwork for expanding our project. We would like to be able to access a web page that uses the API layer to access our `health_check` database table. This table is not necessarily changing at this point, but we can manually adjust it and see the results. It will be a nice first step in ensuring that everything is functioning as expected.

As mentioned earlier, ports can pose a problem, not only from a security perspective but just remembering which ports you have chosen. So, let us make a list of our containers and the ports we are using.

Table 3.1 Container port listing

Container name	Host port	Internal port
CloudBeaver	8080	8080
Postgres	-	5432
Api	8000	8000
Web	3000	80

This information is also available in the docker-compose file, but sometimes it is easier to have a reference sheet. Note that, currently, we are not exposing a Host Port for Postgres. In the next section, we'll dig a little deeper to see why.

3.3.1 Connecting to the database

Our first step is to have the API container connected to the database and grab the result from the table. At this point, we are only looking to get the plumbing sorted out, so we are not going to look at writing a unit test yet. We will be able to easily test this ourselves as we build out the sample. First, we will focus on updating our API layer to connect to the database and return our given status.

We start by updating our requirements.txt to include `psycopg[c]` to support connecting to the database.

Listing 3.5 Our updated requirements.txt

```
psycopg[c]
fastapi
pydantic
uvicorn
```

The `psycopg[c]` is a production appropriate approach but also requires updating our Dockerfile to include additional dependencies. You should now be able to access http://localhost:8000/health/status to view the status. If everything went okay, you should see `[{"id":1,"status":"OK"}]`. If not, you can jump down to section 3.2.2, which discusses some common problems you may encounter.

We have a few things to think about now that we have updated this code. Our `health_check` table is currently built to have a sequential ID and a status, and we decided to return all our records in this example. You can experiment with this by going into CloudBeaver (http://localhost:8080), clicking the `SQL` command, and inserting another record with `INSERT INTO health_check VALUES(DEFAULT, 'testing')`. It is not necessary to capitalize the SQL statements, but we find that doing so helps keep the code more readable when dealing with SQL intermingled with other code. After inserting the record, you will find that navigating back to the health/status endpoint will return all records. So, you should see something similar to `[{"id":1,"status":"OK"},{"id":2,"status":"testing"}]`. Later, we can review and explore ways to address this. Keep in mind that we wanted to provide the health of the current system but that we are not doing anything to update that table at this time.

Exposed ports and internal networks

When specifying our connection string, we selected `port 5432`. You can also see that the `postgres` container is listening on `5432` as there should be a log message saying `listening on IPv4 address 0.0.0.0, port 5432`. However, our list of ports shows we did not specify it, so what gives?

Because we are using Docker Compose, one of the benefits is the creation of an internal network. This allows the containers to talk to each other as if they were on the same network. In our case, the above ports were only for the host system. If we wanted to write scripts or browse the database using tools installed on our desktop, then we could have exposed `port 5432`.

Why can we use CloudBeaver to view the SQL data? Well, that is because we exposed the `8080 port` for CloudBeaver which we can browse with our web browser. So, we are communicating externally with CloudBeaver, but CloudBeaver can use the internal network to hit `port 5432`.

3.3.2 *Troubleshooting PostgreSQL and FastAPI*

It is entirely likely that things did not go as expected when setting up your container or that things will not go as intended when setting up other containers. When starting the containers with `docker-compose up`, be sure to keep an eye on the logs as you may frequently see a meaningful error message. Also note that you may need to scroll back because an earlier container failed, and others are running after it, or an error/warning message was just lost in the shuffle. The following are some of the more common errors you may get in the scenario of a container not starting.

We are trying to make a connection from one container to another, so you may see a message on the console similar to `Is the server running on that host and accepting TCP/IP connections?` This message could mean that the container failed to start, and since it is connecting from our FastAPI code to the database, there may be a syntax error in our code that prevented the container from starting. This error could be a typo in our code or could result from dependencies that we failed to include. If the code is syntactically correct, check the requirements.txt and Dockerfile to ensure that all requirements are being installed.

The error `connection to server at "localhost" (::1), port 5432 failed: Cannot assign requested address` is another one we can run into. This error is common because many examples may refer to `localhost`, and it may be easy to mistakenly use localhost instead of the name/address of the container. You may also assume that since we are running on our computer, you can use `localhost`. Remember, though, that we are running in a container. We should specify the name of the container in our connection string rather than `localhost`.

The error `psycopg.OperationalError: connection failed: FATAL: password authentication failed for user "postgres"` can occur when you have an invalid user

or password specified. Look at the Dockerfile and docker-compose.yml to ensure you have the correct username and password. Remember that at this point, we have defined the username and password in our docker-compose file when setting up the database container, but we also had to hardcode it into our Python code.

Another error we may encounter, especially when generative AI is helping, is a `NameError`. A `NameError` is raised when a local or global name is not found, meaning we tried to use a variable, function, or module that has either not been defined or is not accessible in the current scope. We encountered the error `NameError: name 'status' is not defined` when working through the code ourselves because we were not paying close attention to the code generative AI suggested. The variable `status` was populated by Copilot even though that was not the name of the variable we used. So, this amounted to a simple syntax error, and we had to return the right value.

3.4 Calling our API

At this point, we should be able to confirm that the API has access to the database container. We have one more jump that we would like to make—we want to be able to create a web page that will access the API. Although we have already seen that we can access the API in our browser, we were able to get away with testing that way because our REST API was using a `GET` request, which is the same request used to retrieve web pages. Later, when using some of the other HTTP verbs such as `POST`, `PUT`, and `DELETE`, we will not be able to test in this fashion. Furthermore, we will be looking to build the UI, so we need to ensure connectivity between the web and API containers.

We will create a simple web page that incorporates d3.js to display our results. In our case, we do not need anything fancy, so we will just create a list of the results. If you are not familiar with HTML or d3.js, this would be a wonderful time to see whether our generative AI tool could help you get started.

We prompted ChatGPT (GPT-40) with

CK Create a simple d3.js webpage that calls an API.

It produced an example visible in `chat_gpt_example.html`.

Listing 3.6 Sample page produced by ChatGPT

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>D3.js API Call</title>
    <script src="https://d3js.org/d3.v6.min.js"></script>
</head>
<body>
    <h2>Users from JSONPlaceholder API</h2>
    <ul id="users"></ul>
```

```
    <script>
        // The URL for the API endpoint
        const apiUrl = 'https://jsonplaceholder.typicode.com/users';

        // Use D3 to fetch the data from the API
        d3.json(apiUrl).then(data => {
            // Once data is fetched, populate the list
            d3.select("#users")
                .selectAll("li")
                .data(data)
                .enter()
                .append("li")
                .text(d => `${d.name} (${d.email})`);
        });
    </script>
</body>
</html>
```

This response gives us a great starting point, and with a few tweaks, we have a workable solution for our initial needs.

Listing 3.7 Updated sample page

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Health Check</title>
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>

<body>
    <h2>Health Check</h2>
    <ul id="healthStatus"></ul>
    <script>
        // The URL for the API endpoint
        const apiUrl = 'http://localhost:8000/health/status';

        // Use D3 to fetch the data from the API
        d3.json(apiUrl).then(data => {
            // Once data is fetched, populate the list
            d3.select("#healthStatus")
                .selectAll("li")
                .data(data)
                .enter()
                .append("li")
                .text(d => `${d.id} (${d.status})`);
        });
    </script>
</body>
</html>
```

We start by giving our page a title and importing d3.js.

Creates a heading and updates the ID to be more meaningful for our purposes

Accesses the API endpoint for the status check

Updates the select statement and text fields (ID and text) so that we are writing out the expected data

In addition to customizing the output for our specific need, we had to bump up the version of d3.js to the current level. This is a reoccurring theme when working with generative AI; depending on the model and training data, the latest software may not always be included. It is a good idea to quickly double-check all software release levels just to confirm you are running an appropriate version.

The most important customization that we had to make was pointing the page to the correct API URL. You can refer to the port listing we made earlier; do you have any ideas what that URL will look like? You may have been tripped up by the previous example of connecting the database and API containers where we had to specify the container name. In that example, we were internal to Docker. Now, our web page is external, so we want to access it from the outside and thus use localhost with `port 8000`.

Testing access gives us a CORS (cross-origin resource sharing) error:

```
Access to fetch at 'http://localhost:8000/health/status' from origin
'http://localhost:3000' has been blocked by CORS policy: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
```

CORS

CORS, which is better known as the bane of getting things done quickly, is one of the more common problems you will run into. CORS stands for cross-origin resource sharing and is a security mechanism that allows servers to specify controls over which resources are shared with external callers.

CORS provides the idea of simple requests and preflight requests. We are not really concerned with simple requests because preflight requests are the ones that trigger CORS exceptions. To work around these exceptions, we will need to configure our server to ensure some specific headers dealing with CORS are sent back.

The following headers need to be used:

- `Access-Control-Allow-Origin`—Specifies which sites are allowed to access the resource. We can use a wildcard `*`, but this can pose a potential security risk. Remember, the principle of least privilege? Start with a more focused origin and only broaden it as necessary.
- `Access-Control-Allow-Methods`—Specifies which HTTP methods are allowed. Again, you can use wildcards but should only use what is necessary. In our example, we are only using `GET` so that is the only one we really need. Later, we will be using more requests, but we recommend keeping it as `GET` for now and not updating it to use HTTP methods we have not yet defined, so that CORS will complain again later as it provides good troubleshooting practice.
- `Access-Control-Allow-Headers`—Allows the server to specify what headers can be used for the request.
- `Access-Control-Allow-Credentials`—Indicates whether the browser should include credentials for requests. If you have authenticated requests or need to maintain state with cookies, or otherwise have requests that include sensitive information, set this to `true`.

To correct this error, our API layer needs to be updated with the following.

Listing 3.8 Incorporating CORS into our API

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
import psycopg

DATABASE_URL = "dbname=postgres user=postgres
➥ password=secret host=postgres port=5432"

app = FastAPI()

origins = [
    "http://localhost:3000",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=False,
    allow_methods=["GET"],
    allow_headers=["*"],
)

def get_db():
    conn = psycopg.connect(DATABASE_URL)
    return conn
```

We import the CORSMiddleware from fastapi.

The origins are where we expect to receive requests from; we create this as a separate variable because we expect to update it a few times.

We pass all the parameters here; allow_methods and allow_headers are an integral part of securing our API.

Notice how we tried to keep the origins and methods narrow. In other words, we limited the allowed origins and methods to only what we need for now. Resist the urge to future-proof the code by using wildcards and more methods than you currently have/need. This is because when it comes to security, we want to keep it locked down. We wanted to say "Less is more" but did not want anyone coming back to us saying that they were hacked because they had less security! We mean that less access is better!

With the CORS updates in place, you should be able to build the containers and bring Docker back up. Hopefully, you can now navigate to http://localhost:3000/health.html and see some results. You should be able to again go back into CloudBeaver and enter additional records in the `health_check` table, which will be displayed when you refresh the web page.

3.4.1 Troubleshooting our web site

The main problem you are likely to encounter is the error that we discussed before—the CORS error:

```
Access to fetch at 'http://localhost:8000/health/status' from origin
'http://localhost:3000' has been blocked by CORS policy: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
```

As previously mentioned, this security-related error is easy to work around if you choose to just allow everything. We would caution against this approach as you should maintain some security. Often, your APIs may have a level of redirection built in as they may go through other tools such as WSO2 API Manager (which we discuss in chapter 4). In this case, you may want to just specify that middle layer in your CORS configuration. Or, you may want to limit HTTP requests. Again, the idea is that if you do not need it, then you should not have it.

An `ERR_SSL_PROTOCOL_ERROR` may occur if you are using HTTPS and the API was only listening on HTTP. You may see an error on your web browser console such as `Failed to load resource: net::ERR_SSL_PROTOCOL_ERROR`. For now, we are using HTTP and not HTTPS because we did not want to introduce additional complications with certificates. However, we will work on expanding this later in our project.

You may encounter a `404 error` from d3.js such as `d3.v7.min.js:2 Uncaught (in promise) Error: 404 Not Found`. This is usually going to be a typo in your API endpoint. Another reason could be using the wrong HTTP method (`GET` when you need `POST`), although it is more common to see a `405 Method Not Allowed`. We have seen some frameworks use the `404` error code as well. Other problems also exist, but we start exceeding the scope of this book as they start getting into server and network configurations. So, you will need more assistance if you have eliminated these basic reasons.

We mention this here although it could happen at any time. If Docker is not running, then you may get errors building and starting your project. Seeing an error such as `error during connect: this error may indicate that the docker daemon is not running` is an obvious message that you do not have Docker running. However, we are likely to make that mistake at least a couple of times, so it is best to address it now.

3.5 Container security

Hopefully, you have an idea of how cool working with containers can be. Yes, our application has inherited some complexity but that is a tradeoff for some of the benefits of using this type of architecture.

We also inherit some security risk as part of this convenience. When we build a container using an image, even an official image can have risks associated with it. These come in the form of security vulnerabilities included in the package. It is important to understand these problems as they could be serious exploits in a version of the image or packages you are using. Some images are complex, and it may not be apparent what is actually included in a particular image. That is where tools such as Snyk and Docker Scout come into play. They can scan your containers and report on vulnerabilities in any of the images or included packages.

Snyk offers a command-line-scanning tool that creates an inventory of our Docker containers (figure 3.3). We can then use their web site to view scans and fix potential problems. When new problems are discovered in one of our containers, we will be automatically alerted so that we can begin fixing them immediately (if necessary).

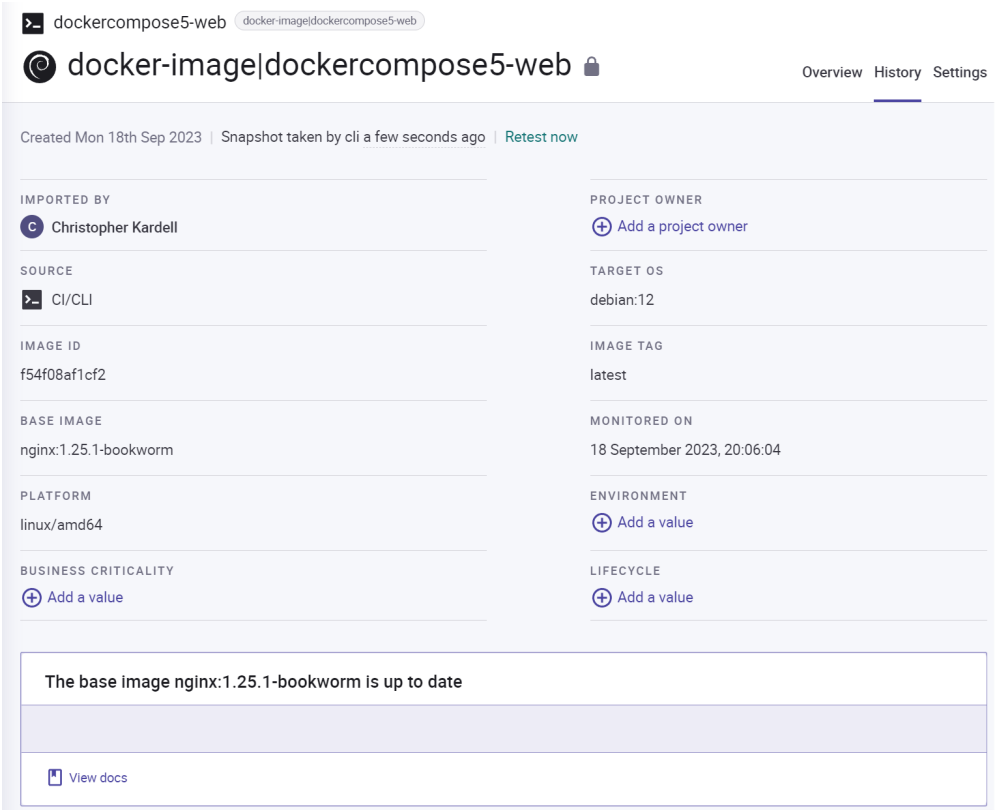


Figure 3.3 Sample results generated by Snyk

Another aspect to pay attention to is port numbers used both within the containers and exposed to the world. We can see ports available using `docker ps`. When configuring containers, it may be helpful to allow ports to be set dynamically. For instance, our `api host port` could be set with `${API_PORT:-8080}:80`, which would allow us to default to `port 8080` unless `API_PORT` has been defined, in which case that would take precedence.

There are numerous options within Docker to help secure containers, and we will briefly touch on some and their application:

- `--read-only`—It marks a container as read-only, which can help limit malicious activity in the event of an attacker gaining access to the container.
- `--memory` and `-cpu`—There are multiple configuration options around memory and CPU. These can configure how many resources are available to a container from the host system. Preventing a container from eating up too much of a resource is important to the overall health of the system.

- `--no-new-privileges`—It prevents the application from gaining any new privileges. This means Docker will prevent you from executing scripts and commands that would give you root privileges. The idea again is to keep malicious code and users contained.

The principles of least privilege and defense in depth

A common notion in security is known as the principle of least privilege (PoLP). We try to use minimal access or only the necessary permissions to complete our work, elevating those permissions only when necessary and then reverting to our original state. Ideally, even administrators or those with root access are not using it as their default sign-on. It is easier to grant users everything or higher permissions, but if their account is compromised, then the attacker has that same access. Perhaps, even more common is a user just making a typo or not paying attention to the system or directory they are in, and causing damage by removing unintended files. We have heard more than one story of a script being run as root that brought down the system. Or someone may be doing `rm *` in the wrong directory and crashing the system!

"Defense in depth" or "security in layers" is another security practice that you will be implementing, regardless of your role in an organization. It simply refers to the broad range of security practices put in place to protect an organization from physical security to network security to application security. As a developer, you are likely going to be involved in the application-level security. This will also expand into data security and access management, as you will need to encrypt and mask data, as well as authenticate and authorize users to the system.

If you are part of a large organization, the managing and security of containers may not be your area, especially if you are working in a software developer role. In a smaller company, you may be wearing multiple hats, and having some knowledge of these concepts will be useful. In our case, we are using containers to play around with our development project, and we just need to understand the CliffsNotes. However, security is a major consideration in all organizations, and understanding some of the basics will not only help you be a better developer but will also give you a different perspective when that next annoying request comes from the security team. If you want to explore Docker security further, check out the OWASP Docker Security Cheat Sheet (https://mng.bz/nR8K).

3.6 Optimizing Docker

Docker is great, especially if you are a developer. There are tons of Docker images that you can use to explore different technologies and tools without the need to install them on your machine (other than with the Docker image). So, you no longer need to clutter up your personal machine with a bunch of software that you later forget about. Of course, this convenience is not without some drawbacks. For instance, if you have started all the Docker projects bundled with this chapter, you may have a lot of wasted

space as Docker created containers, images, and volumes for each, which is why we are just going to take a little bit of a dive into managing and optimizing Docker.

So, why optimize? When we look at optimizing Docker, we are primarily speaking about removing unnecessary components and perhaps building them out from a different image. By reducing the components, we can usually achieve better security, faster deployment, and potentially run-time improvements. As developers, we may haphazardly pull in images with little regard for what they are built on. After all, we usually try to get to our work, but for now, let us see what things look like under the hood.

3.6.1 General maintenance

First, we run the command `docker image ls`. It will show us the images and their sizes. Here is a sample output:

```
REPOSITORY           TAG       IMAGE ID       CREATED          SIZE
dockercompose7-web   latest    9d0257a0f9b0   56 minutes ago   187MB
postgres             latest    e872544a957a   56 minutes ago   417MB
<none>               <none>    056294c7ea5d   58 minutes ago   187MB
<none>               <none>    531ebd9cf3db   58 minutes ago   417MB
dockercompose7-api   latest    5a2bb8fc1b02   24 hours ago     1.09GB
<none>               <none>    9552a83973a9   24 hours ago     417MB
<none>               <none>    606976f59240   24 hours ago     1.09GB
<none>               <none>    237b5dd9e45d   24 hours ago     417MB
<none>               <none>    20ec806118d6   24 hours ago     187MB
<none>               <none>    361454300c20   24 hours ago     1.09GB
```

The first thing you should notice is that we have a bunch of repositories with `<none>`. These are known as dangling images and can appear when we are building/rebuilding our project. We may have brought some of these on ourselves simply by working in our development environment and continually building our containers as we modify our project. Another common reason is using the `latest` tag. We had previously chosen to use the `latest` tag because we always wanted the latest project. Now we could be paying a price for that decision. In our current environment, we personally are not overly concerned about this, but in a production environment, where we had a specific project we were building, we would ensure we had tagged our project to a specific level.

To clean up these dangling images, we can issue the following `prune` command. With all the prerequisite warnings about being careful when removing things, we issue `docker image prune` and are prompted to confirm we want to continue. Then, several images are removed, and we then do `docker image ls` to see what is left:

```
REPOSITORY           TAG       IMAGE ID       CREATED             SIZE
dockercompose7-web   latest    9d0257a0f9b0   About an hour ago   187MB
postgres             latest    e872544a957a   About an hour ago   417MB
dockercompose7-api   latest    5a2bb8fc1b02   24 hours ago        1.09GB
dockercompose6-api   latest    d53ec48f5fec   3 days ago          1.09GB
dockercompose6-web   latest    ef65a4305bf7   3 days ago          187MB
cloudbeaver          latest    9b672f480fc7   2 weeks ago         250MB
```

Then, we take a look at what is left over, and we see we have previous images out there that we presumably do not care about (`dockercompose6-api` and `dockercompose6-web`). We can confirm they are not in use by any (either running or not) containers by issuing `docker ps -a --filter ancestor=dockercompose6-api:latest`. This code will return the status of the container. When nothing is returned, it can be safely deleted.

3.6.2 Optimizing image size

Now that we have done some general clean-up, it may be a little easier to review our images. We can start again by taking a look at the current images with `docker image ls`:

```
dockercompose7-web   latest    9d0257a0f9b0   About an hour ago   187MB
postgres             latest    e872544a957a   About an hour ago   417MB
dockercompose7-api   latest    5a2bb8fc1b02   24 hours ago        1.09GB
cloudbeaver          latest    9b672f480fc7   2 weeks ago         250MB
```

Our biggest image is the `dockercompose7-api` image at 1.09 GB.

One way we can reduce image size is checking whether the particular Docker image supports Alpine Linux. Alpine Linux is a lightweight version of Linux focusing on security, which makes it a good fit for use within Docker.

Postgres went from 417 MB to 239 MB just by switching the image, and the web went from 187 MB to 42.6 MB. Unfortunately, we cannot simply switch to Alpine for our biggest container, the API layer that currently sits at 1.09 GB. This is because we are running `apt-get` in our Dockerfile, and this is not supported by Alpine since it is not based on Debian. We can take a shortcut and use the `slim` tag. By specifying `slim` instead of `alpine`, we do not need to change our Dockerfile and end up with an image that is about half the size, at 557 MB.

Slim vs. alpine

So, what exactly is the difference between the `slim` and `alpine` tags? Both are typically smaller than a standard image, with `alpine` images typically being even smaller than `slim` images.

As the name implies, `slim` images are minimized versions of the standard images, removing nonessential components such as documentation and development libraries but leaving the core functionality intact. Remember, these containers are specifically designed for running whatever you have put in them. They are not meant for people to be logged in and cruising around, so we do not need things such as man pages for documentation.

In contrast, Alpine focuses on security, and it is based on the `musl libc` library and BusyBox. These can cause problems with other software and libraries that you may be installing. As we saw, we had to update our Dockerfile to take advantage of Alpine with some minor changes. Of course, for larger projects, more extensive testing would be needed to ensure everything is working.

We can look into reducing the image even further if we want to pursue using Alpine for the API layer. We will need to update the Dockerfile to use the Alpine Package Keeper (`apk`) instead of `apt-get` to install the needed packages. Your mileage may vary depending on your project and what exactly needs to be installed. In this case, removing the `RUN apt-get` line and replacing it with

```
RUN apk -U upgrade
RUN apk add -no-cache libpq-dev python3-dev gcc musl-dev
```

allowed us to successfully get the project up and running again. This reduced the image size for another 200 MB, bringing it down from 557 MB to 344 MB—quite a difference from the original 1GB!

Choosing a smaller base image in your Dockerfile is one of the most straightforward ways of reducing the image size. One of the mantras of development is "Make it work, make it right, make it fast." That approach works for containers as well. There are benefits of smaller image size, but you do not need to worry about that right at the beginning of your project. When working with a big company, there will likely be a whole team that takes care of this for you, so you may not have a lot of flexibility when it comes to building and running containers.

3.6.3 Optimizing build time

A task that can go hand in hand with optimizing image size is also working on the actual build time. Several tools are available for analyzing Docker images. Two popular tools are Docker Scout and dive, which is a command-line tool. Let us look at running the Docker Scout utility because it is readily available within Docker Desktop. This discussion also provides a good introduction to some of the metrics and terms that get thrown around when using Docker. Figure 3.4 shows the results of analyzing our API image. This screen is from the personal (free) tier of using Docker and Docker Scout. More features are available for the paid tiers (https://www.docker.com/pricing/).

The layers are what we are primarily interested in when considering ways to optimize the build. In a Dockerfile, the order of instructions matters, and each instruction translates into a layer, although there is not always a one-to-one correspondence. We can see the size of each layer and the command being executed. We can also see how the filesystem is affected by each of the commands.

One way to optimize the builds is to ensure we are managing the layers appropriately. Layers should be kept to a minimum, and any adding/removing of files should happen in the same layer. For instance, a command that may download temporary files should also be cleaned up.

We also mentioned that the order of instructions in a Dockerfile matters because Docker uses a caching mechanism to determine if a command needs to be rerun. Therefore, items that change frequently, such as your application code, are usually near the bottom of the instructions, while less frequent changes or changes that may cause a large download are executed near the top. Doing this in reverse order is known as

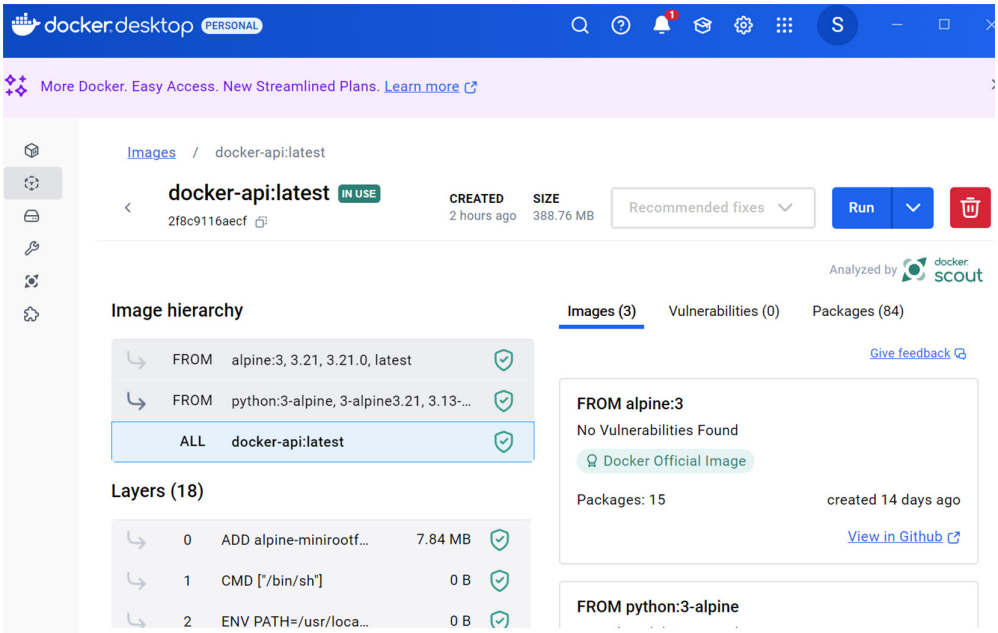


Figure 3.4 Analysis with Docker Scout

cache busting because it breaks Docker's cache mechanism and can cause Docker to rebuild the entire image every time.

Docker squash is another tool that can help with removing layers and other unused files. A few different versions of this utility and also an experimental command within Docker itself for this functionality can be found on the internet.

There is also a .dockerignore file that functions similar to a .gitignore file (if you are familiar with Git). Otherwise, the concept is simple: we can put files or directories into these ignore files, and Docker will ignore them. So, maybe unnecessary configuration files from your IDE, other application related information, and documentation such as READMEs are all good candidates for these files.

Finally, general application maintenance can go a long way, ensuring that dependencies are up to date and that you have minimized CSS, JavaScript, and so forth to keep the images small.

3.7 Removing the hardcoded username/password

Now that we have covered some of the maintenance aspects of working with Docker and Docker images, let us take our advice from earlier and work with the application. One thing that we had to do was specify a username and password for the PostgreSQL instance (remember the `POSTGRES_USER` and `POSTGRES_PASSWORD` from our docker-compose file).

We were able to do this with environment variables; however, when it came to the API container, we hardcoded the values, and that is something we always want to avoid if possible.

> **Avoid magic numbers and hardcoded values**
>
> Magic numbers have unexplained meanings, such as when we define a log level of 1, 2, 3, 4, or 5. Which one is the highest severity? What does it mean to have a log level of 1 versus a log level of 4? That is an example of a magic number.
>
> We consider hardcoded values to be a superset of magic numbers as they come with a similar set of problems. Any time we start using hardcoded values such as usernames/passwords, ports, servers, and similar, it poses a problem. Not only can it be a security concern to have them stored insecurely in source control but can also pose a maintenance nightmare when they are sprinkled throughout the code, and your server moves after 10 years. Finding all those places and testing that everything was changed correctly can be a real pain.
>
> Even though we have yet to examine the application, we will practice good habits by removing our hardcoded values.

Luckily, Docker comes with a way for us to make use of environment variables by applying an `env_file` directive that specifies an environment file. As a bonus, we can refer to the `gitignore` command to prevent this file from being uploaded to source control. Or maybe there is another layer of redirection where the more sensitive information (the database credentials) is referenced and stored somewhere else. If you are not storing the environment variables in source control, be sure that they are well documented somewhere so that new instances, or new employees, know how to set them up. Often, we have found ourselves having someone email us random configuration files because nobody is sure how to set up the system and what exactly is still needed!

The first order of business is to create a file to hold our environment variables. We will need to create a file named .sql_server.conf. The actual name of the file is not important but should be something meaningful. The leading period (.) will often cause the operating system to hide the file and an extension "conf" is a standard for configuration files. Another common naming standard is to use .env. We often see .env files being used in companies, especially when there are source control tools such as Git, which are usually set to ignore .env files. Another example form the Docker documentation available at https://mng.bz/vK5M shows the use of an environment file named webapp.env when defining their webapp service.

The main thing to remember is that the naming of the file is flexible, and neither a specific name nor leading period are required for them to work. However, we are simply establishing a convention, and that may be slightly different from the standards and conventions in place at other companies (remember the saying, "When in Rome . . .").

With the file created, we will move our hardcoded username and password denoted by `POSTGRES_USER` and `POSTGRESS_PASSWORD`, respectively, ending with a file that contains

```
POSTGRES_USER=postgres
POSTGRES_PASSWORD=secret
```

We will also need to update the `docker-compose.yml` to make use of this newly created file. Note that we removed environment and replaced it with `env_file`:

```
  postgres:
    build:
      context: ./db
      dockerfile: Dockerfile
    env_file:
      - ./.sql_server.conf
```

Similarly, we also made a change to the API container to employ the same `.sql_server.conf` in the `env_file` directive. Before going further, we should build and bring our containers up to ensure that everything is still working. Finally, we will need to update our API container to also pull in these variables. We import the `os` module and then use `getenv` to retrieve the environment variables that were set in our .sql_server.conf file:

```
import os
…
POSTGRES_USER = os.getenv("POSTGRES_USER")
POSTGRES_PASSWORD = os.getenv("POSTGRES_PASSWORD")
…
DATABASE_URL = f"dbname={POSTGRES_USER} user={POSTGRES_USER}
➥ password={POSTGRES_PASSWORD} host=postgres port=5432"
```

With those changes, we can remove, build, and bring up the containers to test. You may also want to change the values in the environment to ensure that the changes are indeed working. In addition, do not forget to sign into CloudBeaver to ensure that is working as well.

The `os.getenv` command also allows you to specify a default value if the `environment` variable is not found. So, we could have used `os.getenv("POSTGRES_USER", "postgres")`, but we stayed away from a default value in this example because we wanted to ensure that our containers would not work accidentally if we had set something up incorrectly (because the Postgres user was still valid somehow). Your use cases will vary, so just keep that functionality in mind.

We also updated the `DATABASE_URL`. Substituting the `POSTGRES_USER` for both the `dbname` and `user` in the connection string. This is just a convention in the PostgreSQL database. The `dbname` and `user` are independent of each other, but we kept our SQL script straightforward, so the table was created under the same username. You will also notice that the connection string contains hardcoded values for the host and port. We are leaving it up to you whether you would like to replace those with an environment value that will not hurt anything and may also be an opportunity for you to try out the default value in the `os.getenv` mentioned earlier.

3.8 Health checks

In our sample, we built out a simple health check endpoint. This endpoint is something that applications could use to get a status of the system or application. It can be as fine-grained as you want, checking not just the services that it may be running but also potential problems connecting to other services.

Here we would like to take a moment to look at health checks from a purely Docker perspective as opposed to what may be done at the application level. These health checks can be used to determine whether the container is running correctly (or healthy). Containers can be in multiple statuses such as starting, healthy, or unhealthy, and the importance of knowing the state of a container is one of the benefits of using containers. Unhealthy containers will adversely affect your system and could be the result of bad application code, networking problems, or perhaps a bad update in one of the components being used in the container. Docker and Docker Compose can automatically restart containers in an attempt to resolve potential problems, or you may build a more robust logging and notification system to monitor your containers.

We can determine the health of our containers using the `docker ps` command. You can run the command now; however, unless the image you are using includes a health check, you will not see starting/healthy/unhealthy next to the status; you will only see the up time. To save space, we will format the output of the `docker ps` command with the `format` argument. Running `docker ps --format '{{.Names}} - {{.Status}}'` from the command line shows

```
dockercompose9-web-1 - Up 5 minutes
dockercompose9-api-1 - Up 5 minutes
dockercompose9-postgres-1 - Up 5 minutes
dockercompose9-cloudbeaver-1 - Up 5 minutes
```

First, let's add a health check for the Postgres container. We will update the Dockerfile with

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=10s --retries=3 \
  CMD pg_isready -U ${POSTGRES_USER} || exit 1
```

Notice that the command takes several parameters and that the actual work is done by the `CMD`. The `pg_isready` is a Postgres command that confirms the database is up and running. It requires a user, so we are referencing the `environment` variable we defined.

If we rebuild following the usual procedure of removing the containers, rebuilding, and bringing them back up, then running that `docker ps` command gives us

```
dockercompose9-web-1 - Up 11 minutes
dockercompose9-api-1 - Up 11 minutes
dockercompose9-postgres-1 - Up 11 minutes (healthy)
dockercompose9-cloudbeaver-1 - Up 11 minutes
```

We can see that the Postgres container now shows as healthy. Including health checks helps us prepare our containers for a production environment where a container orchestration tool such as Docker Swarm or Kubernetes will enable our containers to be self-healing, meaning that the containers will be able to be restarted or replaced if they fail these checks. That is a bit beyond the scope of this book though. For the time being, keep in mind the power that health checks provide and the potential difference between a health check at the application level versus health checks from the Docker perspective.

Summary

- Docker and Docker Compose can manage our environment. By building out small pieces of a project into separate containers, we can start expanding each component of our project independently.
- Evaluating team experience, project size, and risk factors is vital when determining the best approach to a project.
- Generative AI tools can increase productivity but have limitations and require verification.
- Docker creates isolated environments for efficiently managing multiple services and technologies.
- Transitioning to microservices should be based on need, not trend, as monolithic architectures can still be effective.
- Dividing a project into manageable components and ensuring initial connectivity is crucial for effective development.
- Health checks in containers provide system status monitoring and enable self-healing in production environments.
- Security and efficient management are achieved by applying the principle of least privilege and using vulnerability scanning tools.
- Container optimization is important in reducing image size and build time.
- Environment variables should be used to avoid hardcoding sensitive information such as usernames and passwords.
- Adhering to best practices in storage, application optimization, and dependency management enhances Docker's effectiveness.

4 APIs: The piping between our components

This chapter covers

- An introduction to the FastAPI framework
- Documenting APIs using the OpenAPI standard
- Testing our APIs using Postman and Locust
- API management with WSO2

In this sprint, we work on another research spike where we explore designing and testing application program interfaces (APIs). An API is simply a way to interact with a software component. Regardless of whether we are dealing with microservices and distributed systems or using packages from `npm`/`pip`, we utilize APIs.

4.1 API-driven design

Generally, when we mention APIs in this book, we are talking about the REST (Representational State Transfer) API standard. However, there are other standards as well, such as GraphQL and the older SOAP (Simple Object Access Protocol). We may also come across message queuing, such as IBM Websphere MQ and RabbitMQ.

Table 4.1 shows some pros and cons of the more common approaches to implementing APIs.

Table 4.1 Pros and cons of different APIs

	REST	SOAP	GraphQL
Pros	Simple; builds on standard HTTP methods Multiple data formats can be supported, although JSON is the most common.	Relies on XML-based messaging Enterprise-level security	Efficient Single endpoint Strongly typed
Cons	Versioning and breaking changes can be complex to manage. The number of endpoints may become inconvenient.	Complex to implement Wordy XML-based messages	Steeper learning curve

APIs act as a contract parties agree to abide by. The consumer of the API agrees to abide by any terms of use and licensing agreements and follow any documentation and standards established by the provider. In turn, the API provider provides a stable and scalable endpoint to access. Subscriptions to APIs also provide tools to make costs, usage, and a Service Level Agreement (SLA) transparent.

With the adoption of distributed computing, many developers and architects have become interested in API-driven design. Often, we can expect our software to be deployed in the cloud or in a containerized environment. In these cases, we want to make sure we consider APIs. As with most design objectives, we want to keep some of the following things in mind:

- *Clear use cases*—An API should not serve as a Swiss Army knife that can deal with customers, accounts, and transactions. Instead, the design should be kept simple, straightforward, and consistent.
- *Versioning*—Anticipate versioning and having to deprecate older API versions. Having to maintain multiple versions of an API can lead to increased technical debt with each change. Tools such as WSO2 API Manager and the Anypoint Platform by Mulesoft can help with versioning, governance, and many other tasks regarding APIs.
- *Documentation*—Comprehensive up-to-date documentation ensures easy consumption of our API. Standards such as OpenAPI allow our APIs to produce documentation that can be made available online with tools such as Swagger and Redoc.
- *Performance and scaling*—Our APIs should be able to scale to larger datasets, which means that we may need to introduce paging for very large datasets. We also need to determine acceptable wait times for API requests and how to handle longer-running requests.

- *Security and validation*—Remember that a UI that validates user input is only half the battle. We need to ensure that the server is validating input, as well to prevent hackers and bad actors from exploiting our APIs.
- *Monitoring and analytics*—Ensuring our APIs are functioning when accessed by end-users and have sufficient capacity can only be done with appropriate monitoring and logging. We may want to consider OpenTelemetry and tools such as Prometheus and Grafana to support the observability and monitoring of our APIs.

There are other considerations as well, and APIs is a topic whole books have been written about. At this point, we just want to provide you with enough knowledge to make you more proficient in your software development role. If you are interested in additional information on RESTful APIs, visit https://restfulapi.net/. If you are looking for a deeper understanding of APIs and additional hands-on exercise, *The Design of Web APIs* (2019, Manning) by Arnaud Lauret is an excellent resource.

4.2 Our APIs

Before developing the endpoints for our APIs, we think about the tasks we need to perform. At this point, we won't be dealing with many complicated tasks, but only upload files, view files, and view records. Table 4.2 shows what our APIs may look like. Note that `:fileId` and `:recordId` are placeholders for identifiers that will be returned by other queries.

Table 4.2 API overview

HTTP method	Uniform resource identifier	Description
`POST`	`/files`	Uploads a file to the system
`GET`	`/files`	Returns a list of uploaded files
`GET`	`/files/:fileId`	Returns details on a specific file
`GET`	`/files/:fileId/records`	Returns records for a specific file
`GET`	`/files/:fileId/records/:recordId`	Returns details for a specific record

While the table provides a good overview of our APIs, it may be helpful to think in terms of the flow of these APIs and how to obtain these identifiers. Figure 4.1 shows the interaction between a user and the API.

PlantUML

PlantUML (https://plantuml.com/) is a fantastic tool for creating many kinds of UML diagrams such as sequence, use case, state diagrams, and more, as well as other diagrams for UI mockups, Gantt diagrams, and WBS diagrams.

PlantUML is a great tool to use for many of these diagrams because it is a text-based interface. We can produce diagrams just by entering text into an editor! That

means they are easy to update, easy to store in version control, and can be compared against previous versions. There are plugins available for our IDE as well, and we often use this tool from within VS Code.

Throughout the book, we use diagrams produced with PlantUML.

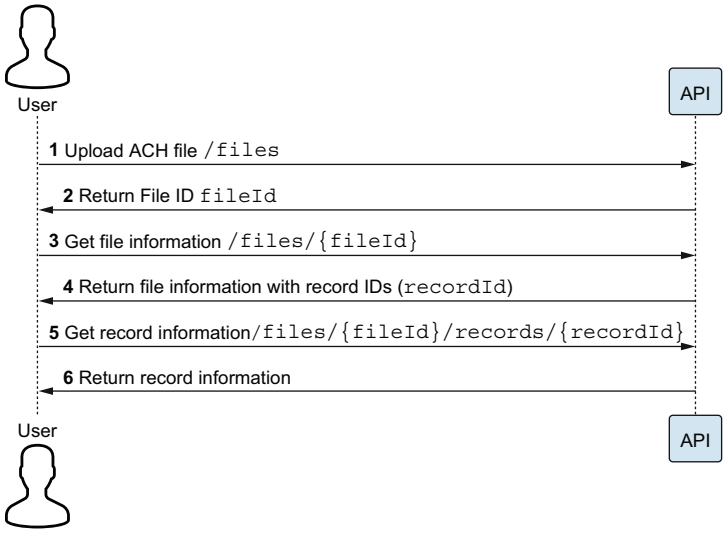


Figure 4.1 Sequence diagram produced by PlantUML

Hopefully, you can better understand now how a user may employ these initial APIs. These APIs should provide us with a good starting point for the project as they meet the requirements regarding uploading and viewing data. We can expand on these APIs by including query parameters to help retrieve specific data. For example, we may want to select files by date/time or select all batch header records from a specific file.

4.3 Building our APIs with FastAPI

Previously, we built a simple health check API using FastAPI, but now we will examine FastAPI in more depth and start building our APIs to deal with ACH processing. Let's take a look at building of our `GET` APIs first as they should be relatively simple to provide stubs for. In other words, we do not need to have a database yet. We can simply return a hardcoded value for now to make sure everything is working as expected.

While this approach will give you some basic understanding of how FastAPI works, it does not mean that we won't worry about topics such as pagination and asynchronous processing, which are required in a full-fledged production system to have a functional API. For now, keep in mind that returning large results such as thousands of ACH

records not only takes time (resulting in the user waiting for the data), but presenting data on a UI becomes difficult. That is where the concept of allowing paging becomes useful. While UI components can do this with a large dataset, it will also become necessary to allow the API to handle requesting portions of the data. Likewise, uploading large ACH files may take time to process, and the user will be unsure whether the process was successful; furthermore, chances are the request will timeout from the user side. Using asynchronous processing, where a response is sent to let the user know the file was successfully uploaded and where further processing is done to load the file, becomes paramount. We will touch on this again in chapter 12.

For now, we start with a simple project that has the following folder structure:

```
ApiSample/
├─ docs/
├─ app/
│  ├─ routers/
│  │  ├─ files.py
│  ├─ __init__.py
│  ├─ main.py
├─ tests/
```

We make sure to install the following requirements for our IDE, which should be maintained in a requirements.txt like when using Docker. However, if we have simply dived in and started installing Python packages, then IDEs such as JetBrains Pycharm allow us to create a requirements.txt file based on the requirements for our project. In this sprint, we focus on gaining experience with APIs and FastAPI, so installing packages into the IDE works well for now:

- `pip install fastapi`
- `pip install pytest`
- `pip install httpx`

We keep the routers in a separate folder, which makes it easier to manage our application as it grows. We could also refer to the YAGNI principle and keep things simple by not using a router yet, but for time and space constraints, we are going to jump ahead slightly. The routers/files.py holds the API endpoints we will build in the upcoming sections. For now, we only focus on the basics, as shown in the following listing.

Listing 4.1 Basic FastAPI router

```
from fastapi import APIRouter

router = APIRouter()
```

Imports APIRouter from fastapi, needed to define URIs for the application

Creates a router that will be used to define specific URIs

We then have the main.py that will pull in the router. The next listing shows how the router is incorporated into the application.

Listing 4.2 FastAPI entry point

```
from fastapi import FastAPI
from .routers import files

app = FastAPI()

app.include_router(files.router)
```

Imports the fastapi class needed to create the application and our previously defined routes

Defines the application and assigns it to the app

Includes our routes in the application

With this basic structure, we can now look at adding some routes!

4.3.1 Retrieving files with GET /files

Our first route should return a list of files (and eventually some information from the file header record). For now, we are going to hardcode a response. Because we have already defined the router, we just need to add the following code.

Listing 4.3 Our first API route

```
@router.get("/files/")
async def read_files():
    return [{"file": "File_1"}, {"file": "File_2"}]
```

A decorator that defines the route /files/

An asynchronous function that returns a hardcoded response for testing

Here we defined the route with `@router.get("/files")` and an asynchronous function called `read_files` that returns a hardcoded JSON array of files.

To check whether this code works, let's write a `pytest`, as shown in the following listing.

Listing 4.4 Testing our first endpoint

```
from fastapi import FastAPI
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_read_files():
    response = client.get("/files")
    assert response.status_code == 200
    assert response.json() ==
➥[{"file": "File_1"}, {"file": "File_2"}]
```

Necessary imports for our test

Defines the TestClient

Defines a test to validate the /files endpoint

Makes a request to the /files endpoint

Asserts the response code is 200 and that the response is as expected

Here we import `TestClient` and our app. Next, we define a client we can use to make requests. By calling the `get("/files")`, we make a `GET` request to the URL `/files`, and that should be a valid endpoint. We assert that we received the HTTP status of

`200`. Then we have our hardcoded JSON string. This should match whatever we were returning from the endpoint in main.py. Even if running this test worked flawlessly, we still might have doubts whether it is running or validating our data properly. This can sometimes be a common concern when we first start using unit tests. We can play around with changing the status code we are checking (make it a value other than `200`), the JSON string, the endpoint from `"files"`, or the request type from `get`. Changing those values should cause the test to fail and verify that the test is validating our logic.

Types of tests

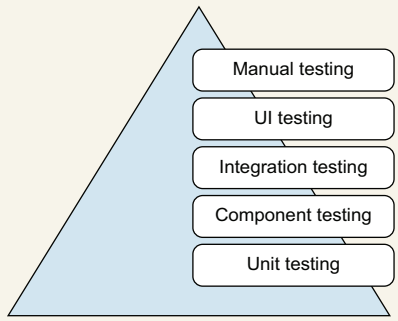
Being able to run unit tests can be of great benefit to our development cycle. Unit tests give us a starting point to execute the code and go through the entire process—from start to finish. But what type of test is this: Is it a unit test, integration test, or something else?

We hope that you agree with the statement that developers should be testing their code. Different systems, conditions, and code bases can make testing difficult, but they should be released without some sort of testing on our end only in extremely rare circumstances.

In most cases, developers will be primarily concerned with *unit tests*. The following figure shows that unit testing makes up the base of the testing pyramid, and this is where we will spend a lot of time creating tests, especially if we are following a test-driven development approach.

Component testing may be where the previous test fits. It typically isolates a component such as an API so that we can evaluate it in isolation, which may help us catch more of the edge cases with malformed requests or other exception conditions.

Integration tests typically test multiple components and involve more dependencies than unit or component testing. As such, they are primarily concerned with ensuring the components can work together.



A typical testing pyramid

UI testing is the next layer. It shouldn't be surprising that validating the UI using automated tools to crawl our UI validating functionality is similar to a user working through various tasks within the application.

At the top of the pyramid, we have *manual and exploratory testing*. This is where we (or the QA team) are trying to find ways to break the software or look for inconsistencies in the interface and behavior.

We have just briefly touched on a few of the major testing categories here. There are additional types of testing and other ways to break it down, which we will cover later.

One final thing to note is that the tests become less isolated, slower, and more fragile as we move up the pyramid. We use the term "fragile" to indicate that the test gets

harder to automate and is more likely to break because of its continually changing nature.

The test category is important only when it comes to the amount of setup and time it may take to get the test to work. Tests near the bottom of the pyramid are usually isolated and run quickly. So, at the moment, it is easy to run this test as a unit test because the API is returning a hardcoded value. Once we are further along and need to retrieve data from the database or other APIs, and components start coming into play, this test may not be appropriate to run as part of our development test suite but rather as part of a continuous integration and delivery (CI/CD) pipeline.

While we believe it is useful to have these tests function as unit tests, the reality is we may want to work this as a BDD-style test since they could be considered higher-level tests.

After ensuring that `pytest-bdd` is installed (`pip install pytest-bdd`), we create a `features` folder and a `step_defs` folder underneath our tests directory. Our folder structure for the tests folder should now look something like

```
tests/
├─ features/
|   ├─ files.feature
├─ step_defs/
|   ├─ test_files_feature.py
```

The `files.feature` will store our scenarios so we can work with the `/files` endpoint. Remember that we are defining the feature and various scenarios in this file using a `Given/When/Then` format. The following listing is a simple example of ensuring the status code.

Listing 4.5 Feature file for our BDD test

```
Feature: API /files

  Scenario: I get a 200 status code when we call /files
    When I make a GET request to the endpoint /files
    Then the status code should be 200
```

Next, to power the feature file, we create corresponding step definitions in our test_files_feature.py.

Listing 4.6 Step definitions for our BDD test

```
import pytest
from pytest_bdd import scenarios, when, then, parsers
from fastapi.testclient import TestClient
from app.main import app
```

Imports what is needed for testing

```
client = TestClient(app)

scenarios("../features/files.feature")

@pytest.fixture
def api_response():
    return {}

@when(parsers.parse('I {http_request_type} the endpoint {endpoint}'))
def when_http_request_to_endpoint(
    api_response,
    http_request_type,
    endpoint
):
    if http_request_type == "GET":
        response = client.get(endpoint)
    else:
        raise NotImplementedError
    api_response["data"] = response

@then(parsers.parse("the status code should be {status_code:d}"))
def then_the_status_code_is(api_response, status_code):
    assert (
        api_response["data"].status_code == status_code
    ), f"Expected {status_code}, but got
{when_http_request_to_endpoint.status_code}"
```

Loads the feature we defined in listing 4.5

Creates a fixture to store responses from the API; the hash will be empty.

Decorates the function with "when" to handle the condition from our feature file. Note that the endpoint is pulled from the feature file and will be /files.

At the moment, we are only handling GET requests and will raise an exception for anything else. We will revist this line to add more functionality as needed. Notice how the response from the API is stored in our api_response fixture.

Checks whether the status code was what we expected, which the feature file specified should be 200

Decorates the function with "then" to handle the "then" phrase from our feature file. Note that the status_code which we specified in the feature file is substituted for status_code.

Here we loaded the `files.feature` and defined functions to handle the `when` and `then` of our feature file. In addition, we have a fixture to store the API response for use between the calls. Notice, that although we coded the `when` to accept multiple types of HTTP requests, we only support a `GET` request at this point. As we create a more robust API that incorporates `POST`, `PUT`, and `DELETE` requests, we will need to expand it, or we will receive a `NotImplementedError`.

Again, we may want to experiment with the scenario of changing the `GET` to another request just to see the `NotImplementedError`, and then change the status code or endpoint to get a feeling for different scenarios we may run into when coding.

4.3.2 Adding a file with POST /files

We now expand our example with a `POST` request to the same `/files` endpoint. This approach could potentially be used to upload our ACH file. Since we have an existing endpoint and tests established, let's look at how we might work on this task from a more behavior-driven development approach. When working with a `POST` request we

usually see a `201 status` code, which means "created." Since we are uploading a file and expecting it to be stored on the server, a `201 CREATED` response should work for us. Thus, we can update our feature file with the scenario shown in the following listing.

Listing 4.7 BDD test for our `POST` request

```
Scenario: I get a 201 status code when we post to /files
  When I make a POST request to the endpoint /files
  Then the status code should be 201
```

We are reusing the same feature file here, so we do not need to adjust any of the "when" or "then" decorators.

When we run this test, we should first receive the `NotImplementedError` since we have not handled our client making a `POST` request, which we address now with the following code.

Listing 4.8 Updating our steps to support the `POST` request

```
if http_request_type == "GET":
    response = client.get(endpoint)
elif http_request_type == "POST":
    response = client.post(endpoint, json={"file": "File_4"})
else:
    raise NotImplementedError
api_response["data"] = response
```

Handling of the previous request remains the same.

We now add a way to handle POST requests.

Continues to alert developers when they try to do something we have not yet coded for

Now that we are making a `POST` request, we should see an error returned from our server because we have not actually handled a `POST` request. Note that in the previous code, we have a hardcoded post value being used for the request. Eventually, we will update the test to make this dynamic, but for now, we just want to get the basic setup. Running our test again results in the test failing because we receive a `405 METHOD NOT ALLOWED` status:

```
Expected :201
Actual   :405
```

This makes perfect sense because we have not actually coded the server to handle our `POST` request. Let's jump over to the `app/routers/files.py` and implement it (feel free to give it a shot before moving on).

Listing 4.9 Defining our `POST` endpoint

```
from fastapi import APIRouter, Request
...
```

Updates the import to include the Request object

Previous endpoint implementation

```
@router.post("/files/")
async def create_file(request: Request):
    return {"file": "File_3"}
```

Creates a function decorated with the POST verb and the URI, which is going to return a hardcoded value for now

Rerunning the test still fails, but this time because we are returning a `200` code.

```
Expected :201
Actual   :200
```

This is technically fine since we have not created anything. However, let's look at how we can update the status code because we will be creating data in the future. We update our files.py to include an import of status and update the `@router.post` as shown in the following listing.

Listing 4.10 Supporting the `201 status`

We import that status object from fastapi.

```
from fastapi import APIRouter, Request, status
...
@router.post("/files/", status_code=status.HTTP_201_CREATED)
```

Previous code

We update the decorator to specify status_code along with a 201.

Setting the `status_code` will return the `201` for all cases, and we can now rerun our test and see that it passes. Note that setting the status code this way will cause this endpoint to always return a `201`. Later, as we expand our APIs, we will explore other ways of setting that status code so that it allows us to have finer control over them. This finer control will allow us to specify additional status codes when errors are encountered in our APIs.

4.3.3 Finishing the rest of the APIs

In the previous section, we created two APIs to support getting a hardcoded list of files and posting a file. You should have enough knowledge of the basics to create the remaining endpoints and unit tests on your own. Take a few minutes to create some unit tests for the remaining endpoints and then code up the remaining APIs to get those tests to pass.

4.4 Adding our API to Docker

Fantastic! Now that we have built our endpoints, we can run these APIs in a Docker container. We gained a lot of experience with Docker in chapter 2, so we can use that here. Essentially, we just need the API container from the previous chapter updated with the code we did here. Let's see how we can get this code up and running in Docker.

4.4.1 Preparing the project

We have tested our code and should feel comfortable deploying it. Having faith in our suite of tests is needed for continuous integration and delivery. If you are not comfortable with deploying code after it passes all your tests, ask yourself why. Then figure out what additional steps and safeguards you need. We may need additional tests to cover edge cases, load testing, better coverage, static code analysis, and similar. The idea of automated tests is to help improve software delivery time and quality.

So, we should be ready to deploy the code. However, we were previously using Uvicorn to run the FastAPI application. Remember that Uvicorn is ASGI (Asynchronous Server Gateway Interface) compliant and goes hand in hand with FastAPI. We need to include it in our current project for completeness' sake. By installing it in our project, we will be able to generate a list of requirements for the project. So, ensure it is installed by using `pip install uvicorn`.

After adding that additional dependency, we want to create a list of requirements that will be installed on the container. Remember that we had previously tried to keep the size of our Docker images to a minimum; furthermore, we should never only assume that a required package for our project exists on the server. Some IDEs will provide options to generate and manage the requirements.txt (the standard naming convention for Python requirements). However, if our IDE does not, we can always generate it from the command line with `pip freeze > requirements.txt`. Once we do that, we can review the requirements to ensure we have our desired packages. For instance, we may want to ensure that `uvicorn` and `fastapi` are present. Note that we may see additional requirements because of other dependencies we have installed.

With the requirements.txt and application folder in hand, we should have everything we need to run this in Docker.

4.4.2 Docker setup

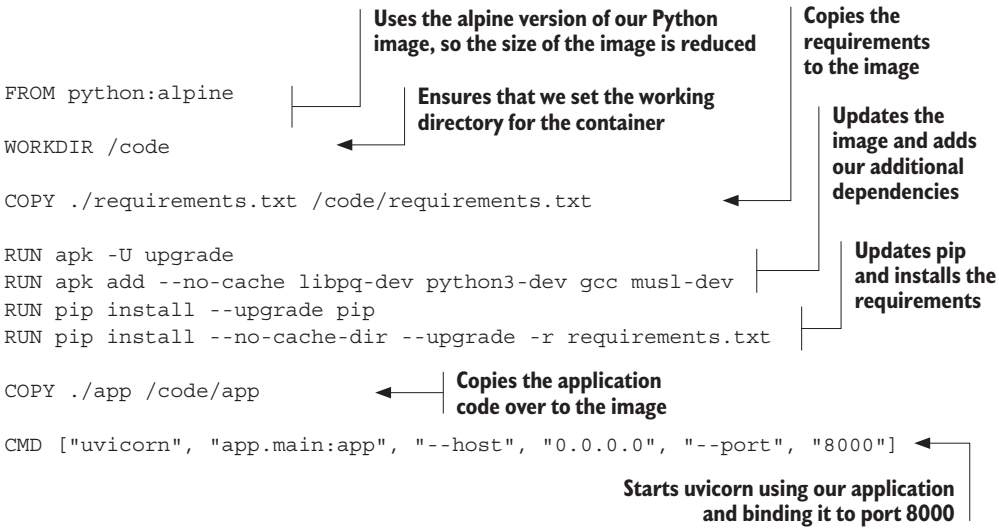
Using our previous Docker experience (we could just copy a previous setup we have or start from scratch), we end up with a folder structure similar to

```
api/
├─ app/
├─ Dockerfile
├─ requirements.txt
docker-compose.yml
```

where the app folder will contain our application code. At the moment, we only need a few source files: main.py, files.py, and the associated __init__.py. Although it is quite possible there may be additional files present specific to the IDE, there is no need to copy those over.

The Dockerfile is where all the heavy lifting comes into play, but since our structure is the same, we do not need to change anything.

Listing 4.11 Our Dockerfile




```
FROM python:alpine

WORKDIR /code

COPY ./requirements.txt /code/requirements.txt

RUN apk -U upgrade
RUN apk add --no-cache libpq-dev python3-dev gcc musl-dev
RUN pip install --upgrade pip
RUN pip install --no-cache-dir --upgrade -r requirements.txt

COPY ./app /code/app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Next, we want the docker-compose.yml file to manage the building of the service for us. We will only have a single service at this point:

```
services:
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    ports:
      - 8000:8000
```

Once these are in place, we should be able to run the usual `docker-compose build` and `docker-compose up` commands to start our container. After navigating our browser over to http://localhost:8000/files/, we should see the hardcoded value returned. In our case, it is `[{"file":"File_1"},{"file":"File_2"}]`.

4.5 Documenting our API

Great! We are moving right along. While most of our API is hardcoded, we will continue to refine it as we move forward. However, the hardcoded values are only a part of what makes our API unusable. Our main problem is that nobody knows about it, and if they knew about it, how would they use it? We need documentation, and OpenAPI (https://www.openapis.org/) comes to our rescue. This is a standard that allows us to create interactive documentation, and others to discover how the API works and how it generates client code and server code, among other things.

Remember API-driven design from the beginning of the chapter? Well, it is possible to give an OpenAPI specification to generate not only client code that would consume

that API, but also the beginnings of the server-side code that will process those requests. With our Docker container up and running, we should have support out of the box for two flavors of documentation based on the OpenAPI specification. We will cover both documentation frameworks in the following sections.

4.5.1 Swagger

Swagger is a set of tools for implementing the OpenAPI definition, but for historical reasons, many people may still refer to the OpenAPI definition as a Swagger specification, Swagger definition, or just Swagger. One of the most useful tools that you will likely find yourself using is Swagger Editor (https://editor.swagger.io/), which provides a side-by-side view of the OpenAPI definition and the resulting output. We can browse the Swagger documentation generated by FastAPI by navigating over to http://localhost:8000/docs. From there, we can view the documentation and experiment with the API. In this case, though, we want to click the link that should take us to http://localhost:8000/openapi.json and display the OpenAPI JSON. We can save or just copy and paste it into the online Swagger Editor. Figure 4.2 is a sample screenshot of what we see.

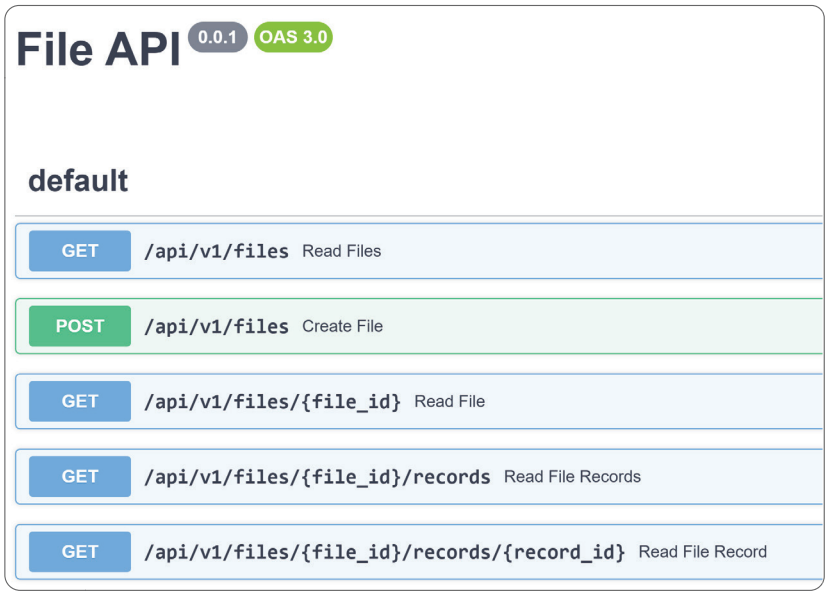


Figure 4.2 Online Swagger Editor

You can play around with editing the API to gain a better understanding. Note that the online editor will prompt us to convert the JSON to YAML. We do not have to do this, but it does make it a little easier to edit in place.

The FastAPI framework supports several ways to help customize this documentation further, and we will be looking at that later on.

4.5.2 Redoc

FastAPI also supports Redoc, a free and open source tool for creating documentation. It provides a responsive three-panel view that has become popular, especially when working with larger APIs. Again, FastAPI provides this documentation out of the box, and we can see it by going to http://localhost:8000/redoc. Figure 4.3 shows our sample API documentation.

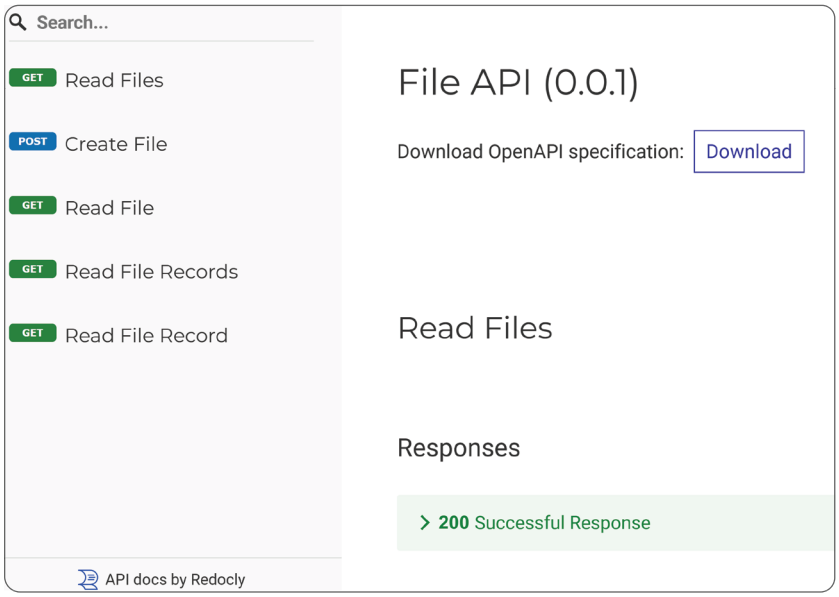


Figure 4.3 FastAPI documentation with Redoc

Stripe, a popular payment service uses a similar three-panel view for their APIs. However, they created a tool called Markdoc (https://markdoc.dev/) that helps generate their documentation. There are more tools for handling API documentation and documentation in general. Futuristic FinTech likely already provides multiple tools for this purpose, so we want to follow our company's current standards and suggest new tools as needed.

4.5.3 What is so great about API documentation?

Documentation is one of those things that can be a struggle to get right, which is why the ability of FastAPI to provide us with OpenAPI documentation is so fantastic. Why? Because we can address one of the main documentation pain points, which is that it

becomes stale quickly. With FastAPI, we can decorate our functions so that they generate the documentation, meaning we are less likely to be out of sync between what the documentation shows and what the code is doing.

That is not all though. We really only touched on the basics of what is available. In addition to being able to interactively see the APIs and try them out, here are some of the other cool features available:

- Providing contact information in the form of URL or email addresses that end users can use
- Licensing information about your APIs
- Pointing to different servers (e.g., a test server, sandbox, or production)
- Providing examples of requests/responses
- Generating server-side code to handle requests to the APIs
- Generating client-side code to interact with the APIs

Those are just some interesting features of designing APIs that conform to the OpenAPI specification. To learn more, check out *Designing APIs with Swagger and OpenAPI* (2022, Manning) by Joshua S. Ponelat and Lukas L. Rosenstock.

4.6 Managing our APIs

At this point, our API is small, consisting of a single `POST` and four `GET` statements. As our API grows, we need to have a strategy in place to manage it. This is best determined early as once APIs are in production and used by clients, it becomes more difficult to get a handle on the situation. Earlier, we mentioned the challenges of managing APIs such as versioning, documentation, monitoring, and so forth. We introduce the use of API management software so that you have some knowledge of its capabilities. We use API management as an umbrella term to cover the components that are found in the tool.

We will incorporate an API Manager by WSO2 (https://wso2.com/) into our solution just to highlight some of the benefits of using an API manager. There are many API management solutions, but we have worked directly with WSO2 and Mulesoft Anypoint (https://mng.bz/4a2V) in the FinTech industry.

The WSO2 solution was easy to integrate into our solution. In addition, the documentation and tutorials for WSO2 are highly informative. As mentioned before, the use of Docker makes switching out components easy. Feel free to work on integrating other API managers into your solution that may better match your environment. The underlying mechanics and benefits from working with an API manager should be the same across the board.

4.6.1 Benefits of API management

API managers can provide a more centralized approach to managing your APIs. If your business has multiple applications, they are likely to run on different servers, require different authentication, and expose multiple points to the outside world from a

networking perspective. API managers provide a way to centralize your API management. They offer features that help monetize APIs, protect against potential attacks with rate limiting (which can also be used for monetization by supplying different tiers), and provide authentication/authorization.

Another aspect of managing APIs in larger corporations is providing a central place for our users to access. If we have distinct services and business units in our company, they more than likely have different servers handling their APIs, yet want to provide a single experience to their users. We can have all the APIs sent to a single endpoint, and then the API manager can direct the request to the appropriate server in your network.

Once our requests are flowing through a centralized location, we can provide additional benefits to our users. One of them is a common interface to view and manage APIs as an end-user. If you have experience with APIs such as Google APIs, you should be aware that they offer a console to subscribe to various APIs. Sometimes, as companies grow and acquire other companies or merge, multiple products can be listed under the same company but provide a disjointed experience. By using an API manager, customers can view all your API offerings in one location.

In addition to a better user experience, API managers offer increased analytics for our APIs. We can find out the number of calls, which API versions are in use, and a plethora of other insights. These analytics can lead to additional benefits such as rate limiting, load balancing, and monetization. Many APIs offer a pay-as-we-play model, where we are charged by API usage. API managers help facilitate that monetization process.

Finally, lifecycle management is another benefit of using a third-party tool such as WSO2. Having a tool that helps enforce good lifecycle management practices provides a better experience for your customers and developers. We can see the distinct phases of an API and move APIs between different states. As shown in figure 4.4, we have an API that is currently in a published state, with options to bring it back to a created or deprecated state.

4.6.2 Adding WSO2 API manager to our project

First, we add WSO2 to our Docker setup, create a new folder named wso2, and place the following Dockerfile there:

```
FROM wso2/wso2am:latest
```

There is not much to our Dockerfile, but for consistency, we are sticking with the same layout as for previous setups. The real work happens in our docker-compose.yml file where we added the WSO2 service:

```
  wso2:
    build:
      context: ./wso2
      dockerfile: Dockerfile
    ports:
```

```
    - 9443:9443 # Management Console
    - 8243:8243 # HTTPS Requests
    - 8280:8280 # HTTP Requests
```

`port 9443` is the management console where we will be doing most of our work, and the other ports (8243/8280) are the default ports that we ensure are exposed for their respective request types. Once we complete the usual `docker-compose build` and `docker-compose up` commands, we should be able to log in with the default username and password of `admin/admin`.

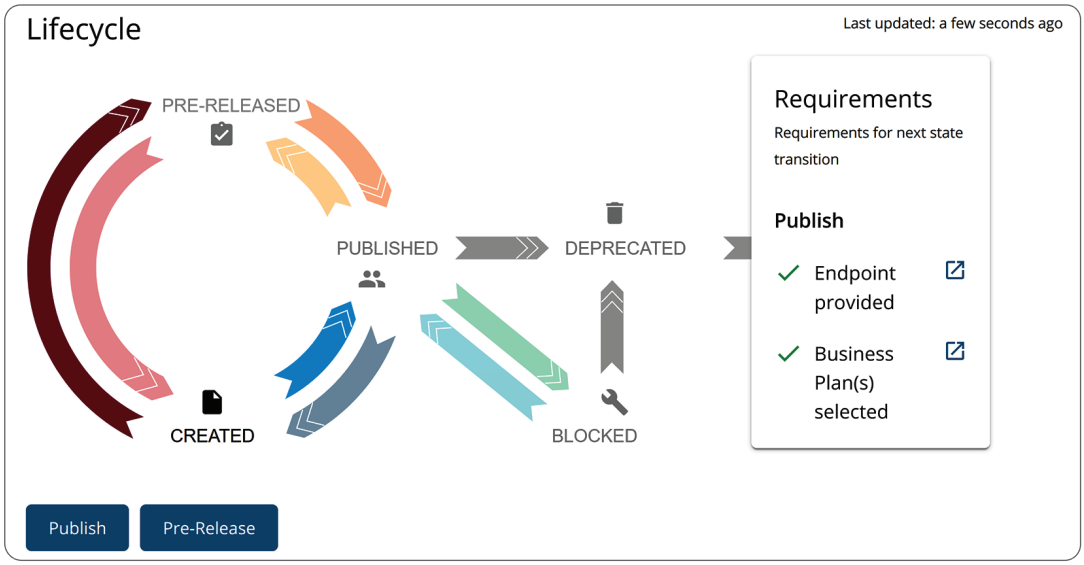


Figure 4.4 Lifecycle management by WSO2

4.6.3 OpenAPI Version

One problem that we encountered while defining our API to WSO2 was `OpenAPI content validation failed!` We loaded our OpenAPI specification into the online Swagger Editor and received the error `Structural error at openapi should match pattern "^3\.0\.\d(-.+)?$" pattern: ^3\.0\.\d(-.+)?$`.

The online Swagger Editor provided more information, and while it may be a little cryptic for those having no experience with regular expressions, it pointed to the version number being incorrect. It seems that both our Docker image for WSO2 and the Swagger Editor do not currently support the latest version, but our FastAPI container does. To be sure we are using the correct version, we can double-check the OpenAPI site.

ChatGPT was aware of the latest version, and when we posted the specification and asked it "Is this a valid OpenAPI specification?" we received the following response:

The OpenAPI specification you have posted is quite extensive, and at a glance, it seems to be structured correctly according to the OpenAPI 3.1.0 specifications. However, there are several nuances and small details in such a spec that need to be right for it to be valid. Here are some key points and common issues to check in a detailed spec like this.

Out of curiosity, we also tried to type in an invalid OpenAPI version (a much higher number), and ChatGPT said that it was not valid.

We could go about tackling this problem in a few ways. It might be that there is an updated version of WSO2 that supports version 3.1.0. However, this may not always be practical or quick in a production environment. We could look to downgrade FastAPI back to a previous version, but let's assume there is functionality or bug fixes we need in the current version. In this case, we can update our API to report back the version of 3.0.0. We could also just manually update the generated document before uploading it to WSO2. However, there is also an option from within WSO2 to go directly to a URL, and we want to be able to employ that.

We update our main.py, adding the following.

Listing 4.12 Adjusting the OpenAPI version

```
from fastapi.openapi.utils import get_openapi
…
def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema
    openapi_schema = get_openapi(
        title="File API",
        version="0.0.1",
        routes=app.routes,
        openapi_version="3.0.0"
    )
    app.openapi_schema = openapi_schema
    return app.openapi_schema

app.openapi = custom_openapi
```

Imports the get_openapi to allow us to customize the OpenAPI documentation; our previous code remains the same.

Defines a custom method

These fields are required when calling get_openapi.

This is how we updated the OpenAPI version, and we had to fix a problem with WSO2 API manager.

With that in place, if we rebuild and restart Docker, we should see `"openapi":"3.0.0"` instead of `"openapi":"3.1.0"` when viewing the JSON. We should also see the error resolved when viewed in the online Swagger Editor. Now, we can move on to creating the API in WSO2.

4.6.4 Create an API in WSO2

We can create an API by going through the following steps:

1. Click the Create API button.

2. Under REST API, click Import Open API.
3. For the OpenAPI URL, enter http://api:8000/openapi.json. We should see that the URL is validated by the system.
4. Enter a name (this is what the API will be known as in WSO2):
   a. The context is a way to distinguish the API URL in WSO2.
   b. The version of the API.
   c. The endpoint should be http://api:8000.
5. Click Create.
6. On the left pane, under Deploy, click Deployments.
7. Click Deploy.
8. On the left pane, under Test, click Try Out:
   a. We will be directed to a similar Swagger Doc screen we saw earlier.
   b. Click Generate Key.
9. Try out your APIs.

With this API in place, the thing to notice is that we no longer make a request directly to our backend of http://api:8000. Instead, we are going to make requests to https://localhost:8243/sample/0.0.1/files, which is the API manager, the context we chose, version, and our API. If we were to keep the API manager in place, we would stop exposing `port 8000` in Docker so that we would force anyone interested in using the APIs through the single-entry point.

Note that as we currently have things set up, we won't be saving the APIs permanently in the manager. So, remember that if we spend a lot of time setting up and configuring the APIs, anything we do is not permanent. We just wanted to set up this sandbox because we believe it is beneficial for you to have some experience with API management tools as you are likely to encounter them, especially in larger companies.

4.6.5 Troubleshooting

When working with an API manager and trying an API out, the most common error is

```
  "code": "900902",
  "message": "Missing Credentials",
  "description": "Invalid Credentials. Make sure your API invocation call
has a header: 'Authorization : Bearer ACCESS_TOKEN' or 'Authorization :
Basic ACCESS_TOKEN' or 'apikey: API_KEY'"
}
```

The `Missing Credentials` error is caused by not selecting Generate Key before testing the API. When working with APIs in a production environment, we have to generate keys to use with the API either through a dashboard or through another API. In trying out our APIs, we are simply generating a temporary token to use for a brief time.

Working with an API manager or through the OpenAPI documentation pages is a helpful way of doing some initial evaluation and exploratory testing. However, once we

settle on the APIs that we are going to use, we will need to start using additional tools that are more specialized and offer more robust functionality. The following sections illustrate some of the additional tools that we will be using.

4.7 Testing with Postman

The Postman API platform is a popular tool to assist with the API lifecycle, collaboration between developers and other teams, as well as with API governance. We have primarily used it in testing capacity, although the platform offers many more capabilities.

One of our favorite features is the ability to `import/export curl` versions of the command. `curl` is a ubiquitous command line utility that lets us get or send data using a variety of protocols. Many systems administrators will ask you for a `curl` equivalent of your `RESTful API` call when troubleshooting network/routing problems, so having the ability to produce one quickly and easily can be helpful for your entire team.

Collaboration between team members is also easy with Postman, and we have worked closely with our QA department to help develop test scripts that can be used for component and integration tests. Let's look at how we can use Postman to validate data that is returned from our API. This is similar to what we might be doing with our unit tests while developing. However, the perspective is different because these may be run as part of a larger CI/CD pipeline or as part of QA testing process.

We will continue to use the http://localhost:8000/files as our example. Of course, we could also take on the challenge of building a Postman test using an API manager layer. In Postman, we can create a collection that is used to group various requests. Then we can create a new `GET` request that points to the http://localhost:8000/files and click Send. We should see a nicely formatted JSON response that we have condensed here: `[{"file":"File_1"},{"file":"File_2"}]`.

Once we have confirmed we are successfully connecting and receiving responses from the endpoint, we can add a few tests as well: click the Tests tab, and you can add some straightforward tests to get an idea of how the syntax works. First, let's test that we have a valid `200` response code using the following listing.

Listing 4.13 Sample Postman test

```
pm.test("Response should be JSON and have a 200 status code", function () {
    pm.response.to.be.json;
    pm.response.to.have.status(200);
});
```

Check if response format is json

Check if response status is 200

When sending our request, we should see that the Test Results tab is populated, and we have a passing test. We add a few more tests, as shown in the following listing.

Listing 4.14 More Postman tests

```
pm.test("Response should have a file attribute", function () {
    pm.expect(jsonData[0]).to.have.property('file');
});
```

```
pm.test("file attribute should be 'File_1'", function () {
    pm.expect(jsonData[0].file).to.eql('File_1');
});

pm.test("Response has at least 2 items", function () {
    pm.expect(jsonData).to.be.an('array').that.has.lengthOf(2);
});
```

Some of these tests are fragile—meaning they may be easily broken by the data or ordering—so we would expect this code to run against known data. Let's use the test that looks for `File_1` to make it a little more robust. Because our data is in an array, we should be guaranteed when parsing that we will always parse them in that order, as opposed to an object where the `key/value` pairs may not always be returned in the way we see it. Still, the ordering of the JSON depends on factors outside of our control. We may be guaranteed that there is a `File_1`, but maybe it is not always returned first. How can we write a more robust test? What if we iterated through the array using a `for-each` loop?

Listing 4.15 Using a `for-each` loop in Postman

```
pm.test("There should be one instance of File_1", () => {
    let found = false;
    jsonData.forEach(function(item, index) {
        pm.expect(item).to.have.property("file");
        if ( item.file === 'File_1' ) {
            found = true;
        }
    })
    pm.expect(found).to.be.true;
})
```

Now we define a variable to indicate whether we found our desired item, so no matter where it appears in the array, we will find it. That is a more flexible test, and depending on your familiarity with the language, it might be your first attempt, and since it works, that may be enough. However, we can clarify the test a bit using the `some` method.

Listing 4.16 Cleaning up our test

```
pm.test("There should be one instance of File_1 using some", function () {
    const expectedValue = 'File_1';
    const result = jsonData.some(function (item) {
        return item.file === expectedValue;
    });
    pm.expect(result).to.be.true;
})
```

Note that the `some` method is part of ECMAScript 5+ and is available outside of Postman. We have found that using Postman and these test scripts is a fantastic way to introduce newer QA resources to some type of automation. We just scratched the surface of

what Postman has to offer; it is a great tool that we have seen used at various FinTech companies, as well as in other industries.

4.8 Load testing with Locust

Earlier, we introduced the testing pyramid that primarily included testing types we would consider functional. Here, we introduce load testing, which falls into the non-functional testing category. Figure 4.5 outlines some types of testing.

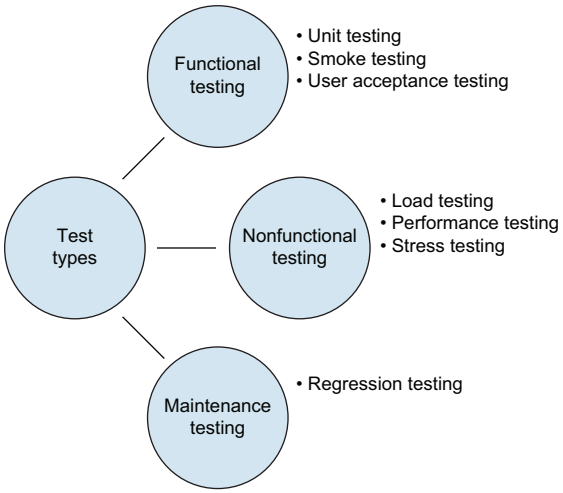


Figure 4.5 Software testing categories

We want to ensure our APIs will be responsive, see where performance bottlenecks might be, and determine the capacity of the system for future planning. To prepare for all of that, we can look to a framework called Locust. Locust lets us use our budding Python skills to script tests used for load testing.

We can easily get started by telling ChatGPT to "create a sample locust test for the http://localhost:8000/files endpoint," and that provides some easy-to-follow steps. First, we want to install Locust with `pip install locust`. Also, we create a locustfile.py as follows.

Listing 4.17 Simple Locust file

```
from locust import HttpUser, task, between

class FileUser(HttpUser):
    wait_time = between(1, 2.5)

    host = "http://localhost:8000"

    @task
    def get_files(self):
        self.client.get("/files")
```

From the directory with locustfile.py, run the command `locust`. This will prompt you to open your browser to the default location http://localhost:8089. Here, we will enter the number of users, spawn rate, the host, and how long it should run for (we clicked the Advanced options) to get that (see figure 4.6).

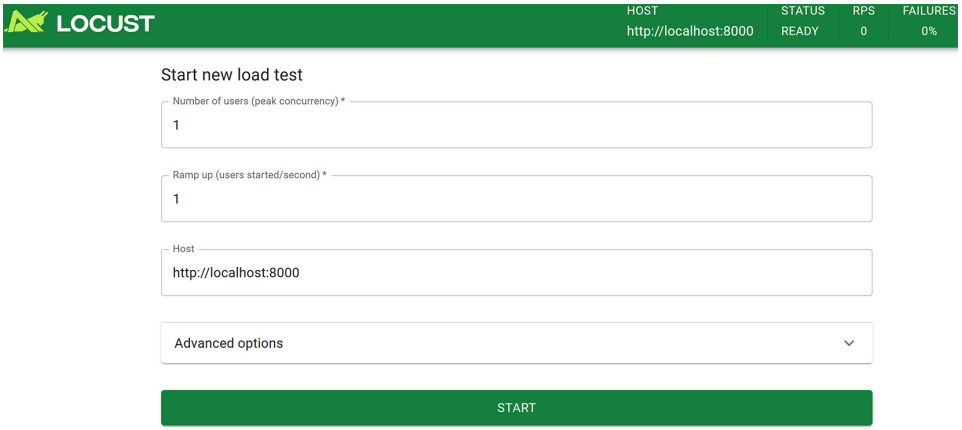


Figure 4.6 Starting a new test with Locust

From there, we can see the test run and get metrics around how our API is performing. Obviously, in this case, we are dealing with hardcoded values, so we will have a better opportunity later to come back and explore load testing our APIs. However, we can go ahead and get some experience with Locust and load testing now. Can we add tests for the other endpoints? If we need a reminder of what they were or what they should be returning, do not forget our living documentation at http://localhost:8000/docs. With additional tests, our code should look like the following listing.

Listing 4.18 Locust test

```
from locust import HttpUser, task, between

class FileUser(HttpUser):
    wait_time = between(1, 2.5)

    host = "http://localhost:8000"

    @task
    def get_files(self):
        self.client.get("/files")

    @task
```

Creates a FileUser class that inherits from HttpUser

Sets a random wait time ranging from 1 to 2.5 seconds between requests to simulate a user pausing between actions

Defines the host where the API requests will be sent

The task decorator defines a user-simulated action.

Issues a GET against the /files URI

```
    def get_file(self):
        self.client.get("/files/1")

    @task
    def post_file(self):
        self.client.post("/files", json={"file_id": 5})

    @task
    def get_records(self):
        self.client.get("/files/1/records")

    @task
    def get_record(self):
        self.client.get("/files/1/records/1")
```

Now, we can rerun Locust to see it hitting the various endpoints. Although our endpoints are currently hardcoded, they will return the path parameters used. So, to make things more interesting, let's add some code to each of these tests so that the parameters are unique, and that will show us hitting more endpoints in the Locust dashboard. Take a few minutes to see whether you can replace the hardcoded `file_ids` and `record_ids` with a random number, and then rerun the tests. The next listing shows the sample we came up with.

Listing 4.19 Locust

```
import random
from locust import HttpUser, task, between

class FileUser(HttpUser):
    wait_time = between(1, 2.5)
    host = "http://localhost:8000"

    @task
    def get_files(self):
        self.client.get("/files")

    @task
    def get_file(self):
        file_id = random.randrange(1, 10)
        self.client.get(f"/files/{file_id}")

    @task
    def post_file(self):
        file_id = random.randrange(1, 10)
        # Sending POST request to the "/files" endpoint
        self.client.post("/files", json={"file_id": file_id})

    @task
    def get_records(self):
        file_id = random.randrange(1, 10)
        self.client.get(f"/files/{file_id}/records")

    @task
    def get_record(self):
```

Generates a random file_id in the range from 1 to 9

```
        file_id = random.randrange(1, 10)
        record_id = random.randrange(1, 20)
        self.client.get(f"/files/{file_id}/records/{record_id}")
```

After rerunning our tests, we can see a lot more variety in the endpoints being accessed. At this point, we may have noticed a problem when executing the tests in Locust on the dashboard. If not, take a look at the Failures tab in Locust. You should see a number of failures for the `POST` method on `/files` with the error `HTTPError('405 Client Error: Method Not Allowed for url: /files')`. It looks like we missed something in our API code. But before checking it, let's take a look at another big problem with our Locust tests—no `assert` statements.

Similarly to all our other tests, we want to make sure we are doing some sort of the `assert` statement. Let's start by ensuring a valid response each time. Feel free to give that a try before reading further.

Listing 4.20 Locust test

```
...                                                    # Previous code setup from listing 4.19
@task                                                  # Uses the Locust task decorator to define what Locust should be executing
def get_files(self):                                   # Defines a get_files method that executes a GET against the /files URI and checks the HTTP status was 200
   response = self.client.get("/files")
   assert response.status_code == 200,
      f"Unexpected status code: {response.status_code}"

@task
def get_file(self):
   response = self.client.get(f"/files/{file_id}")
   assert response.status_code == 200,
      f"Unexpected status code: {response.status_code}"

@task
def post_file(self):
   response = self.client.post("/files", json={"file_id": file_id})
   assert response.status_code == 200,
      f"Unexpected status code: {response.status_code}"

@task                                                  # Makes requests using a random file_id in the range from 1 to 9
def get_records(self):
   file_id = random.randrange(1, 10)
   response = self.client.get(f"/files/{file_id}/records")
   assert response.status_code == 200,
      f"Unexpected status code: {response.status_code}"

@task                                                  # Makes requests using a random file_id in the range from 1 to 9 and a random record_id in the range from 1 to 19
def get_record(self):
   file_id = random.randrange(1, 10)
   record_id = random.randrange(1, 20)
   response = self.client.get(f"/files/{file_id}/records/{record_id}")
   assert response.status_code == 200,
      f"Unexpected status code: {response.status_code}"
```

Now that we have some assert statements, we can see an `Unexpected status code: 405` in the Exceptions tab in addition to the Failures tab. Let's check out the logs for the API container and see what's there. The logs show two interesting problems:

```
"GET /files/3 HTTP/1.1" 200 OK
"POST /files HTTP/1.1" 405 Method Not Allowed
"POST /files HTTP/1.1" 405 Method Not Allowed
"GET /files/6/records HTTP/1.1" 307 Temporary Redirect
"GET /files HTTP/1.1" 200 OK
```

As we saw, the `POST` is getting a `405 Method Not Allowed`, and we also see a `307 Temporary Redirect`. When we review our code that defines the endpoints, we see that there was a trailing slash for some endpoints but not for others:

```
@router.get("/files")
@router.get("/files/{file_id}")
@router.post("/files/", status_code=status.HTTP_201_CREATED)
@router.get("/files/{file_id}/records/")
@router.get("/files/{file_id}/records/{record_id}")
```

All the Locust requests left the trailing slash off the requests, so let's make sure we are consistent in both the server and the client. In our example, we are going to remove the trailing slashes from the server portion. Rebuilding our containers and rerunning our tests give us a more consistent experience without errors in the logs or failures in Locust. We do see that we still have exceptions in Locust. This is because we coded all our assert statements to be expecting a `200` return code. However, when we refer to the documentation (or by looking at the code), we are reminded that our `POST` request responds with a `201` not `200`. With that taken care of, we see our tests running normally, with review failures, exceptions, and the associated analytics as shown in figure 4.7.

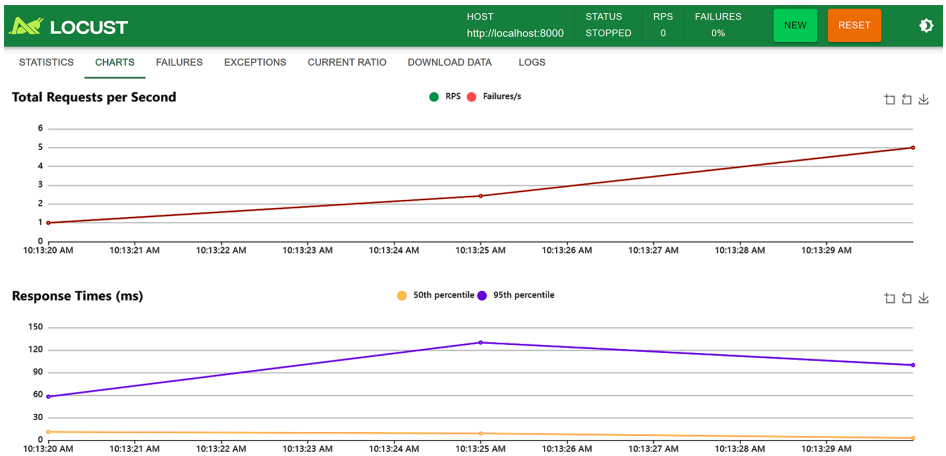


Figure 4.7 View of successive Locust runs

4.9 Improving our code

Now that we have both unit and load tests, you should feel comfortable working within your code to make improvements. Even though the code is straightforward, we can still take advantage of some benefits of FastAPI. This section shows the benefits of versioning, using prefixes, documentation, and validation.

4.9.1 Using a prefix and versioning

You may have noticed that each of our endpoints in the router includes `/files`. While this works fine, we would also like to include a version number, as well as put the endpoint under an `api` endpoint in the URL. We want to end up with something like `/api/v1/files` as our initial standard. Using `/api` helps with organizing our APIs in one area. This can help with web server configuration, firewall rules, monitoring, and similar. The `v1` indicates the API version, and we can use it to support multiple API versions. Remember that once end-users start consuming the API, it is unwise to make breaking changes to your API that would force them to upgrade their applications. In these cases, you should provide the new API under `/v2` and eventually phase out prior versions when possible. Remember, the `/api/v1` is just a suggestion. You could name the endpoint anything you choose and use any versioning scheme based on your needs.

We refactor our code to add the additional `/api/v1` to our endpoints by using the `prefix` parameter available in `APIRouter`. We update our files.py:

```
router = APIRouter(prefix="/api/v1/files")
```

Then, we remove the `/files` from each of our defined endpoints. Table 4.3 shows the differences.

Table 4.3 Effects of using a `prefix` parameter on the router

Before	After
`@router.get("/files")`	`@router.get("")`
`@router.get("/files /{file_id}")`	`@router.get("/{file_id}")`
`@router.post("/files ", status_code=status.HTTP_201_CREATED)`	`@router.post("", status_code=status.HTTP_201_CREATED)`
`@router.get("/files /{file_id}/records")`	`@router.get("/{file_id}/records")`
`@router.get("/files /{file_id}/records/{record_id}")`	`@router.get("/{file_id}/records/{record_id}")`

We can see that by using the prefix, we now have a little less typing to do in each endpoint. We also need to rerun our unit tests to see whether things still work, and as we may have suspected, our tests are all failing with `404 errors` because they are no longer going to the correct URL. We need to go ahead and update our unit tests to include the `/api/v1` and rerun them.

4.9.2 Adding Pydantic

Pydantic is a widely used data validation library that comes bundled with FastAPI. We will incorporate it here to move away from our hardcoded strings (although we will still be returning hardcoded values). By adding Pydantic, we position our code to be more easily expanded in the future and are able to take advantage of some of its features for validation and documentation.

Later, we will look to build objects that model actual ACH files and records. For now, let's take a look at how Pydantic will enhance our current APIs. Our first endpoint performs a `GET` on `/api/v1/files/{file_id}` where we previously returned a JSON string. It looks something like this (with the new prefix defined).

Listing 4.21 Our previous hardcoded function

```
@router.get("/{file_id}")
async def read_file(file_id: str):
    return {"file_id": file_id}
```

Now, we create an `AchFile` class that will model the hardcoded value we were returning in the previous listing.

Listing 4.22 Our first `Pydantic` class

```
from pydantic import BaseModel, Field

class AchFile(BaseModel):
    file_id: str
```

With that simple class in place, we can update the router to return an instance of that class.

Listing 4.23 Updating our route to use our newly created `Pydantic` class

```
…
from ach.AchFile import AchFile
…
@router.get("/{file_id}")
async def read_file(file_id: str) -> AchFile:
    return AchFile(file_id=file_id)
```

Our unit tests should still pass, so we have not broken anything. If we were to start this up in Docker, we would run into an error of `ModuleNotFoundError: No module named 'ach'`. This is because Python does not know where to find our class, and there is a possibility that we even forgot to copy it over! Before starting, add another `COPY` statement to your Dockerfile and set the `PYTHONPATH`. This should get our code up and running.

Listing 4.24 Updating our Dockerfile

```
COPY ./app /code/app
COPY ./ach /code/ach

ENV PYTHONPATH /code
```

Now we should be able to bring down Docker, build, and bring our containers up again. If we take a look at the documentation, we will see an `AchFile` in the schema section. Let's continue to expand on the documentation by adding a `Field` method to provide additional documentation.

Listing 4.25 Expanding the documentation of our `AchFile` class

```
from pydantic import BaseModel, Field
…
    file_id: str = Field(examples=["1", "2"], title="ACH File ID",
description="The ACH file ID used to identify the file in the system")
```

Imports the Field function

Adds parameters that will help with the OpenAPI documentation

By adding some examples, title, and description, the schema becomes more meaningful. Figure 4.8 shows how the OpenAPI documentation was updated.

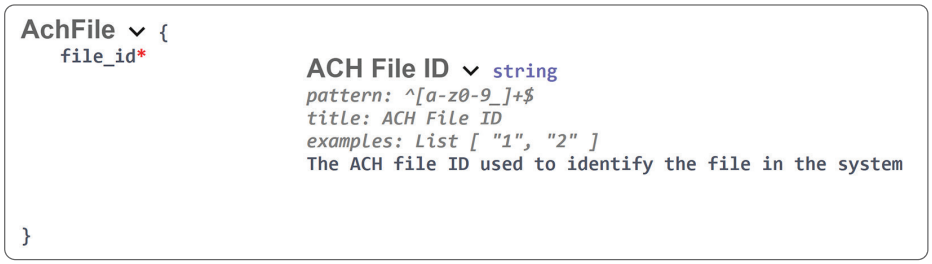


Figure 4.8 Our updated `AchFile` schema

This is just the beginning of making our API user-friendly. We will expand on the concept as we build more APIs.

4.9.3 Validation with Pydantic

Another benefit of Pydantic is its built-in validation capabilities. Validation of data entering the environment is a must in any production system (unless we are looking to test the resilience of our system). We should not rely on the UI or another system to validate data. If end-users are given free rein to enter any type of data in a field, they will do so, and it does not matter whether it makes sense. By combining Pydantic with

FastAPI we can get a lot of validation for free where we may have once needed a lot of boilerplate code.

For example, we could enforce our `file_id` to be at least one character and contain only lowercase letters, underscores, and numbers. There are multiple approaches to enforcing such validations. In Pydantic, for instance, we can use a regular expression. By simply adding a pattern parameter to the field, we can enforce this constraint:

```
pattern=r"^[a-z0-9_]+$"
```

Now that we are enforcing validation, let us jump back to `main.py` and add an exception handler when a validation error occurs. In this case, we will return a `400 status` along with some detailed error information. We need to be careful that our exception messages do not leak sensitive information. In this case, it is a validation error triggered by missing or malformed fields. Still, this may be more appropriate for a development environment; in production, we may want to have a more robust method of translating the error into a known message to reduce the risk of exposing any sensitive data. Our exception handler should look like the following.

Listing 4.26 Validation in Pydantic

```
from fastapi import FastAPI, Request, status, HTTPException
from fastapi.responses import JSONResponse
…
@app.exception_handler(ValidationError)
async def validation_exception_handler(request: Request, exc:
     ValidationError):
    return JSONResponse(
        status_code=status.HTTP_400_BAD_REQUEST,
        content={"detail": exc.errors()}
    )
```

Decorator that registers a custom exception handler for ValidationError exceptions

Ensures the HTTP status of 400 is returned along with error details

From here, we can add additional unit tests that confirm our validation behavior.

Listing 4.27 Validating our exception behavior

```
def test_read_file_lower_case():
    response = client.get("/api/v1/files/lower_case_file_id")
    assert response.status_code == 200
    assert response.json() == {"file_id": "lower_case_file_id"}

def test_read_file_validation_error():
    response = client.get("/api/v1/files/this_IS_invalid")
    assert response.status_code == 400
    assert response.json()["detail"][0]["msg"] ==
➥"String should match pattern '^[a-z0-9_]+$'"
```

A test that passes validation

Testing when we receive an exception condition. Notice the status code of 400 and also a detailed message explaining what the pattern is.

We add the positive case to show that we accept a lowercase `file_id`, and when uppercase values are used, we see the error telling us what the pattern should be.

In fact, we get a lot of useful information returned by default. Whether it is appropriate to show all this to your end-user may be another matter. API usage is in most cases fine to return detailed information, but we want to caution against revealing too much. For instance, you may notice that most sites do not tell you whether you have typed your username or password incorrectly. They simply say that one or the other is incorrect. This is a conscious design decision made to prevent malicious users from attempting to determine a list of valid users. Therefore, we sometimes return more generic information to the user and log the details in a system log. In this case, we are fine since it is acting as an interactive guide to the API.

Listing 4.28 Detailed error that we receive from Pydantic

```
{
  "detail": [
    {
      "type": "string_pattern_mismatch",
      "loc": [
        "file_id"
      ],
      "msg": "String should match pattern '^[a-z0-9_]+$'",
      "input": "DFSDF",
      "ctx": {
        "pattern": "^[a-z0-9_]+$"
      },
      "url": "https://errors.pydantic.dev/2.4/v/string_pattern_mismatch"
    }
  ]
}
```

We just touched on the basics of validation in Pydantic and FastAPI. A broad range of validation options in addition to regular expressions, as well as the ability to create custom validators, are available. We will explore this in later chapters.

4.9.4 Troubleshooting Pydantic

We may see a `500 Internal Server Error` if we failed to copy over our updated API code. In our custom handler, we are intercepting the validation error thrown and returning a `400 error` code. If we see something similar to error shown in the following listing, we have likely made that oversight or our handler is not working.

Listing 4.29 Docker error in FastAPI

```
docker-api-1   | INFO:     192.168.0.1:37630
 ➥- "GET /api/v1/files/DFSDF HTTP/1.1" 500 Internal Server Error
docker-api-1   | ERROR:    Exception in ASGI application
docker-api-1   | Traceback (most recent call last):
docker-api-1   |   File "/usr/local/lib/python3.12/
```

```
➥site-packages/uvicorn/protocols/http/h11_impl.py",
➥line 408, in run_asgi
docker-api-1   |     result =
➥await app(  # type: ignore[func-returns-value]
```

Summary

- API-driven design promotes an API-first approach to development and ensures your software meets today's distributed nature.
- Exploring the design and testing of APIs is crucial for interaction with software components in microservices, distributed systems, and packages.
- REST is the primary API standard discussed, but GraphQL and SOAP are other important standards with unique pros and cons.
- APIs act as contracts requiring adherence to usage terms and providing stable, scalable endpoints with SLAs for consumers.
- API-driven design emphasizes clear use cases, versioning, comprehensive documentation, and robust security and validation.
- Effective API monitoring and analytics are essential for capacity assessment, using tools such as OpenTelemetry, Prometheus, and Grafana.
- Initial API tasks are simple, focusing on uploading and viewing files and records with a clear interaction flow.
- FastAPI is used for building APIs, starting with basic `GET` endpoints, and supporting pagination and asynchronous processing for performance.
- Unit tests validate API endpoints during development, while additional tests such as BDD can assess API behavior and performance.
- API documentation via OpenAPI standards ensures alignment between implementation and user understanding, offering tools such as Swagger and Redoc.
- API management tools such as WSO2 centralize management, provide lifecycle control, and facilitate API monitoring, monetization, and user experience.

5 Storing our ACH files

This chapter covers

- Creating tables within our PostgreSQL database
- Designing a relational database capable of storing ACH files
- Using Python and Pydantic to validate ACH records and store them in our database
- Ensuring that our records are parsed and stored correctly by implementing unit testing with `pytest`

In this sprint, we use another research spike to explore how to define our database. Databases store and persist our data across application instances, while providing a way to query and ensure the integrity of that data. Here, we examine how to store our ACH files in a database. After initial analysis, we expand our APIs to store an ACH file in the database. Continuing along that track, we expand our ACH parser to store the individual fields as well. Finally, we wrap up the chapter by examining how storing ACH data affects our unit and load tests.

The introduction of a database is necessary in our project because an ACH file is a flat file. The current ACH system that Futuristic FinTech uses relies on flat files,

and they can be challenging in many areas, including performance, querying, and data integrity. For instance, if a customer questions when a transaction was loaded, all the ACH files must be loaded and parsed to perform the search, which is time-consuming. Furthermore, keeping the parsed ACH files in memory becomes unfeasible given the number of records being dealt with.

5.1 Designing our database

When a user uploads files to our ACH dashboard, we obviously need the ability to save them, or our system will not be very useful. The current ACH dashboard for Futuristic FinTech does not use a relational database. Instead, once uploaded, the files are parsed and stored in flat files (i.e., unstructured text files), which makes more sophisticated functionality a chore to implement. The ACH dashboard we are replacing uses only the filesystem to store the files. To provide more advanced processing, we want our ACH dashboard to be backed by a relational database, and we work through the initial review and implementation of various database designs and concepts to support our dashboard. Often, we need to have these types of research stories included in our sprints to examine different ways we may go about when implementing a desired feature.

There are at least a dozen different relational databases, and FinTech uses many of them. Our choice of database is often already determined by the database our company uses. We have seen Oracle, MySQL/MariaDB, and PostgreSQL used by FinTech—to name just a few. In our case, we have already set up an environment that enables PostgreSQL to run in a Docker container, and we have seen how to build/initialize tables during startup and view our data through CloudBeaver. We can now start expanding our database to accommodate storing ACH files.

Databases do much more than just storing our data—they can help ensure the reliability and consistency of the data and relationships, a concept known as *referential integrity*. Referential integrity in our database is a fancy way to say that we will ensure our tables are appropriately related and the fields are correctly defined. For instance, recall that ACH is a fixed-length format, meaning the individual fields are fixed length. We may store the file ID modifier from the file header record as a `VARCHAR(1)` since it can only be a single uppercase character (or 0 through 9). Similarly, we may want to store the total debit entry amount in the file control (i.e., file trailer) record as `MONEY` or `NUMERIC(12,2)`. The `NUMERIC(12,2)` defines a field with a precision of 12 significant digits and a scale of 2, which is the number of decimal digits. Whether you'll use `MONEY` or `NUMERIC` is up to you, but we favor the `NUMERIC(12,2)` representation as it closely resembles the field definition.

Another aspect of referential integrity is preventing orphan records. Remember that there is a hierarchy of records in an ACH file, that is, file control → batch header → entry records → etc. For example, an orphan record may occur if we have not defined our database carefully and we were to delete a batch header record. Once we delete a batch header, all the entry and addenda records (as well as the batch control record)

belonging to that batch are no longer valid, and we should delete them. Likewise, deleting the file control record should delete all records associated with that file. In our relational database, we can achieve this by creating a `FOREIGN KEY`—which references the other table entries—and using `ON DELETE CASCADE`.

Our initial database will use the inherent benefits of a relational database by defining the following:

- *Primary keys*—Unique identifiers for each record in a table
- *Foreign keys*—A link between two tables where a field (or fields) in one table refers to unique data (such as the primary key) in another table
- *Constraints on the fields*—For example, `NOT NULL` (to ensure data is present), `UNIQUE` (to ensure data is unique across all rows), and `DEFAULT` (to assign a default value if none was provided)
- *Data integrity*—Attained by defining appropriate data types and sizes of fields

We'll first look at storing ACH files with only Prearranged Payment and Deposit (PPD) data to keep things simpler. The PPD code is typically used for direct deposits and recurring payments such as payroll and pensions, so it is a widely used code that may frequently affect you (without your knowledge). To get an overview of what our database will look like, we again rely on PlantUML for a rendering of a proposed database structure.

Listing 5.1 PlantUML definition of our database

```
@startuml

object ach_file_uploads {
    ach_files_id: UUID
    filename: VARCHAR(255)
    file_hash: VARCHAR(32)
    credit_total: NUMERIC(12,2)
    debit_total: NUMERIC(12,2)
}

object ach_files {
    ach_files_id: UUID
    record_type: VARCHAR(1)
    record_id: UUID
    parsed: BOOLEAN
    sequence: NUMERIC
    unparsed_record: VARCHAR(94)
}

object ach_file_header_records {
    record_id: UUID
    fields for record type
}

object ach_batch_header_records {
```

Begins a PlantUML definition

Defines a table for our diagram

Defines fields within the table for our diagram

```
    record_id: UUID
    file_header_id: UUID
    fields for record type
}

object ach_entry_detail_ppd_records {
    record_id: UUID
    batch_header_id: UUID
    fields for record type
}

object ach_addenda_ppd_records {
    record_id: UUID
    entry_detail_id: UUID
    fields for record type
}

object ach_batch_control_records {
    record_id: UUID
    batch_header_id: UUID
    fields for record type
}

object ach_file_control_records {
    record_id: UUID
    file_header_id: UUID
    fields for record type
}

ach_file_uploads::ach_files_id <-- ach_files::ach_files_id

ach_files::record_id <-- ach_file_header_records::record_id
ach_files::record_id <-- ach_batch_header_records::record_id
ach_files::record_id <-- ach_entry_detail_ppd_records::record_id
ach_files::record_id <-- ach_addenda_ppd_records::record_id
ach_files::record_id <-- ach_batch_control_records::record_id
ach_files::record_id <-- ach_file_control_records::record_id

ach_batch_header_records::file_header_id ->
     ach_file_header_records::record_id
ach_entry_detail_ppd_records::batch_header_id ->
ach_batch_header_records::record_id
ach_addenda_ppd_records::entry_detail_id ->
ach_entry_detail_ppd_records::record_id
ach_batch_control_records::batch_header_id -->
ach_batch_header_records::record_id
ach_file_control_records::file_header_id -->
ach_file_header_records::record_id

@enduml
```

Shows the relationships between keys in the table

Ends the PlantUML definiton

The preceding definition is presented in figure 5.1, which shows how we may define our fields and the relationships between our tables. This is not an exhaustive list of the fields in tables, but it gives us an idea of how our tables will be related. The arrows

represent the foreign key constraints present in the table. For instance, we can see how `ach_files_id` in the `ach_files` table field is defined as a Universally Unique Identifier (UUID) and references the `ach_files_id` from `ach_file_uploads`.

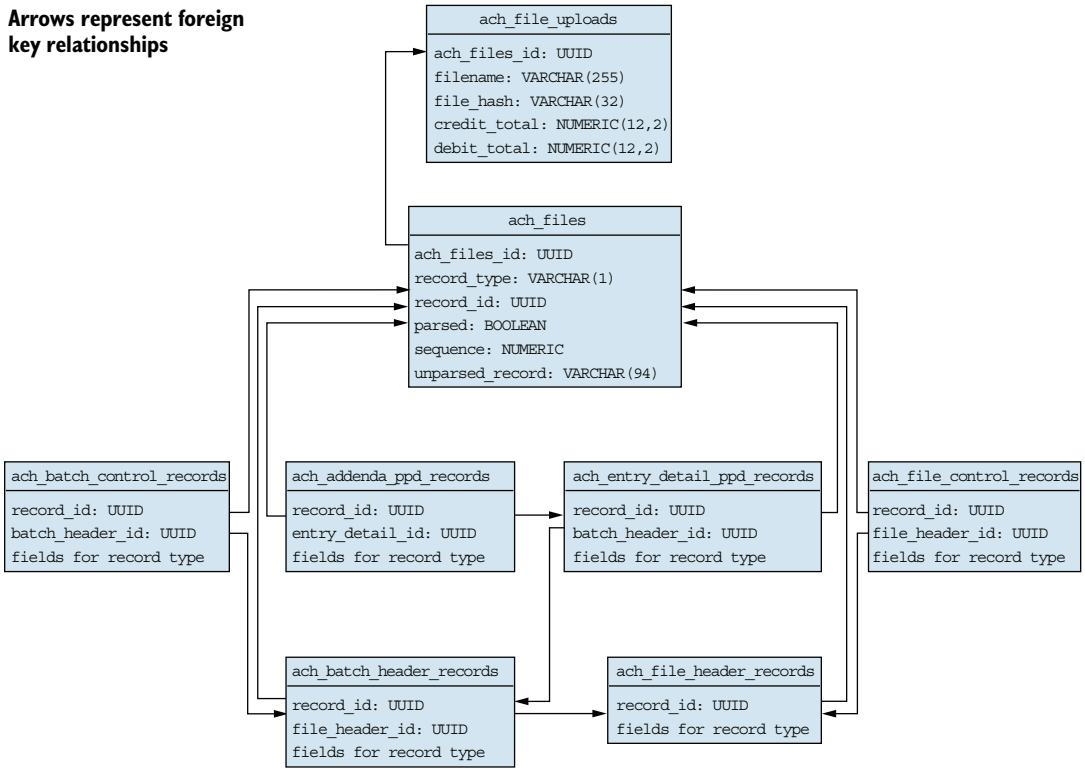


Figure 5.1 Diagram showing relationships between our tables

Figure 5.1 also conveys our desire to accomplish the following goals:

- Maintain the ordering of the records in our file
- Assume records will be unparsable and accommodate for that with unparsed records
- Maintain referential integrity by having parsed records refer to the unparsed ones

While the database structure seemingly meets those goals and the diagram gives us a visual guide to get started, there is always room for improvement. Regardless of whether this structure was provided to us by a subject matter expert (SME) or the interpretation of a database analyst (DBA), there may be opportunities to refine our work as we move through the project. With our diagram in hand, we should have an idea of how we want the database to look and can start working on it. However, we need

to follow the general pattern of defining a test, building the table/fields, and finally building the API. When working with an SQL database, it is important to understand that companies will certainly have different approaches to managing their data. Some companies may extract the SQL portion away from the developer, either by using an object-relational mapper (ORM) such as SQLAlchemy or by rolling their own. An ORM helps simplify code by abstracting the database and providing benefits such as database agnostic, optimizations, and improved productivity.

Other companies may require you to write the SQL yourself because they like the level of control afforded by direct SQL. ORMs may make complex queries difficult or inefficient. Furthermore, troubleshooting queries and performance problems may also be more difficult to track down. First, we show here how to use straight SQL commands as we have been doing and then move to SQLAlchemy so that you can get acquainted with both approaches. There are always several factors that will be added, regardless of whether we use one approach over the other or combine them. Usually, the existing software will dictate our approach, so be careful not to stray too far from existing software standards initially as this could create a maintenance nightmare.

5.2 Using SQL directly

Chapter 3 explored parsing an ACH file and creating accompanying unit tests. We also created a simple API that accessed a database and returned results. So, we have all the pieces we need to store an ACH file in a database. Now we only have to put them together. In our first approach, we update our parser to store the ACH file in the database. We make the following assumptions:

- The database is always up and running.
- We do not have any data that we want to preserve.
- We are running our code from an IDE and not inside Docker.

In other words, we are just working on the process of being able to parse and store the records in the database. We build out the previous `AchFileProcessor` to allow it to store ACH files. The next listing shows adding the necessary code to obtain a connection to the database. Since we are in a research sprint, we have hardcoded the database connection parameters such as username and password. Later, when we are certain that this is the right approach, we can start to abstract some of these hardcoded values so we have a more flexible configuration by using `environment` variables or secret management tools such as AWS Secrets Manager (https://aws.amazon.com/secrets-manager/) or HashiCorp Vault (https://www.hashicorp.com/products/vault).

Listing 5.2 Adding database fields

```
class AchFileProcessor:
    records = []
    exceptions = []
    last_trace_number = None
    expected_record_types = ["1"]
```

```
POSTGRES_USER = "someuser"
POSTGRES_PASSWORD = "supersecret"
DATABASE_URL = f"dbname={POSTGRES_USER} user={POSTGRES_USER}
➥    password={POSTGRES_PASSWORD} host=postgres port=5432"
...

def get_db():
    conn = psycopg.connect(self.DATABASE_URL)
    return conn
```

Temporarily hardcodes our username and password

A hardcoded host and port, but we should consider parametrizing those as well.

A new function to return a connection to the database

That should provide us with the ability to connect to the database; however, we will need to use this code. Before we begin parsing the ACH file, we want to simply store the unparsed records in the database in what we may want to consider an extract–load–transform (ELT) approach rather than directly parsing the records through an extract–transform–load (ETL) approach.

ELT vs. ETL

There are two approaches to handling data when processing it. They are usually discussed with regard to data warehousing and business intelligence, but ACH processing has unique challenges. ETL (extract–transform–load) is a traditional approach and may cross our mind first when processing data. For instance, we know that we want to parse each of these ACH records into their respective fields and store them in a database. However, dealing with data that may not be formatted correctly all the time is one of the challenges in ACH processing. With an ETL approach, this invalid data may cause processing to halt completely.

In an ELT (extract–load–transform) approach, the data is loaded and then transformed once it is going to be used. Typically, we see ELT when dealing with very large data and using data warehouses such as Snowflake or Redshift, where there is enough processing power to perform transformations on request.

So, why do we care about these approaches? Often, financial institutions will allow some leeway with the data and will not always reject a file if an error is considered recoverable. These conditions can vary from one financial institution to another and with their customers. For example, a count of records on a control file may be updated if incorrect rather than just rejecting the file or batch. Or, invalid data in a field may just be updated to spaces before loading it. While we can rely on logging and exceptions to keep track of changes (and we should be logging any such changes), we still want to keep a record of the original file so that a banker has the opportunity to review the data.

5.2.1 Adding records to the ach_files table

We will take the approach of loading our ACH file unaltered into the database before parsing out any records. Figure 5.2 illustrates a basic flow for this section.

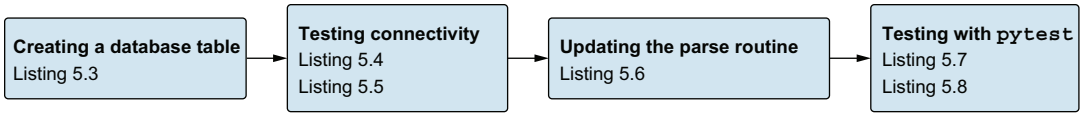


Figure 5.2 Flowchart for section 5.2

Listing 5.3 shows an `ach_files` table. So, before parsing any records, let's just add all our records to that database table.

Listing 5.3 Simple `ach_files` table

```
-- Create the uuid extension
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Create the ach_files table
CREATE TABLE ach_files (
    ach_files_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    file_name VARCHAR(255) NOT NULL,
    unparsed_record VARCHAR(94) NOT NULL,
    sequence_number INTEGER NOT NULL
);
```

Allows PostgreSQL to create UUIDs

Primary UUID key

The unparsed ACH record stored as is

A sequence number used to maintain ordering when retrieving data

With this code, running `docker-compose down`, `docker-compose build`, and `docker-compose up` should allow our table to be created, and now we just need to update the code to write out some records!

Before we write out any records, we must ensure that the basics are working. So, we simply add the following code at the beginning of the existing parse routine. This code merely gets a connection to the database, obtains a cursor that can be used to execute SQL statements, and then closes both the cursor and connection.

Listing 5.4 Testing the connection

```
def parse(self, filename) -> [List, List]:

    conn = self.get_db()
    cursor = conn.cursor()
    cursor.close()
    conn.close()

    with open(filename, "r") as file:
        lines = file.readlines()
```

Calls our new get_db() function to connect to the database

Creates a cursor used to execute SQL commands

Closes the cursor

Closes the connection

If we were to run our code at this point, we may encounter several problems. First, our hardcoded `DATABASE_URL` `string` has a host of `postgres` and a port of `5432`. We are

working within an IDE and not within Docker, so `postgres` is not the correct name to use. Indeed, we should see an error `psycopg.OperationalError: connection is bad: Unknown host` if we were to run the program. Instead, we want to use `localhost` because we are calling from the system hosting Docker.

In addition, we need to expose the port for our container. Our docker-compose.yml should look like the one in listing 5.5. Without exposing the port, we would see an error similar to `psycopg.OperationalError: connection failed: :1), port 5432 failed: could not receive data from server: Socket is not connected`, which would hopefully tip us off that there is a problem with our port. Remember that these types of problems will always be popping up in our development, and we just need to retrace our steps to see what we missed.

Listing 5.5 Updated docker-compose.yml file

```
postgres:
  build:
    context: ./db
    dockerfile: Dockerfile
  ports:
    - 5432:5432                    <-- Exposes the standard PostgreSQL port
  env_file:
    - ./.sql_server.conf
```

Once we have a basic connection working, we can start writing out the records. We could use a method similar to the one in listing 5.4 where we handle opening and closing the connections manually; however, having to remember to close the connections can be error-prone as we may forget to close them (we have seen files not being closed in a production environment for years only to spring up when moving to a new version of software or another vendor). We will use Python's `with` statement as it automatically handles closing of various resources when a particular block of code is exited. In fact, we have already used it when reading our file, so we can simply expand on that.

Listing 5.6 Updated parse function

```
    def parse(self, filename) -> [List, List]:

        with open(filename, "r") as file, self.get_db() as conn:
            lines = file.readlines()
            sequence_number = 0

            for line in lines:
                sequence_number += 1
                line = line.replace("\n", "")

                with conn.cursor() as cursor:
                    cursor.execute(f"INSERT INTO ach_files
➥ (file_name, unparsed_record, sequence_number)
➥ VALUES (%s, %s, %s)", (filename, line, sequence_number))
                    conn.commit()
```

Creates a connection as part of the existing with statement

Initializes our sequence_number to zero

Increments the sequence_number for each record

Creates a cursor using the with statement, inserts, and commits the record

We can rerun our unit test `test_parsing_ach_file.py`, which will run our sample file through our code, and then check CloudBeaver to verify the records have been added. This is a good start: we can store the records in our database, and it should not be much of a stretch to use a similar approach to go into our individual parsing functions and store the data there.

One thing that we need to do is update our unit test to pull the data from the database instead of relying on data that is being returned, because our goal is to store all the data in the database and not return anything other than a status. For now, let's take a look at updating the `pytest` to get the record count from the database.

Listing 5.7 Updated `pytest`

```
import os
import psycopg
import pytest

from ach_processor.AchFileProcessor import AchFileProcessor

POSTGRES_USER = "someuser"
POSTGRES_PASSWORD = "supersecret"

DATABASE_URL = f"dbname={POSTGRES_USER} user={POSTGRES_USER}
➥password={POSTGRES_PASSWORD} host=localhost port=5432"

def get_db():
    conn = psycopg.connect(DATABASE_URL)
    return conn

def test_record_count():
    dir_path = os.path.dirname(os.path.realpath(__file__))
    file_path = os.path.join(dir_path, "data", "sample.ach")
    expected_result = 41
    parser = AchFileProcessor()
    records, exceptions = parser.parse(file_path)
    with get_db() as conn, conn.cursor() as cursor:
        cursor.execute("SELECT COUNT(*) FROM ach_files")
        record_count = cursor.fetchone()[0]
    assert (
        record_count == expected_result
    ), f"Expected {expected_result}, but got {record_count}"
```

We pull the code from the AchFileProcessor that connects to the database. This is a temporary solution and we will eventually need to refactor the code again when more tests need to connect to the database.

Simple query to fetch the count of records stored in ach_files

This test should work when Docker is first brought up, but subsequent tests will fail because the records are being added. So, our second iteration fails with

```
Expected :41
Actual   :82
```

Our the third iteration fails with

```
Expected :41
Actual   :123
```

We need a way to clear the database before each test. Ideally, we want to have a database that only exists for the length of the test, but first, let's see how we may clear the table after each test. We can use a `pytest.fixture` that will perform any setup/teardown for our individual tests.

Listing 5.8 `pytest.fixture` **to setup and teardown our unit test**

```
@pytest.fixture
def setup_teardown_method():
    print("\nsetup test\n")
    yield
    print("\nteardown test\n")
    with get_db() as conn, conn.cursor() as cursor:
        cursor.execute("TRUNCATE ach_files")
…
def test_record_count(setup_teardown_method):
```

Anything preceding yield will execute before the test.

yield to allow the test to execute

Anything following yield will execute after the test.

Gets a connection and cursor, then trunctuates the table to clear it

Includes our fixture in the unit test

With this code, our test will pass repeatedly because the table is cleared after each run. We have worked through our problem of having the database retain records between runs. Of course, we need to be careful. If our unit tests point to a database that is not a development server, we run the risk of wiping out needed data. For this reason, we may want to look at other options such as an in-memory database, mocking, or `pytest-postgresql`.

Dealing with databases

A common way to work with databases for unit testing is to use an in-memory database such as SQLite or H2. This allows us to run a database that exists entirely in memory and therefore can be isolated to our unit testing. The benefits are usually quick execution and the isolation of data. The drawback is that many times there may be functionality that does not exist between our production database and these in-memory databases, which can lead to problems when trying to create a unit test. For instance, SQLite has five data types to define data, while PostgreSQL has over 40 types. This is not to say that one is inherently better than the other—we just highlight the challenges we may face if our tables use the unsupported data types. We may end up fighting unnecessary battles to get our unit tests to run. That is why we should have additional tools and techniques we can use.

Mocking with a tool such as pgmock can also remove the need for a database in your test. In our scenario, we are actually testing whether the data has made it to the database, so mocking does not really provide a viable solution but something to look into later.

`Pytest-postgresql` is a package that helps us manage `postgresql` connections for a `pytest`, which offers the best of both worlds by allowing us to connect to a test database or create/clear/modify tables as part of the test.

(continued)

As our project progresses, we will find that managing our data for tests and keeping the tests isolated becomes increasingly harder. In chapter 10, we eventually start incorporating the Testcontainers package to isolate our tests. This approach will also be beneficial as the infrastructure for the project matures and we begin running our tests as part of a build pipeline.

The ability to store data is necessary in most applications, but as we have seen, it also adds more complexity to our design and coding. In this section, we started small by ensuring that we could connect to the database and by storing our unparsed records, which helped minimize the number of code changes required. As we move forward, the complexity of our application will gradually increase. As we work through parsing and storing our ACH files, we should also keep in mind that we will be retrieving and aggregating data from the database for our ACH dashboard. A way to easily store our ACH files may lend itself to one database structure, while the ACH dashboard may be better served by an alternative structure. Our job is to strike an acceptable balance between those two goals.

5.3 Storing the file header record

The ACH file header record should be the first ACH record we encounter in an ACH file. So, it makes sense that this is the first record we should explore by adding it to the database. We first show how we may approach it using ChatGPT and then how we go through a complete example with GitHub Copilot installed in our IDE.

5.3.1 Using generative AI

Generative AI can help with a lot of boilerplate code that can get repetitive after a while. Depending on your experience level, that boilerplate code may be new to you, and it may be beneficial to go through the process a few times. Once it becomes tedious, that may be a great sign we can start to lean on generative AI tools. For instance, we may prompt ChatGPT with the following general prompt:

CK Please create a Postgres table to store a parsed Nacha File Header Record.

We then received the `CREATE TABLE` statement from ChatGPT that does a great job incorporating `CHAR`, `VARCHAR`, and `NOT NULL`.

Listing 5.9 File header generated by ChatGPT 40

```
CREATE TABLE nacha_file_header (
    id SERIAL PRIMARY KEY,
    record_type_code CHAR(1) NOT NULL,
    priority_code CHAR(2) NOT NULL,
    immediate_destination CHAR(10) NOT NULL,
```

```
    immediate_origin CHAR(10) NOT NULL,
    file_creation_date CHAR(6) NOT NULL,
    file_creation_time CHAR(4),
    file_id_modifier CHAR(1) NOT NULL,
    record_size CHAR(3) NOT NULL,
    blocking_factor CHAR(2) NOT NULL,
    format_code CHAR(1) NOT NULL,
    immediate_destination_name VARCHAR(23),
    immediate_origin_name VARCHAR(23),
    reference_code VARCHAR(8)
);
```

The file_creation_time is an optional field, so it could be NULL.

These fields are also optional in the Nacha standard. Notice how ChatGPT used VARCHAR instead of CHAR since these fields may be padded with spaces.

From personal experience, we prefer to use `VARCHAR` for most fields to avoid unnecessary padding. We have not encountered any meaningful performance effects from using `CHAR` versus `VARCHAR`. Storing ACH records may be one of the areas where using a `CHAR` could make sense since the fixed-length fields will not have any unnecessary space. `CHAR` can often be misused when declared too large, and any unused space is padded.

Out of curiosity, we asked ChatGPT if `CHAR` or `VARCHAR` was more performant in a `Postgres` database. After making a nice comparison between the two, it updated its example (without us asking it to) to use `VARCHAR` instead of `CHAR`! We were okay with this since our preference is to use `VARCHAR`.

5.3.2 Full example

ChatGPT can be of great help if we have a good idea of what we want to accomplish or if we don't mind spending time configuring the prompt. Otherwise, we may want to work through the process aided by Copilot. Figure 5.3 shows the flow we will be using in this section. Here, we update the unit test at the end of our process because this is a relatively short development cycle. If we do not spend too much time coding before testing, we should be fine with working on the unit test after we do a little coding.

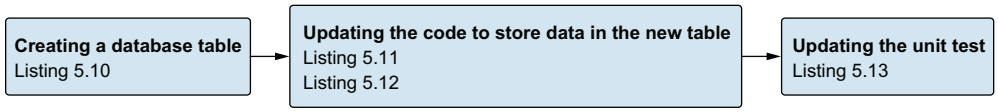


Figure 5.3 Flow and associated code listings for this section

Using our knowledge of storing the records, we should be able to store a parsed record in the database, and once that is complete, the other record formats should fall into place. Recall that we returned the parsed record as a dictionary, as shown in the following listing.

Listing 5.10 A dictionary with a parsed file header record

We use hardcoded values for the offsets because the offsets will not be changing, and we want the ability to quickly refer to fields in question when problems arise.

We strip off extra spaces from fields as necessary.

```
        return {
            "record_type_code": line[0],
            "priority_code": line[1:3],
            "immediate_destination": line[3:13].strip(),
            "immediate_origin": line[13:23].strip(),
            "file_creation_date": line[23:29],
            "file_creation_time": line[29:33],
            "file_id_modifier": line[33],
            "record_size": line[34:37],
            "blocking_factor": line[37:39],
            "format_code": line[39],
            "immediate_destination_name": line[40:63].strip(),
            "immediate_origin_name": line[63:86].strip(),
            "reference_code": line[86:94].strip(),
        }
```

Now we want to take that record and store it in a database instead of simply returning it. We could keep this method of parsing and call it from another routine that was also interested in storing the data in a database. Another approach we may choose is to create dedicated parser classes or utility methods to handle parsing. Any of these approaches would help keep the code clean and reusable. For the sake of simplicity, we are going to take this routine and simply store the data in a database.

First, we want to create a table for storing ACH file headers in the database, and the next listing shows the sample table. At this point, we are going to keep it simple and only supply the fields we need to store the data without referencing any foreign keys or other basic constraints such as `field length` and `NOT NULL`. With Copilot installed, many of these field names were automatically populated, which saved us some time and effort.

Listing 5.11 Table to store ACH file headers

```
CREATE TABLE ach_file_headers (
    ach_file_headers_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    record_type_code VARCHAR(1) NOT NULL,
    priority_code VARCHAR(2) NOT NULL,
    immediate_destination VARCHAR(10) NOT NULL,
    immediate_origin VARCHAR(10) NOT NULL,
    file_creation_date VARCHAR(6) NOT NULL,
    file_creation_time VARCHAR(4),
    file_id_modifier VARCHAR(1) NOT NULL,
    record_size VARCHAR(3) NOT NULL,
    blocking_factor VARCHAR(2) NOT NULL,
    format_code VARCHAR(1) NOT NULL,
    immediate_destination_name VARCHAR(23),
    immediate_origin_name VARCHAR(23),
    reference_code VARCHAR(8)
);
```

Once we have the table, we can move on to updating our code. We chose to pass the connection object (previously created in listing 5.4) into our routine. We could have also stored the connection object as part of our class, but passing it in as a parameter will make it easier to unit test our routine. Different situations may require alternative approaches, so by no means is this the only way to accomplish our task.

Listing 5.12 Passing the connection object

```
match record_type:
    case "1":
        result = self._parse_file_header(conn, line)
    case "5":
```

Now that we have the connection object, we can update the parsing routine to store the data.

Listing 5.13 Updating the `_parse_file_header` routine

```
def _parse_file_header(self, conn: Connection[tuple[Any, ...]],
➥ line: str) -> Dict[str, str]:
    self.expected_record_types = ["5"]

    file_header = {
    ...
    }

    conn.execute(f"INSERT INTO ach_file_headers (ach_file_headers_id, "
➥                       f"record_type_code, priority_code,
➥ immediate_destination, immediate_origin,"
➥                       f"file_creation_date, file_creation_time,
➥file_id_modifier, record_size,"
                        f"blocking_factor, format_code,
➥immediate_destination_name,"
                        f"immediate_origin_name, reference_code) "
                        f"VALUES (DEFAULT, %(record_type_code)s,
%(priority_code)s, %(immediate_destination)s, "
                        f"%(immediate_origin)s, %(file_creation_date)s,
➥%(file_creation_time)s, "
                        f"%(file_id_modifier)s, %(record_size)s,
➥%(blocking_factor)s, %(format_code)s, "
                        f"%(immediate_destination_name)s,
➥%(immediate_origin_name)s, %(reference_code)s)"
➥                             , file_header)

        return file_header
```

We add a connection parameter.

The return statement becomes a variable named file_header.

We can execute directly on the connection to insert records.

We used named variables as placeholders.

The file_header variable passes our values in.

We should now be able to insert our file header records into the database. Next, we can go back and update the unit test.

Listing 5.14 Updated unit test example

```
import pytest
import psycopg
from psycopg.rows import dict_row
from typing import Dict
from ach_processor.AchFileProcessor import AchFileProcessor

POSTGRES_USER = "someuser"
POSTGRES_PASSWORD = "supersecret"

DATABASE_URL = f"dbname={POSTGRES_USER} user={POSTGRES_USER}
➥ password={POSTGRES_PASSWORD} host=localhost port=5432"

@pytest.fixture
def setup_teardown_method():
    print("\nsetup test\n")
    yield
    print("\nteardown test\n")
    with get_db() as conn:
        conn.execute("TRUNCATE ach_file_headers")

def get_db(row_factory = None):
    conn = psycopg.connect(DATABASE_URL, row_factory=row_factory)
    return conn

def test_parse_file_header(setup_teardown_method):
    sample_header = "101 267084131 6910001340402200830A094101DEST NAME
➥              ORIGIN NAME            XXXXXXXX"

    expected_result: Dict[str:str] = {
…
    }

    parser = AchFileProcessor()
    with get_db() as conn:
        result = parser._parse_file_header(conn, sample_header)

    with get_db(dict_row) as conn:
        actual_result = conn.execute("SELECT * FROM
➥ ach_file_headers").fetchone()
        del actual_result["ach_file_headers_id"]

    assert result == expected_result, f"Expected {expected_result},
➥ but got {result}"
    assert actual_result == expected_result,
➥ f"Expected {expected_result}, but got {actual_result}"
```

Creates another get_db function that takes a row_factory parameter

Uses the new parameter in the connect method

Expected result is the same as before.

Leaves the current parse result in place

Gets another connection specifying the result should be returned as a dictionary

Compares the two results

Removes ach_file_headers_id from the returned result

Returns the result; actual_result will be a dictionary.

Here, we duplicated some of our code, such as the `get_db` method with a new parameter. It is important that as we move through the code, we keep an eye on this type of duplication and consider pulling out these methods into utility or helper classes whenever possible. The IDEs by JetBrains (and others) will often point to duplicated code and provide automated options to extract code into functions.

We also left the original result comparison in place because the method is still returning the records. As we continue to improve the project with additional functionality, we will likely remove this in favor of other return information (such as whether the record was parsed). We should now understand how the parsed ACH records can be stored in the database. For the ACH dashboard, to provide a meaningful interface with aggregated ACH data, we need to be able to store all the parsed ACH information in the database. In the next sections, we will explore how the code evolves as we work with it, not only expanding the features, but also addressing some nonfunctional requirements such as maintainability and testability.

5.4 Storing the rest of our ACH records

We have now created a database and stored data in two separate tables, so we should have enough of a framework to store the data for the remaining records in the database. The process is the same for all the remaining tables:

- Create the table and restart Docker.
- Update the parse routine.
- Update the unit test and verify your results.

If embracing the test-driven development approach, we can swap the updating of the parse routine and the unit test. Either way, we should be working in short feedback cycles, allowing any errors to shake out early in the process and not after we have implemented a dozen tables. We also find opportunities for cleaning up the code and making it better overall.

Given the presumed structure of our database, this might be the perfect place to add a few tables. Check whether the database structure makes sense—is there anything that should be changed?

The following section will go over some of the lessons learned and insights gained from working through the rest of the tables that had to be added.

5.4.1 Storing ACH files challenge: Lessons learned

While adding the additional tables, we should have made some observations about the code and taken opportunities to make improvements to our code. Were there any that stood out? It is important to note and resolve possible pain points such as duplicated code, inefficient processing, confusing logic, and so forth. Sometimes, we can really get a sense of accomplishment by cleaning up code to be more straightforward and maintainable. We will go through some of the problems we encountered when adding additional database tables.

With our use of `psycopg3` (which is confusingly defined as `psycopg`) instead of `psycopg2`, we found that our generative AI tools tended not to take advantage of some newer enhancements. For instance, GitHub Copilot insisted on declaring `cursor` methods at first but seemed to learn our preference for using the connection, and after a while, it stopped offering them. This makes sense as Copilot is supposed to learn and adapt to our style and coding preferences. In recent releases of these tools, we have also seen the inclusion of retrieval-augmented generation (RAG), which helps large language models (LLMs) such as ChatGPT stay current with up-to-date information. Of course, we will have to see how they perform over the long term since a lot of the training data would not have used the newer features.

When creating tables, GitHub Copilot did a good job naming the columns, and they matched up with the documented field names in most instances, with only a few minor exceptions. This was helpful since we had already created the field names in our Python code to match the NACHA standards, and because we were able to use named parameters in our SQL queries, it was a breeze to transition from returning records to writing out to the database. Another feature that we were continually impressed with (although it did not always work) was the ability of Copilot to assist in writing our unit tests. We had created unit tests for each record that needed to be parsed, and when we broke it down into a dictionary for comparison purposes, Copilot lent a helpful hand and populated the individual fields with our test data. Even though it was off by a character or two, in some cases, it was definitely helpful overall.

As we went through and built our unit tests, we did eventually create a `SqlUtils` class to handle a few of the repetitive tasks. We started by moving the database parameters such as the user, password, and the URL to the routine. Then later, we expanded this to handle taking a parameter of a row factory so that we could return a dictionary of the values. We also created a routine to truncate the table so that repetitive tests had clean tables to work with. Therefore, our assertions would not fail when we expected a single record but retrieved multiple.

Similar to removing duplication of code with the `SqlUtils` class, we also removed hard coding of the table names throughout the unit tests by creating a variable to hold the table name and using Python f-strings to create the SQL commands where necessary (but not for parameter passing). It is important to note that we still used parameterized queries whenever possible, even though we consider this internal tooling and would not expect malicious code to be entered. We did, however, consider the possibility of checking the passed table names against the database information schema to ensure they were valid. However, that seemed a bit overkill for internal tooling.

We were bitten several times by neglecting to put the setup/teardown `pytest.fixture` into the unit test methods as we were coding. This often led to errors when repeatedly running our tests since the database was not always in a clean state. It happened often enough that we considered creating a class hierarchy that would incorporate the clearing of the table so that we would save ourselves from ourselves. However, it felt too early in our process to add that, so we stayed away from it for the moment.

5.5 Storing exceptions

We should now have the basics of working with Python and PostgreSQL. Having successfully stored our sample.ach file in the database should boost our confidence. However, we should have also noted that our exceptions are not being stored in the database yet. We will want to keep track of those as well. It is common for files to be rejected for various reasons, and ACH processors need to be able to determine whether the file can be fixed manually or a new one should be requested from the originating party. Figure 5.4 shows the flow for this section.

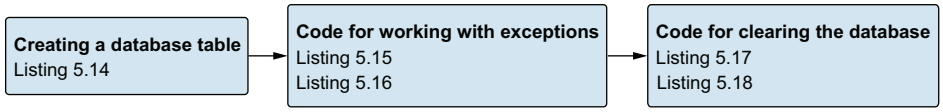


Figure 5.4 Flow and associated code listings for this section

The first order of business is to create the table. We start with a straightforward record that only contains the error description. As our project expands, we may find that we need to break exceptions out by record types, add references to specific records (to help maintain the integrity of the database should records be removed or updated), or implement several other improvements that will become more obvious as we enhance the project. However, those concerns are beyond the current scope of what we need to accomplish, so we will cross that bridge when we come to it in chapter 8.

Listing 5.15 Simple exceptions table

```
CREATE TABLE ach_exceptions (
    ach_exceptions_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    exception_description VARCHAR(255) NOT NULL
);
```

With this table created, we can restart Docker and add a method to our AchFileProcessor.py to insert the record.

Listing 5.16 Simple method to write the table

```
    def _add_exception(self, conn: Connection[tuple[Any, ...]],
➥ exception: str) -> None:
        conn.execute(f"INSERT INTO ach_exceptions
➥ (ach_exceptions_id, exception_description) "
                     f"VALUES (DEFAULT, %(exception)s)",
➥ {"exception": exception} )
        return
```

Simple method passing a connection and an exception string

Inserts the string into the table

With this task completed, we can replace the array we used to hold our exceptions with a single call. We also need to update the various test cases that covered these exceptions, such as an incorrect addenda indicator, invalid record lengths, and ensuring the trace numbers are ascending. Updating these unit tests provides additional opportunities for creating maintainable code.

We removed the passing of our exceptions back to the calling routine, so code such as `records, exceptions = parser.parse(file_path)` becomes simply `records = parser.parse(file_path)`. However, this change made us immediately retrieve the exceptions because our unit tests were validating the number of exceptions and exception message text. We chose to add another method to `SqlUtils` to handle this.

Listing 5.17 Method to retrieve ACH exceptions

```
    def get_exceptions() -> list:
        with SqlUtils.get_db() as conn:
            exceptions = conn.execute(f"SELECT exception_description
➥ FROM ach_exceptions").fetchall()
        flattened_list = [item[0] for item in exceptions]
        return flattened_list
```

Used with keyword to get a database connection

Grabs all exceptions in a single execute command

Returns a list of exceptions instead of tuples

With the helper function in place, we can now return the exceptions with `exceptions = SqlUtils.get_exceptions()`, and the existing logic for the unit test should work without any modifications.

As a result of storing our exceptions in a table, we now have multiple tables that need to be truncated in our unit tests. We could continue to call the `SqlUtils.truncate()` method with the new table. At this point, each test has a maximum of only two tables. However, we would prefer a way to clear all our tables because that will ensure our database is empty for each test. Listing 5.17 showed a truncate-all method that clears our tables. Obviously, this approach should be used with care as we are now truncating all the tables in the database. We could also have dropped and recreated the database; however, this gives us more control since each table is accessed individually. We have worked with projects that have expanded on this type of approach to examine tables for unexpected data, such as determining whether data has unexpectedly been written out to other tables. Of course, our needs will vary from one project to another, and it may be that you do not need or want to truncate the data using this method.

Listing 5.18 SQL to truncate all tables

```
DO $$
   DECLARE
      r RECORD;
```

Creates an anonymous block of code

Declares a variable of type RECORD, a placeholder for a row that has no predefined structure

```
BEGIN
   FOR r IN SELECT table_name FROM information_schema.tables
➥ WHERE #D  table_schema = 'public'
   LOOP
      EXECUTE 'ALTER TABLE ' || quote_ident(r.table_name)
➥ || ' DISABLE TRIGGER ALL';
   END LOOP;

   EXECUTE (
      SELECT 'TRUNCATE TABLE ' || string_agg(
➥quote_ident(table_name), ', ') || ' CASCADE'
        FROM information_schema.tables
       WHERE table_schema = 'public'
   );

   FOR r IN SELECT table_name FROM information_schema.tables
➥ WHERE table_schema = 'public'
   LOOP
      EXECUTE 'ALTER TABLE ' || quote_ident(r.table_name) ||
➥ ' ENABLE TRIGGER ALL';
   END LOOP;
END $$;
```

Signifies the start of a new transaction

Disables any triggers on the table

Gathers a list of tables and truncates them

Enables the triggers again

Commits the current transaction

At this point, we should have arrived at a similar point to where we were with our running code from chapter 3. Recall we are able to perform some basic parsing on some simple ACH files, with the major change being that we are now storing the results of our hard work into a database. We can take a moment to congratulate ourselves, but only for a moment, because while we have the ability to parse the file, we do not have a way for a user to load a file. The next sections look at expanding our API to upload an ACH file and see the cascading effect it has on our database.

5.6 Uploading an ACH file

We may consider this our most important change yet. Of course, being able to parse a file is necessary, but it is often allowing interaction with the user that feels the most rewarding for many developers. This is possibly true because being able to work and interact with our project just feels like an accomplishment. And we feel that unit testing gives us much the same rewarding experience, which is why we like testing so much! Figure 5.5 shows the flow for this section.

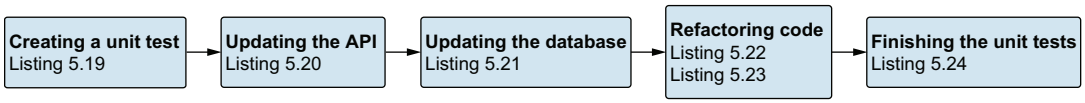


Figure 5.5 Flow and associated code listings for this section

We have previously built a basic API using hardcoded values. We will be taking that code and adding the ability to upload an ACH file. From there, we'll be expanding the APIs to retrieve the data from our database instead of the hardcoded values. In the original ACH dashboard, one of the problems was a lack of control in processing ACH files.

Before creating a new API, let's just ensure we can unit-test one of our existing endpoints with a hardcoded value. The next listing shows what that looks like. Copilot was able to produce the majority of this code as we typed, so it is just a matter of making sure it does what we intended.

Listing 5.19 A unit test for FastAPI

```
from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)

def test_read_files():
    response = client.get("/api/v1/files")
    assert response.status_code == 200
    assert response.json() == [{"file": "File_1"},
➥ {"file": "File_2"}]
```

Imports the TestClient needed for unit testing

Imports our application

Defines a client

Defines a method for testing

Calls our endpoint and saves the response

Assert statements validate the response

So, that was pretty easy. Now, we would like to work on being able to `POST` an ACH file to the backend, get a response that it has been uploaded, and then begin parsing the file. Why not parse the file and then return a response to the user? The processing of our ACH file may take some time, especially when we consider that it will eventually need to interact with other systems to execute tasks such as verifying accounts/balances, OFAC validation, fraud analysis, and any other tasks an ACH processor may want to perform. Rather than have a user wait, we can verify the file is uploaded and start a task to perform the parsing.

We already have an endpoint for posting a file, so we can add a unit test and then upload the file. Since the file is going to be uploaded using an `HTTP POST` request, it will be encoded using `multipart-formdata`. This means that we need the package `python-multipart` because this is a requirement for parsing requests encoded with that format. If we do not have it installed, we will receive a friendly reminder (in the form of an error).

Listing 5.20 `python-multipart` not installed

```
E   RuntimeError: Form data requires "python-multipart" to be installed.
E   You can install "python-multipart" with:
E
E   pip install python-multipart
```

Creating a test for uploading should look something like the following listing.

Listing 5.21 Unit test for uploading a file

```
def test_upload_file():
    with open("data/sample.ach", "rb") as test_file:
        response = client.post("/api/v1/files",
                   files={"file": test_file})
    assert response.status_code == 201
```

Opens the sample file

Uses the client to POST the file

Ensures we receive a 201 status

We will also need to update the endpoint to receive the file. There is no need to return anything as we are just interested in the status code.

Listing 5.22 Updated endpoint to receive the file

```
from fastapi import APIRouter, Request, status,
➥ File, UploadFile
@router.post("", status_code=status.HTTP_201_CREATED)
async def create_file(file: UploadFile = File(...)):
    return None
```

Additional imports are needed.

Defines the file as UploadFile

Returns nothing; only the status code as necessary

Now we should have a successful unit test, and the real work can begin. Uploading a file manually or through some automatic process should be the driving force behind all our ACH tables. We should store some file information such as filename, time uploaded, a file hash (for tracking of duplicate files), and whatever other information we may decide is needed. The UUID of this record should then be included in any child tables (all the previous tables we have just created). If we had more experience or worked through the problem in a different order (perhaps uploading a file first), we may have avoided having more rework, but we would have also had to introduce more database concepts to start with. A benefit of this approach is the actual rework required to incorporate the changes. Often, developers become paralyzed because of the fear of change. Large complex systems sometimes seem like a house of cards where one wrong move can cause everything to come tumbling down. Having good unit test coverage and confidence in the tests can go a long way to alleviate these fears. The fear of changing and improving working software eventually leads to code rot.

We will repurpose our `ach_files` table to be our main table that contains the upload information and rename `ach_files` to `ach_records` so that its only job is to store the unparsed ach records. The updated table definitions are in the following listing.

Listing 5.23 Updated table listings for `ach_files` and `ach_records`

```
CREATE TABLE ach_files (
    ach_files_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    file_name VARCHAR(255) NOT NULL,
    file_hash VARCHAR(32) NOT NULL,
```

The ach_files table is duplicated and repurposed to store the upload details.

```
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
);

CREATE TABLE ach_records (
    ach_records_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    ach_files_id UUID NOT NULL REFERENCES ach_files(ach_files_id)
➥ ON DELETE CASCADE ON UPDATE CASCADE,
    file_name VARCHAR(255) NOT NULL,
    unparsed_record VARCHAR(94) NOT NULL,
    sequence_number INTEGER NOT NULL
);
```

The ach_files table is duplicated and repurposed to store the upload details.

The former ach_files will become ach_records.

Creates a foreign key named ach_files_id that references the ach_files_id from ach_files

Removes the file_name as it is now stored in the ach_files table

The new table `ach_records` uses the keywords `REFERENCES`, `ON DELETE`, and `ON UPDATE` to create a foreign key to the `ach_files` table. This feature allows the database to maintain its referential integrity. For instance, when we delete an ACH file out of `ach_files`, we do not want to go into every table and delete associated data. Instead, we want to define a relationship between the tables where all associated data is removed if we were to delete the `ach_file`. Once we have completed updating our tables, we can see this in action. This will also affect our tests. Once we have implemented referential integrity, we will need to ensure the `FOREIGN KEY` constraints are maintained.

For example, if we want to have a unit test write out records to the `ach_records` table. we need a valid `ach_files_id` (it must exist in the `ach_files` table). For this reason, we will likely be looking at expanding the `SqlUtils` class we had developed earlier to set up some generic records and make this easier. Maintaining referential integrity can mean extra work on our side when setting up for testing, but it is worth implementing. We have worked in systems with the benefit of referential integrity and have seen programs crash or loop because of incomplete relationships. Usually, a lack of referential integrity is apparent in a system where developers have written various utility programs to scan and fix the data (which we have had to do ourselves on more than one occasion).

Additionally, we want to store the MD5 (or whichever algorithm you prefer) hash of the ACH file so we to identify duplicate files. First, we can get the hash from the command line using `Get-FileHash -Path ".\sample.ach" -Algorithm MD5 | Select -Object -ExpandProperty`, which in our case prints out `18F3C08227C3A60D418B3772213A94E3`. We'll keep this info handy because we'll be storing the hash as computed with Python in the database and expect it to be the same.

Interestingly enough, when coding the function, Copilot prompted us to use string interpolation for the SQL query, `f"INSERT INTO ach_files (file_name, md5_hash) VALUES ('{file.filename}', '{md5_hash}')`, but we continue with our standard parameterized query because we want to maintain secure coding practices. Queries that use string interpolation are ripe for SQL injection (https://mng.bz/QDNj). Our code for the `/api/v1/files` endpoint is shown in the next listing.

Listing 5.24 Updated endpoint to upload ACH files

```
@router.post("", status_code=status.HTTP_201_CREATED)
async def create_file(file: UploadFile = File(...)):
    contents = await file.read()
    md5_hash  = hashlib.md5(contents).hexdigest()
    with DbUtils.get_db_connection() as conn:
        conn.execute(
            f"""
            INSERT INTO ach_files (file_name, file_hash)
            VALUES (%s, %s)
            """, (file.filename, md5_hash)
        )
    return None
```

Reads the uploaded file

Uses hashlib to create a hash of the file

Uses our newly created DbUtils to return a connection

Simple SQL to INSERT a record into ach_files

We created a simple `DbUtils` class to return a database connection. We imagine it will be used in multiple places in our project and thus extracted it into its own file.

Listing 5.25 `DbUtils` for creating a database connection

```
import psycopg

POSTGRES_USER = "someuser"
POSTGRES_PASSWORD = "supersecret"

DATABASE_URL = f"dbname={POSTGRES_USER} user={POSTGRES_USER}
➥   password={POSTGRES_PASSWORD} host=localhost port=5432"

def get_db_connection(row_factory = None):
    conn = psycopg.connect(DATABASE_URL,
➥row_factory=row_factory)
    return conn
```

Hardcoded values to be removed later

Creates and returns a connection

With these changes, we should be able to run the unit test and then head over to our CloudBeaver UI and see the newly added record and manually compare the record hash. However, manually checking the record is no fun, so as a final piece, we should update our unit test to incorporate the same `SqlUtils`, as we have seen previously, and ensure that we have at least one record in our table. The following listing shows the updated unit test.

Listing 5.26 Updated unit test for our upload file

```
…
from tests.SqlUtils import SqlUtils
import pytest
…
@pytest.fixture
def setup_teardown_method():
    yield
    SqlUtils.truncate_all()
```

Required imports

Defines our fixture to ensure all tables are truncated

```
…
def test_upload_file(setup_teardown_method):
    with open("data/sample.ach", "rb") as test_file:
        response = client.post("/api/v1/files", files={"file": test_file})
    assert response.status_code == 201
    assert SqlUtils.get_row_count_of_1('ach_files')
```

Includes the fixture in our unit test

Ensures there is one row in the ach_files table

Being able to upload a file successfully may not seem like a big deal, but we must remember that all other functionality of our ACH dashboard is based on this one piece. Functionality such as error checking and recovery, crediting/debiting accounts, fraud analysis, and much more is available to us now that we can upload ACH files. After completing this task, we can expand on the database structure.

5.7 Storing records with integrity

As mentioned before, we want our database to have referential integrity, which can simplify navigation in database tools, and more importantly, have the ability to prevent dangling records. We have seen that databases without referential integrity require maintenance programs that run periodically to clean up those dangling records, as well as infinite loops and program crashes resulting from incomplete data. A great benefit of using relational databases is their support for data integrity. We highly recommend keeping it in mind when designing databases.

In our examples so far, we have defined a `PRIMARY KEY` and how to update the `ach_records` table. In listing 5.23, we added our first `FOREIGN KEY` by using the `REFERENCES` keyword, which starts us on the path of maintaining referential integrity. We continue to update the remaining tables and code with foreign keys and review the effects of using them on our tables, code, and unit tests.

If we try running our unit tests at this point, we will see errors referring to an `UndefinedColumn` since we updated our table definitions in listing 5.23. This is not a bad thing. The fact that we have built unit tests allows us to refactor our code with confidence. We start with diving into our `test_record_count` unit test (from listing 5.7) to resolve our error of `psycopg.errors.UndefinedColumn: column "unparsed_record" of relation "ach_files" does not exist`. The offending code is shown in the following listing.

Listing 5.27 Invalid column names

```
conn.execute("""
                INSERT INTO ach_files
[CA] (file_name, unparsed_record, sequence_number)
                VALUES (%s, %s, %s)
                """, (os.path.basename(filename), line, sequence_number))
```

Depending on your IDE and setup, this statement may already be flagged. In PyCharm, we have defined a connection to the database, so the fields `unparsed_record` and

`sequence_number` are flagged with an `Unable to resolve column` error. If your IDE does not have that capability, then the stack trace also shows the line number. Given this information, we notice that the `INSERT` statement is pointing to the wrong table as we had renamed `ach_files` to `ach_records`. Changing that and rerunning the test gives a new error `psycopg.errors.NotNullViolation: null value in column "ach_files_id" of relation "ach_records" violates not-null constraint`. We defined `ach_files_id` as a foreign key, and it cannot be null—in fact, it must point to a valid record in the `ach_files` table. Recall that our API will create this record when a file is uploaded. So, it is plausible that the `AchFileProcessor` will now need to be called with both the filename being parsed and a valid `ach_files_id`. We can update the unit test so that the parse routine is called with this ID. The `parser.parse(file_path)` will need to become `parser.parse(ach_files_id, file_path)`, but it is not enough to define `ach_files_id`. We need a way to create a valid `ach_files_id` because we have to maintain database integrity. We could drop the constraints during testing, which may be an option if our unit tests were primarily concerned with functionality. However, in this case, we would like to maintain the constraint. So, we will need to create a function to insert the record, then take a step back and consider our path forward as there may be a few different routes we want to take.

Remember that we already coded an `INSERT` statement to insert the `ach_files` record into our API. We could duplicate that query in our unit test and then grab that inserted value. That would work, but we will have duplicated code, and we want to try to avoid that whenever possible. We could add a routine to `SqlUtils` to handle this task for us, which other methods would have access to because other unit tests may also need the functionality. However, `SqlUtils` is meant to assist us in our tests only, and we have already seen that we will need this functionality elsewhere. Perhaps we should create a utility class for the `AchFileProcessor` to insert our desired record, which would also allow us to refactor the existing SQL queries out of the parsing routines. There is no definitive answer. Depending on the project, our needs, programming style, and experience, we may see other alternatives or have preferences on how to approach this problem. In this case, we believe that using Pydantic is the best way to move forward.

5.8 Using Pydantic

In the previous section, we ran into an problem of how to handle inserting a record to make our unit test work. While there are several approaches we could take, Pydantic will help further refactoring and expansion of our project. We had introduced Pydantic previously to help document our APIs in chapter 4, so we know that it already has some benefits. Another benefit is that we will be able to define our tables and fields in a way that allows developers not to have to remember which fields are present. In other words, we start abstracting the SQL from the parsing logic. Let's start applying that with our `ach_files` table so that we can create a record and return a valid record ID, which is all we really wanted to do to get around our first unit test problem.

Why is there no object-relational model framework?

An object-relational model (ORM) has many benefits and is widely used in many different industries. We are holding off on incorporating an ORM such as SQLAlchemy for now because we want to ensure the reader is exposed directly to SQL in case those skills need brushing up on.

Later, we will show how to include an ORM into the project, so hang in there. Of course, if you are familiar with ORMs, you can dive right in and start using them.

We first create a Pydantic model for our table, as shown in the following listing. This simple model provides a basic layout for modeling the data that'll be written to our `ach_files` table.

Listing 5.28 Our Pydantic model for `ach_files`

```
from typing import Optional
from datetime import datetime
from pydantic import BaseModel, UUID4


class AchFileSchema(BaseModel):
    id: Optional[UUID4] = None
    file_name: str
    file_hash: str
    created_at: Optional[datetime] = None
```

Necessary import statements

The class extends the Pydantic BaseModel.

Our field definitions; note the Optional keyword for fields (such as ID) that our database will supply.

Next, we can go ahead and define a class that will provide some basic Create, Read, Update, and Delete (CRUD) methods to handle working with the database table. The following listing shows the `AchFileSql` class we will create to wrap the logic for dealing with our `ach_files` database table.

Listing 5.29 `AchFileSql` class

```
from typing import Optional
from uuid import UUID
from psycopg.rows import dict_row
from ach_processor.database import DbUtils
from ach_processor.schemas.AchFileSchema
➥import AchFileSchema


class AchFileSql:
    def insert_record(self, ach_file: AchFileSchema)
➥ -> UUID:
        with DbUtils.get_db_connection() as conn:
            result = conn.execute(
                """
                INSERT INTO ach_files(ach_files_id,
➥ file_name, file_hash, created_at)
```

Required imports for the class

Creates a function to insert the record and return the UUID

```
                                    VALUES
➥(DEFAULT, %(file_name)s, %(file_hash)s, DEFAULT)
                                    RETURNING ach_files_id
                                     """, ach_file.model_dump())

        return result.fetchone()[0]

    def get_record(self, ach_file_id: UUID)
➥ -> AchFileSchema:
        with DbUtils.get_db_connection(row_factory=class_row(AchFileSchema))
➥ as conn:
            result = conn.execute(
                """
                SELECT * FROM ach_files WHERE ach_files_id = %s
                """, [ach_file_id.hex])

            record = result.fetchone()

            if not record:
                raise KeyError(f"Record with id
➥ {ach_file_id} not found")

            return record
```

Uses the RETURNING keyword to return the ID of the newly inserted record

Uses model_dump to create a dictionary that references the fields

Creates a function to return a specified record

By using the row_factory of class_row, we can return the record directly.

If nothing is found, raises an error

Finally, we create a unit test to verify our newly created classes. We create this unit test as a class just to show another way to help organize our tests. We also introduce the `autouse` option on our `pytest` fixtures so we do not have to include them in every method.

Listing 5.30 Unit testing our new classes

```
import pytest
from tests.SqlUtils import SqlUtils
from ach_processor.database.AchFileSql import AchFileSql
from ach_processor.schemas.AchFileSchema
➥ import AchFileSchema


class TestAchFileSql:
    @pytest.fixture(autouse=True)
    def setup_teardown_method(self):
        print("\nsetup test\n")
        yield
        SqlUtils.truncate_all()

    def test_insert_record(self):
        ach_file_record = AchFileSchema(
            file_name="sample.ach",
            file_hash="1234567890"
        )
```

Required imports

Our fixture now uses autouse.

Truncates the tables when finished

```
        sql = AchFileSql()
        ach_file_id = sql.insert_record(ach_file_record)
        retrieved_record = sql.get_record(ach_file_id)
        assert SqlUtils.get_row_count_of_1("ach_files")
➥ is True, "Expected 1 record"
        assert retrieved_record.file_name == ach_file_record.file_name,
➥ f"Expected {ach_file_record.file_name}, but got
{retrieved_record.file_name}"
```

Inserts the record

Immediately returns the record

Assert statements validate our results

This is a great start to taking our project to the next level. You may be wondering why we did not write our test first in a test-driven development manner. Sometimes, it is easier to take a test-later approach, especially when demonstrating new concepts. Again, it is not so about writing the test but about working in a short cycle. So, as soon as we got something to test, we started testing it.

Recall that with Pydantic, we get field validation and the ability to document our APIs. We will be looking at that later, but for now, we should continue refactoring our code to take advantage of Pydantic. Once we have the code refactored using Pydantic and our unit tests passing, we can move along to handling APIs.

5.9 Lessons learned

The process of including Pydantic and separating our code required us to refactor not only the ach_file_processor.py but our unit tests as well. Refactoring allowed us to improve our code in both areas and obtain code that is cleaner and easier to test. Unfortunately, we also ran into some problems when we finished creating a database structure that followed the original specification from figure 5.1. Did you notice any problems with the structure before refactoring the code?

Inserting foreign keys into that database structure revealed a problem with maintaining data integrity in the `ach_files` table. While deleting a record from those child tables would behave correctly, the `ach_files` table did not get rid of all the records we desired. For instance, if we were to delete a batch, we would expect the associated type 6–8 records to be removed as well, and this cannot happen in the current structure.

This situation is not uncommon. Often, when given specs for a project—depending on the experience of the project stakeholders and the time allowed for researching areas of the project—it may not be possible to flush out all the design details. In a scaled Agile context, we may consider this project an enabler—more specifically, an *exploration enabler*. Exploration enablers help explore prospective solutions and include activities such as research and prototyping. If these are not completed sufficiently before we got a project, we may run into problems we had here. There are several alternatives to handle the situation, and the right answer will likely depend on the reevaluation of project requirements.

To recap the project requirements, we wanted to find a solution that

- Provided both ETL and ELT options when processing a database.
- Provided data integrity over the entire file. As the ACH file is a hierarchy, there are multiple scenarios that require additional records to be cleaned up when a file, batch, or entry are deleted.

Let's propose a few different options we may need to present to the business stakeholders.

First, we could ditch parsing the individual records into the database and just store the unparsed records. This would certainly simplify the database design as we only have one table to deal with. Of course, this approach limits the usefulness of having a relational database and would require additional application code to handle the data integrity we mentioned earlier. For instance, if the user wanted to delete a batch, we would have to ensure that all the records were deleted by the application code, which would expand and complicate our application code.

Second, we could ditch storing unparsed records in the table. This could potentially be a good solution if we determine we do not need the unparsed records. Of course, it also means that our file needs to conform to the database constraints, and the business has already required the unparsed records to be retained in the event of invalid data (such as non-numeric data in a numeric field) that may cause a parsed record to be rejected from the database. This seems to be a hard requirement to get around.

Third, we could investigate setting up a *database trigger* to delete the records. Database triggers are code that can be automatically executed when certain events occur in the database. We may be able to create triggers in the parsed record tables that execute when a record is deleted to also eliminate the associated unparsed record. That does not sound like much fun though.

We opted for another route to address this problem—reorganizing the table into the structure where we used an unparsed record table for each type. This required substantial refactoring of the tables and associated application code, but for us, it made the most sense if we wanted to maintain the requirements for keeping unparsed and parsed records. The updated database diagram is shown in figure 5.6, along with a reference to the original layout from figure 5.1.

This structure gave us the data integrity we were originally looking for, but we now have to find a way to view all the unparsed records. To accomplish this, we create a database view. A *database view* is a virtual table that is the result of a stored query. By using a view, we can save ourselves and others who need to use our database the tedious task of tying the unparsed data together. The next listing shows the created database view.

Listing 5.31 Creating a database view

```
CREATE VIEW combined_ach_records AS
SELECT
    r1.ach_records_type_1_id AS primary_key,
    r1.unparsed_record,
    r1.sequence_number,
    r1.ach_files_id
```

```
FROM
    ach_records_type_1 AS r1

UNION ALL

SELECT
    r5.ach_records_type_5_id,
    r5.unparsed_record,
    r5.sequence_number,
    r1_r5.ach_files_id
FROM
    ach_records_type_5 AS r5
JOIN ach_records_type_1 AS r1_r5
    USING (ach_records_type_1_id)


…
UNION ALL

SELECT
       r9.ach_records_type_9_id,
       r9.unparsed_record,
       r9.sequence_number,
       r1_r9.ach_files_id
  FROM
       ach_records_type_9 AS r9
  JOIN ach_records_type_1 AS r1_r9
 USING (ach_records_type_1_id)
```

UNION ALL will combine the results from the successive SELECT statements.

The USING statements allow for a more concise syntax instead of ON table.field =.table.field. It can be used in PostgreSQL when the joining fields share the same name among tables.

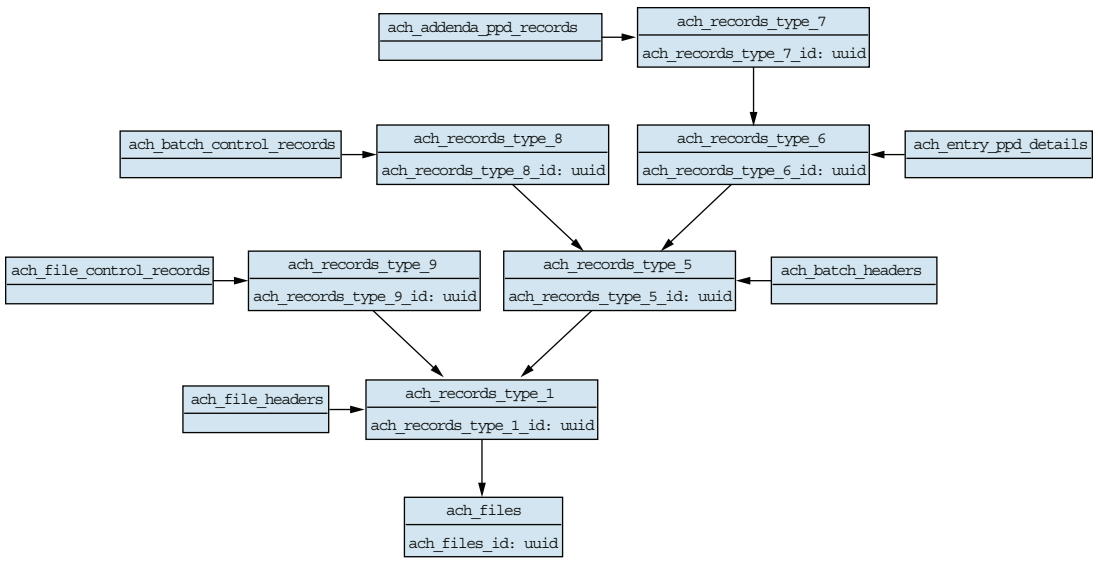


Figure 5.6 Updated table diagram

This code allows us to view the entire ACH file as it was uploaded to the system, without any manipulation from our processing. Having a way to view the raw file allows us to present users with options to view the entire ACH file contents, as well as exception records to help them troubleshoot any problems.

5.10 Coding changes

Now that we have settled on this database structure, it was a matter of writing any additional unit tests and getting the existing ones updated to run correctly. Let's look at some of the changes made to support this structure.

5.10.1 Creating Pydantic schema for the unparsed ACH records

All the unparsed records share the `unparsed_record` and sequence number fields. Therefore, this is a good opportunity to create a class structure that will inherit those fields so that we do not have to type them every time. We create an `AchRecordBaseSchema`.

Listing 5.32 Base schema for our ACH records

```
from abc import ABC
from pydantic import BaseModel

class AchRecordBaseSchema(ABC, BaseModel):
    unparsed_record: str
    sequence_number: int
```

Importing ABC allows us to create an abstract class.

The BaseModel is required for Pydantic.

Our class inherits from both ABC and BaseModel.

Fields that will be present in all ACH record classes

With that schema, we can define each record type, as shown in the following listing.

Listing 5.33 ACH record schema

```
from typing import Optional
from pydantic import UUID4

from ach_processor.schemas.ach_record.ach_record_base_schema import
     AchRecordBaseSchema

class AchRecordType1Schema(AchRecordBaseSchema):
    ach_records_type_1_id: Optional[UUID4] = None
    ach_files_id: UUID4
```

Required imports

This is a Pydantic class by virtue of being a subclass of AchRecordBaseSchema, which was a subclass of BaseModel.

We have an ID field that is marked as optional since it will be assigned by the database.

The ach_files_id field is passed in and is not optional as it is a foreign key to the file that was uploaded.

5.10.2 Creating Pydantic schema for the parsed ACH records

The Pydantic definitions for each of the parsed records are less interesting at this point as we just inherit from the Pydantic `BaseModel` record and define the necessary fields.

We will expand on the fields for constraints, validation, and documentation later. For now, we just keep them simple.

Listing 5.34 Pydantic schema for an ACH batch header record

```
from pydantic import BaseModel, UUID4


class AchBatchHeaderSchema(BaseModel):
    ach_records_type_5_id: UUID4
    record_type_code: str
    service_class_code: str
    company_name: str
    company_discretionary_data: str
    company_identification: str
    standard_entry_class_code: str
    company_entry_description: str
    company_descriptive_date: str
    effective_entry_date: str
    settlement_date: str
    originator_status_code: str
    originating_dfi_identification: str
    batch_number: str
```

5.10.3 Unit test changes

As part of code refactoring, we ensured that our tests were cleaned up as well (listing 5.35). First, we updated our `setup_teardown_method` to set `autouse` to `true` and then ensure the `SqlUtils.truncate_all` method was executed first. We may have previously chosen to clear the tables after the tests were run, which is a good practice that cleans up any data from our test. However, it also has the unfortunate side effect of cleaning up the data when tests fail, which is not very helpful when we want to examine the database after a test. To make debugging and troubleshooting easier, we decided to clear the data before the test. This also ensures the database is ready as we do not have to rely on a previous test to clean up after itself. Adding the `autouse` parameters means we no longer need to pass the fixture to our tests. We also used `truncate_all` instead of a specific table since we now have multiple tables being used.

Listing 5.35 Updated `pytest` fixture

```
@pytest.fixture(autouse=True)
def setup_teardown_method():
    SqlUtils.truncate_all()
    yield
```

5.11 Design and different approaches

It was a fair amount of rework to add foreign keys to the database, so we may be thinking about ways to minimize that work or why we did not just add them in the beginning. As

you saw, we dove into parsing the ACH file, which we accomplished, and then moved on to supporting some additional functionality that required a lot of rework. This was partly because we wanted to show some ACH basics before getting into overall functionality and additional database knowledge.

However, let's consider whether we approached the problem from a functionality standpoint and whether the user was going to need these files uploaded. Had we then started with the `ach_files` table and associated endpoint, we could have incorporated foreign keys from the beginning. Assuming you had knowledge and experience working with ACH files and APIs, it would certainly have been a valid approach. Then we could have proceeded in much the same fashion, with the exception that we may have had a better-designed database from the start.

This just goes to show that initial design can be important as well as the approach to the problem you take. We spoke earlier about enabler stories, or what is also known as research spikes. They reduce risk and help us understand requirements and otherwise gain a better understanding of the tasks that will need to be performed. We cannot stress the importance of these types of stories, especially when working with large complex systems. We can always expect some rework to be required when we get into a project, either due to unidentified requirements or changes in scope that could not be accounted for. Hopefully, research spikes help minimize those instances, but it can often be difficult for a business to devote time to something they do not immediately see a benefit from. This will obviously become a problem because problems that are flushed out by enabler stories will not be identified and will go unnoticed in PI planning. The approach we may end up taking may require rework that could have been identified earlier, and therefore, our story points could have been allocated correctly.

We have seen numerous examples where a project is seemingly complete, only to have a subject matter expert point out something that was missed in a system demo and requires rework. It is important to remember that we will have to deal with these types of situations at times as it may be often tempting to take an easier way out. In such situations, we will need to judge the effects of the change on the project timeline, amount of rework, risk involved, technical debt incurred, and so forth. For instance, the business may have decided that we did not need referential integrity of the database and that it would be faster to write a program that could be manually run to check for dangling records/missing relationships.

Summary

- This chapter shows how to create a database that could store our unaltered records, as well as a parsed version of the record that will be beneficial when necessary to provide details on the loaded ACH files.
- We saw the implications of adding referential integrity to our database late in the process and the need to rework both our code and unit tests. When a feature requires rework to be done correctly, it can often be put on the back burner.

- Despite the extra work to implement these types of features, it is important to champion them both to the team members and management so they are not left by the wayside.
- Defining databases is crucial for data storage, querying, and integrity in applications.
- ACH files face challenges with performance, querying, and data integrity when handled as flat files.
- Relational databases offer advantages, such as primary and foreign keys, constraints, and data integrity.
- Implementing referential integrity prevents orphan records and ensures database consistency.
- A research spike (enabler story) is beneficial when evaluating database design and various implementation approaches.
- ELT and ETL offer different benefits for processing ACH files and handling errors.
- Pydantic helps in modeling database tables, abstracting SQL, and enhancing documentation and validation.
- Uploading files and integrating APIs are foundational for expanding functionality in an ACH system.
- Data and referential integrity are critical for relational databases to prevent errors and improve reliability.
- Continuous testing, refactoring, and revisiting initial design choices help maintain and improve database performance and structure.

6 Taking the next step with Next.js

This chapter covers

- Using Next.js to build a dashboard
- Using Material UI components
- Using WireMock to help stub our API calls

Until this point, we have worked through research spikes to explore creating and parsing an ACH file, creating a database, and generating APIs to access our data. In this final research spike, we'll start working on our application dashboard. A dashboard is the user interface (UI) from which it is possible to upload and view files, view records, and obtain any other useful information. While there are plenty of options available for creating a UI, we have decided to go with Next.js as it is built on React, which is also very popular. However, since everything is powered by APIs, the UI could be swapped with Remix, Vue, Angular, or any other technology used in our company.

Chapter 4 showed us how to take an API-driven design approach. While we have not built all our APIs by switching to the UI yet, we will begin to see what data we want (or need) to be returned as part of our APIs. By stubbing out the data fields, we can ensure our UI is making exactly the same calls required for the real APIs.

6.1 Getting started with Next.js

Getting started with Next.js is very easy, and we chose it partly because it is easy to get an application up and running, but also because it can grow to support any application we care to create. Next.js is a full stack React framework. React itself is a library used for building web interfaces. Although it is simple to learn, it does require a basic understanding of some common web technologies such as

- *Hypertext Markup Language (HTML)*—The standard markup language for building web pages
- *Cascading Style Sheets (CSS)*—A way to describe how HTML elements should be displayed
- *Document Object Model (DOM)*—Defines properties, methods, and events for HTML elements and allows languages such as JavaScript to add/change/remove HTML elements and attributes, as well as react to HTML events

React builds and improves on the listed web technologies by providing additional concepts to facilitate UI development. Some key concepts of React are

- *Components*—Reusable and self-contained pieces of code that render parts of the UI. They are the building blocks of our application.
- *State*—Holds dynamic information within a component, which can be used to change its behavior and trigger rendering.
- *Hooks*—Help manage the state and lifecycle of components.

All of this may seem like a lot to digest, especially if you are new to UI development. The truth is that you should still be able to walk through this research spike and gain a basic understanding of these concepts.

So, how does Next.js fit into all of this? The React web site (https://react.dev) recommends using Next.js (or Remix) because it is a full-featured production-grade React framework. That means that Next.js comes with a lot of powerful features, such as

- File-system-based routing that supports multiple layouts, nested routing, and error handling
- Client-side rendering (CSR) and server-side rendering (SSR), depending on the needs of the application
- Data fetching with `async/await`
- Support for multiple styling methods such as CSS Modules, Tailwind CSS, and CSS-in-JS
- Optimization for images, fonts, and scripts

To get started, we have to ensure that we have Node.js installed (https://nodejs.org/en) because we need access to its tools, beginning with the `npx` command. `npx` is a command line tool that helps simplify the running of `npm` (Node Package Manager) commands.

We can execute `npx create-next-app@latest` to answer a few questions and then start the project with `npm run dev`. We should be able now to log in to http://localhost:3000 and see whether the default page comes up. Of course, the default page is a test to make sure we have our setup correctly. We want to develop a UI that allows a user to easily load files and review associated data. Usually, the business already has a design in mind that is based on existing standards. As we are not required to follow any particular design, a good option is to start using a sample template. There are plenty of options out there, but a great way to get started is by using the dashboard template provided by Material UI at https://mng.bz/Xx66. We pulled the template for MUI 5.x, which can be viewed at https://mng.bz/yWnq. As previously mentioned, we are not limited to this particular template, but it does provide us with a basic layout that we can quickly customize to create our final project.

We recommend grabbing the TypeScript versions of the template as we will be using TypeScript throughout the project. TypeScript is a superset of JavaScript, which means that your existing JavaScript skills can be employed when using TypeScript. The benefit of TypeScript is that it adds explicit types to JavaScript (among other things). We believe that the explicit types help with the maintainability of the code over the long term by helping catch errors earlier during development. The long-term maintainability of the code is very important, especially when we consider how long our code may be out in the field, and given FinTech, that can be a long time! While we will get the basics of TypeScripts within this project, *Essential TypeScript 5* (2023, Manning) by Adam Freeman is a great resource, or you can check out the TypeScript tutorial available at https://www.typescripttutorial.net/.

Before we can get the project started with the sample template, we need to ensure the dependencies are installed, which can be done with the command line in the next listing. This command installs a few requirements for Material UI, as well as for `recharts` (for our visualization needs) and `axios` (for making API calls). These packages were chosen because we have seen them in use in FinTech, and they have solid documentation to help us if we get stuck.

Listing 6.1 Installing dependencies for our dashboard

```
npm install @mui/icons-material @mui/material @emotion/styled
@emotion/react recharts axios
```

Installs all the components needed for our dashboard

With that complete, the project should be up and running. Now we only need to customize the dashboard and add additional functionality.

One of the ways we like to begin with new code is to dive right into it and start adding additional functionality or refactoring existing code. Let's jump right in!

6.2 Creating an upload page

To help you get started, Next.js has extensive documentation and a great tutorial (https://nextjs.org/learn) if you are completely new to React and web development. We begin with Next.js routing as we find it straightforward and just need to create a simple structure on our filesystem. We can easily create a page that will handle our ACH file uploads, as shown in the next listing. The page.tsx is key to having our web page displayed. TSX is a syntax extension for TypeScript that lets us write HTML-like markup inside the Typescript file. You may also see JSX pages in various tutorials, which are just the JavaScript equivalents.

Listing 6.2 Our directory structure for the upload page

```
src/
├─ app/
|  ├─ uploads/
|  |  ├─ page.tsx
```

This file-based routing will cause Next.js to display page.tsx when we navigate to /uploads.

We want to create a simple page that allows us to drag and drop files onto the page to be uploaded. We will stay away from the actual file uploading for now to focus on some of the design first. Given the structure from listing 6.2, we want to be able to navigate to http://localhost:3000/uploads, so we put a placeholder in page.tsx to ensure we can confirm things are working.

Listing 6.3 Placeholder upload page

```
import Typography from "@mui/material/Typography";
import * as React from "react";

export default function Uploads() {
    return (
        <Typography variant="h1">Hello, Uploads!
➥</Typography>
    )
}
```

We export a default function named Uploads.

Typography is a component from MUI that allows us to create text; here, we are using a variant of h1, which will give us a very long text, just like the HTML h1 tag would.

With that in place, you should be able to navigate to the upload endpoint and see "Hello, Uploads!"

6.2.1 Testing our upload page

Even though we have not created a substantial page, let's take a moment to see how we may test our pages in React. We will use Jest, a popular option for testing React apps. First, we need to install the required dependencies.

Listing 6.4 Jest dependencies

```
npm install -D jest jest-environment-jsdom
➥@testing-library/react @testing-library/jest-dom ts-node
```

Additional libraries to help support testing efforts

Next, we generate a basic configuration with `npm init jest@latest`. Then, we create a directory structure where we keep our tests.

Listing 6.5 Test directory structure

```
__tests__/
├─ pages/
|   ├─ uploads.test.tsx
```

As shown in listing 6.5, the standard procedure is to create a high-level `__tests__` directory as the subdirectories do not matter then (you only want them to make sense). The uploads.test.tsx will store our actual test code. For now, we only want to ensure that our "Hello, Uploads!" text is displayed when the upload component is rendered. To accomplish that, the following listing shows the sample test.

Listing 6.6 Simple Jest test

```
import { render, screen } from '@testing-library/react'
import Uploads from "@/app/uploads/page";
import '@testing-library/jest-dom/jest-globals';
import '@testing-library/jest-dom';
import {describe, it} from "@jest/globals";
import expect from "expect";

describe('Upload Page', () => {
    it('renders a heading', () => {
        render(<Uploads />)

        const heading = screen.getByRole('heading', { level: 1 })

        expect(heading).toBeDefined();
        expect(heading).toHaveTextContent(/^Hello, Uploads!$/);
    })
})
```

The needed import statements for our test

The it keyword starts our test, along with the header of 'renders a heading'.

The describe keyword allows us to group tests, which will be displayed under the title "Upload Page."

Uses a regular expression to ensure the text matches our expected text

Ensures the variable is defined

Retrieves the h1 tag using the ARIA role. The level parameter corresponds to the size of the header: h1, h2, and h3 would use levels 1, 2, and 3, respectively.

Renders our component

The unit test defined in listing 6.6 should look somewhat familiar, given the test scripts we created in Python with `pytest`. Of course, the syntax may differ slightly, but the approach is the same. It would be interesting to revisit the `getByRole` method used to get our header tag and its contents. This method uses the role defined by the Accessible Rich Internet Applications (ARIA) standard. By creating our unit tests using `getByRole`, we take an accessibility-first approach to our web page design. Assistive technologies will be interacting with our web UI through ARIA tags.

If you are developing within an IDE such as JetBrains, WebStorm, or VS Code, you may be able to run these tests directly from within the IDE. Or, you could execute the command `npm run test` from the command line. Executing from the command line helps illustrate the importance of the `describe` and `it` keywords. As shown in listing 6.6, the `Upload Page` heading is pulled from the `describe` keyword. Each test defined by the `it` keywords is also represented, As shown in figure 6.1, these keywords keep our tests organized and the output readable.

As we can see, unit testing is not just important for our backend Python code, but for the UI as well.

If you have directly browsed the uploads page, you can see that we have lost our navigation bars for the page. The next section shows how to add them back.

```
PASS __tests__/pages/uploads.test.tsx
  Upload Page
    √ renders a heading (52 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.279 s
Ran all test suites.
```

Figure 6.1 Output from running our Jest test

6.3 Refactoring our navigation

As with any code we did not write ourselves, we need to poke around to get a feel for the layout and structure. Since we will be customizing the dashboard for our own needs, we need to understand how the dashboard navigates between components. This may take some detective work, but it's worth it. After some digging, we find that the navigation for our pages is controlled by two components that we can see in our Dashboard.tsx—the `AppBar` and the `Drawer`. We can start by refactoring them into their own components. After we perform the refactoring, we can decide how to include them on subsequent pages.

The first order of the business is to extract the `AppBar` tag from Dashboard.tsx and move it to its own component in `src/app/components/navigation/TopMenuBar.tsx`. The directory structure and naming conventions can be anything within reason. The component then contains the existing `AppBar` logic with a few updates to handle, regardless of whether the drawer is open or closed.

Listing 6.7 Navigation menu bar

```
...
interface TopMenuBarProps {
    toggleDrawer(): void;
    drawerOpen: boolean;
}
```

Defines an interface for the TopMenuBar that takes a function to execute when the button is clicked and a boolean to determine whether the drawer is open

```
…
export default function TopMenuBar({toggleDrawer, drawerOpen}:
➥ TopMenuBarProps) {

    const [open, setOpen] = useState(drawerOpen);

    useEffect(() => {
        setOpen(drawerOpen);
    }, [drawerOpen]);

    return (
    <AppBar position="absolute" open={open}>
…
<IconButton edge="start" color="inherit" aria-label="open drawer"
➥ onClick={toggleDrawer}>
…
    </AppBar>
    );
}
```

Maintains the state in the component

When the parameter drawerOpen changes, updates the state accordingly

The AppBar's open parameter is our internal state.

When the icon is clicked, the toggleDrawer function that was passed in is executed.

We also want to do something similar for the sidebar navigation, so we refactor the `Drawer` tag from Dashboard.tsx, moving it to its own component.

Listing 6.8 Sidebar navigation

```
…
interface SideBarNavProps {
    toggleDrawer(): void;
    drawerOpen?: boolean;
}
…
export default function SideBarNav({toggleDrawer, drawerOpen}:
➥ SideBarNavProps) {

    const [open, setOpen] = useState(drawerOpen);

    useEffect(() => {
        setOpen(drawerOpen);
    }, [drawerOpen]);

    return (
    <Drawer variant="permanent" open={open}>
…
            <IconButton onClick={toggleDrawer}>
…
    </Drawer>
    );
}
```

Defines an interface for the SideBarNavProps that takes a function to execute when the button is clicked and a boolean to determine whether the drawer is open

Unpacks the SideBarNavProps as part of the function call

Maintains the state in the component

When the parameter drawerOpen changes, updates the state accordingly

Our open state is used to determine the Drawer state.

When the icon is clicked, the toggleDrawer function that was passed in is executed.

We can then use these components in the Dashboard.tsx as shown in the next listing.

Listing 6.9 Updated Dashboard.tsx

```
…
const [open, setOpen] = useState(true);
const toggleDrawer = () => {
    setOpen(!open);
};
…
      <TopMenuBar toggleDrawer={toggleDrawer}
➥ drawerOpen={open} />
      <SideBarNav toggleDrawer={toggleDrawer}
➥ drawerOpen={open} />
…
```

Stores the state of the drawer

Creates an arrow function named toggleDrawer we call to set the open state

Uses our refactored components which accept a function to show/hide the drawer and the current state of the drawer

Working through this refactoring exercise introduced us to Material UI and Next.js. In addition, we are now obviously able to include these components in our upload page, which was our original goal. While we have refactored the code, there is some noticeable duplication handling regarding whether the drawer is open. With the way we have it at the moment, we will need to duplicate the `open` state and `toggleDrawer` function in each component we choose to use them in, which is less than ideal since they will always need to be used together.

Navigation component challenge

While there may be the occasional need to duplicate code, we probably do not need or want to do it in this case. There are a few ways we may want to handle this (maybe we do not want to break out the `AppBar` and `Drawer` elements). If we refactor them into a single component, we would only have to import that single component. Or perhaps, since we have them broken apart already, can we just wrap them into a single component? The goal of this exercise is to eliminate the need to duplicate the `open` state and `toggleDrawer` function in every component.

6.4 Converting recent order to recent ACH uploads

Now that we dipped our toes into some Next.js, let's take a look at reworking the `Recent Order` component to list our recent ACH uploads instead. As always, we are looking to work in short development cycles. This means we first use the hardcoded data that is part of the template and update it to fit our needs. Then, we replace the hardcoded data with a mocked API call. The template has a function to create the rows of data, as shown in the next listing.

Listing 6.10 `createData` function

```
function createData(
  id: number,
  date: string,
  name: string,
```

The existing parameters we need to update

```
  shipTo: string,
  paymentMethod: string,
  amount: number,
) {
  return { id, date, name, shipTo, paymentMethod, amount };
}
```

The existing return statement we need to update

We repurpose this code to include fields that we want to see when viewing loaded files for the day. For now, we can imagine that we would like to see the date the file was loaded (possibly the time as well), the name of the file, and both the credit/debit amounts for the file. The required updates are shown in the next listing.

Listing 6.11 Updated `createData` function

```
function createData(
    id: number,
    date: string,
    name: string,
    creditAmount: number,
    debitAmount: number,
) {
    return { id, date, name, creditAmount, debitAmount };
}
```

Updates parameters for createData. We do not necessarily need to worry about the data types at this point. We just use the standard types of number and string. We could define the date as a Date type but that is unnecessary at this point.

Updated return statement

Next, we create our rows of data with the new amounts. Note that we have two amount fields, and whenever we have amounts, we want to include a variety of them to ensure the data will display as desired (determined by the stakeholders in the project). Depending on the API and datatype we use, there could be some look and feel problems with the UI when displaying amounts. Therefore, listing 6.12 shows data we are passing, and we want to be sure we hit some of the edge cases such as

- Trailing zero, which will likely be thrown out when using a numeric datatype
- Dollar amounts greater than $999 to see a separator (such as a comma) for the amount

Listing 6.12 Sample data

```
createData(
    1,
    '03/20/2023',
    'sample1.ach',
    1866.90,
    1234.10
)
```

Returns a date; this may come through from the API as a timestamp we'll need to handle.

Ensures our numeric values cover some of the use cases we outlined

Depending on the API, you may be receiving a formatted amount, in which case, you obviously do not need to worry about the mentioned problems. However, when we design APIs, we usually try to keep the formatting to a minimum and not enforce

formatting/locale specifics unless necessary. So, we favor returning `1000.0` for $1,000.00 because, depending on the location, we may use a space and/or decimal to separate numbers. Of course, use cases differ from one company to another and one project to another. In our situation, we are in the unique position of having control over both the UI and the backend, which means we could return our amount exactly how we want it displayed and just put it directly onto the page.

Although unlikely to be a problem with this application, localization (L10n) may be a concern to us since ACH deals with US-based financial institutions, which is why we favor the unformatted amounts being returned from the API. Perhaps we plan for our dashboard to be abstract enough to be easily adapted to SEPA or BECS, so we need to ensure flexibility in our application.

Next, we update the table to display the updated rows. We take the data stored in `rows` and use the `map` function to take each row and create our table rows.

Listing 6.13 Updated table

```
<Table size="small">
   <TableHead>
      <TableRow>
         <TableCell>Date</TableCell>
         <TableCell>Filename</TableCell>
         <TableCell align="right">Credit Total</TableCell>
         <TableCell align="right">Debit Total</TableCell>
      </TableRow>
   </TableHead>
   <TableBody>
      {rows.map((row) => (
         <TableRow key={row.id}>
            <TableCell>{row.date}</TableCell>
            <TableCell>{row.name}</TableCell>
            <TableCell align="right">
               {`${row.creditAmount}`}
            </TableCell>
            <TableCell align="right">
               {`${row.debitAmount}`}
            </TableCell>
         </TableRow>
      ))}
   </TableBody>
</Table>
```

The updated table headers

We use map to create a new array of <TableRow> tags for each element.

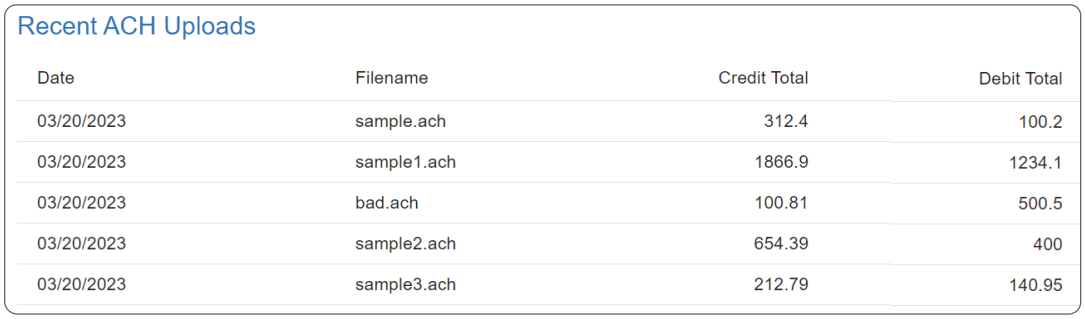
Each table row contains a key property.

Using our row data for date and name

We display the amount fields as-is to see how they look.

At this point, if we view our web page, we should notice that we are experiencing the problems we mentioned—truncating the trailing zero and missing a thousands separator, as shown in figure 6.2.

We could accomplish some basic formatting by using `toFixed` and a hardcoded dollar sign, as in listing 6.14.



Recent ACH Uploads

Date	Filename	Credit Total	Debit Total
03/20/2023	sample.ach	312.4	100.2
03/20/2023	sample1.ach	1866.9	1234.1
03/20/2023	bad.ach	100.81	500.5
03/20/2023	sample2.ach	654.39	400
03/20/2023	sample3.ach	212.79	140.95

Figure 6.2 Unformatted credit and debit totals

Listing 6.14 Basic formatting

```
<TableCell align="right">
   {`$${row.creditAmount.toFixed(2)}`}
</TableCell>
<TableCell align="right">
   {`$${row.debitAmount.toFixed(2)}`}
</TableCell>
```

Performs some initial formatting on the amounts, forcing the use of the dollar sign and ensuring we see two decimal places by using toFixed(2)

This code gets us basic formatting with only the thousands separator. Instead of having to remember to format currency everywhere, which can lead to inconsistencies and lock us into a locale, let's define a function we can call whenever we want to format a currency. Not only will this provide consistency in our presentation of currencies, but it should also allow us to find places where we are displaying currencies in our code (should we ever need to).

To accomplish this, we will be using `Intl.NumberFormat`, which not only can format currencies for different locations but also for other units of measurements such as mph/kh and liter/gallon. The following listing shows our simple exported function that wraps `Intl.NumberFormat`.

Listing 6.15 Wrapper for `Intl.NumberFormat`

The function takes in a number.

Different currencies can be passed in but default to USD.

Locale can be controlled as well, defaulting to en-US.

```
export function formatCurrency(value: number,
           currency: string = "USD",
           locale: string = 'en-US'): string {
    return new Intl.NumberFormat(locale, {
        style: 'currency',
        currency: currency
    }).format(value);
}
```

The actual work is done by Intl.NumberFormat that we style as currency.

We can also create some Jest tests to ensure we understand how this code functions, as well as confirm that we are seeing the expected formatting. The following listing shows example Jest tests we may want to use to confirm formatting is taking place.

Listing 6.16 Jest tests for our `formatCurrency` routine

```
import {describe, it} from "@jest/globals";
import expect from "expect";
import {formatCurrency} from "@/app/utils/CurrencyUtils";

describe('Formatting Currency', () => {
    it('Has trailing zeros', () => {
        const result = formatCurrency(1.0)

        expect(result).toBe('$1.00')
    })

    it('Default currency is USD', () => {
        const result = formatCurrency(1.1)

        expect(result).toBe('$1.10')
    })

    it('Has comma separator', () => {
        const result = formatCurrency(1000.0)

        expect(result).toBe('$1,000.00')
    })

    it('Has comma separators for large numbers', () => {
        const result = formatCurrency(1000000.0)

        expect(result).toBe('$1,000,000.00')
    })

    it('German formatting', () => {
        const result = formatCurrency(1000000.0, 'EUR', 'de-DE')

        expect(result).toBe('1.000.000,00\u00A0€')
    })
})
```

Imports the necessary functions

Uses describe to group the tests together

Ensures the trailing zeroes are not dropped

Ensures the defaults are taken

Checks that a comma is included

Checks that multiple commas are added

Tests out German formatting in euros. The \u00A0 is a Unicode notation for no-break space.

After confirming our tests run successfully, we can replace our table cells that called `toFixed(2)` with a call to our new `formatCurrency` routine.

Listing 6.17 Using the new `formatCurrency` function

```
<TableCell align="right">
   {{formatCurrency(row.creditAmount)}}
</TableCell>
```

Replaces the manual formatting with a call to our new function

```
<TableCell align="right">
   {{formatCurrency(row.debitAmount)}}
</TableCell>
```

Figure 6.3 shows our amounts formatted exactly the way we wanted it, with commas and two significant digits after the decimal.

Recent ACH Uploads

Date	Filename	Credit Total	Debit Total
03/20/2023	sample.ach	$312.40	$100.20
03/20/2023	sample1.ach	$1,866.90	$1,234.10
03/20/2023	bad.ach	$100.81	$500.50
03/20/2023	sample2.ach	$654.39	$400.00
03/20/2023	sample3.ach	$212.79	$140.95

Figure 6.3 Formatted credit and debit totals

It is often easier to use an existing component as a starting point and customize it to meet our needs. After our customizations, we have a working component capable of displaying our uploaded ACH files to the user.

6.5 Mocking our ACH uploads data

In the previous section, we customized our component to fit our particular use case of viewing recent ACH file uploads. However, we used hardcoded values, and while that helped us get a feel for the fields we need, it is not very useful for anything other than mocking up the UI. We have a database that we will want to retrieve this data from, but we are not quite there yet in terms of building APIs. As an intermediate step, let's add the API call but have it retrieve mocked data instead of actually hitting the endpoint. Mocking data not only lets us get started quickly, but also has a variety of other uses such as for testing edge cases, simulating bad data, and other scenarios. So, learning about mocking data will have long-lasting benefits beyond this example.

Why would we choose to do something like this? The reality is that projects can span multiple people, teams, departments, and even companies. It is entirely possible that we may be assigned to work on the UI, while the APIs and backend code are being developed in parallel. This type of parallel development is often going on when we are trying to bring our product to market as fast as possible, and we cannot wait for the APIs and backend to be finished before we start working on the frontend. Or perhaps, we need to call a third-party API, but contract negotiations are still ongoing, and we do not have access to the API.

In these cases, a tool such as WireMock can allow our code to make the same API call but return a predefined response. We could also use this tool to obtain error responses or edge case data that we may not have readily available, which can help us test our code.

6.5.1 Adding WireMock to Docker

We add WireMock to our project in the same manner—by creating a Dockerfile first and then referencing it in our docker-compose.yml, as shown in the following two listings. The needed Dockerfile does not have much in it and could be contained in the docker-compose.yml, but we prefer to keep the docker-compose.yml as short as possible.

Listing 6.18 Dockerfile for WireMock

```
FROM wiremock/wiremock:latest-alpine

ENTRYPOINT ["/docker-entrypoint.sh",
➥ "--global-response-templating", "--disable-gzip",
➥ "--verbose"]
```

Notice that in listing 6.19, we have introduced volumes to our WireMock service definition. These volumes exist on the local file system and are made available by mapping them to a directory in the container. Similar to the way ports are mapped, we specify the host directory first and then the directory it should map to in the container. WireMock allows us to specify stub mappings and file definition stubs (more on this later). Using volumes will allow us to make changes to the mappings without the need to build repeatedly. While a Dockerfile allows the `VOLUME` keyword where we could specify volumes, the recommended approach is to use the docker-compose.yml file because it allows for a separation of duties between building (Dockerfile) and runtime (docker-compose.yml).

How about using `COPY` commands in the Dockerfile to move our mappings to the container? That would be another option to use if we were interested in building an image containing mappings that we would then distribute to our team. However, in this case, we are only interested in working with these mappings locally, and we do not need to rebuild our container whenever we want to utilize an updated mapping.

Listing 6.19 Updates to docker-compose.yml

```
wiremock:
  build:
    context: ./wiremock
    dockerfile: Dockerfile
  volumes:
    - ./wiremock/__files:/home/wiremock/__files
    - ./wiremock/mappings:/home/wiremock/mappings
  ports:
    - 8081:8080
```

Map locations on our filesystem to our container

With that complete, we can start our Docker containers again and navigate over to http://localhost:8081/__admin/, where we should receive a response similar to the following listing.

Listing 6.20 WireMock admin response

```
{
  "mappings" : [ ],
  "meta" : {
    "total" : 0
  }
}
```

This code confirms that we indeed have WireMock up and running. Perhaps not unsurprisingly, we do not have any mappings defined yet. So, let's tackle that in the next section.

6.5.2 Setting up mappings in WireMock

WireMock is a powerful tool, and we only scratch the surface in this section. However, it should be enough to get us started, and we can always investigate its additional features as we need more functionality (remember our YAGNI principle from chapter 3). We are primarily interested in stubbing out mock responses so that when our UI makes an API call, we are returning some meaningful data. We start by creating a simple stub in the `./wiremock/mappings` directory named hello_world.json. The name is not extremely important at this point. The next listing shows the contents of the file.

Listing 6.21 Our first stub

```
{
    "request": {
        "method": "GET",
        "url": "/api/v1/hello/world"
    },
    "response": {
        "status": 200,
        "body": "Hello World!",
        "headers": {
            "Content-Type": "text/plain"
        }
    }
}
```

If we try to access http://localhost:8081/api/v1/hello/world, we receive a response such as `No response could be served as there are no stub mappings in this WireMock instance`. What gives? We need to make an empty `POST` call to http://localhost:8081/__admin/mappings/reset for WireMock to be able to pick up our changes. There are some Postman samples associated with this text that can get you started. After that call,

you should be able to access the endpoint again and see "Hello, World!" Furthermore, if you request the mappings from http://localhost:8081/__admin/, you will see something similar to the following listing.

Listing 6.22 WireMock admin response with our new stub

```
{
    "mappings": [
        {
            "id": "ac60ce7e-63f4-417e-894b-3e25ad496c42",
            "request": {
                "url": "/api/v1/hello/world",
                "method": "GET"
            },
            "response": {
                "status": 200,
                "body": "Hello World!",
                "headers": {
                    "Content-Type": "text/plain"
                }
            },
            "uuid": "ac60ce7e-63f4-417e-894b-3e25ad496c42"
        }
    ],
    "meta": {
        "total": 1
    }
}
```

That should be painless. Now, let's look at how we might want to mock our response to obtain the information for uploaded ACH files. We should already have a good idea of the fields we want to return because we have already used hardcoded data when building the UI. So, we will use that data and create ach_files_response.json in the `__files` directory this time. By defining responses in the `__files` directory, it will save us from complicating our mappings and allow other APIs to reuse responses as necessary. A response sample is shown in the next listing, and we can use as much or as little data as we want.

Listing 6.23 Sample ACH response body

```
[
    {
        "id": "45bc2000-0a14-4c0a-ae7f-938b3fdab575",
        "date": "03/01/2024",
        "filename": "sample.ach",
        "creditTotal": 312.40,
        "debitTotal": 100.20
    },
…
]
```

Sample JSON that contains our test data

Now that we have the contents of the response defined, we need to define a mapping for it. The mapping is similar to what we have already seen, except that we now specify the `bodyFileName` in the response with the filename of ach_files_response.json we defined before.

Listing 6.24 ACH response mapping

```
{
    "request": {
        "method": "GET",
        "url": "/api/v1/files"
    },
    "response": {
        "status": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "bodyFileName": "ach_files_response.json"
    }
}
```

The request will only execute on a GET.

Further refines the mapping with the URL

The response will be a status code 200.

Defines any headers that are needed

Uses bodyFileName and specifies a name in the __files directory to return

With that defined, we just need to refresh our mappings for the changes to take effect. We can use multiple methods, and they are listed in the order of their impact on the overall environment (from least to most impactful):

1. The Admin API can be restarted to reload the mappings.
2. Restart the container.
3. Restart our Docker compose instance completely.

Once the mappings have been refreshed, navigating to http://localhost:8081/api/v1/files will show the response. Figure 6.4 illustrates the response using Postman, which provides some highlighting/formatting of the output and allows us to check returned headers easily.

WireMock is a powerful tool, and we only scratched its surface. For now, it is enough to get us going with building the UI. Next, we look at how to make requests to the backend.

6.5.3 Accessing endpoints with axios

The first thing we need to do, if not done already, is ensure axios is installed with `npm install axios`. After that, we can import it into our components as needed and get to work!

Now we can open the RecentAchUploads.tsx, remove the `createData` function, and clear out our test data since we will be retrieving our data from the WireMock instance. Instead of hardcoded data, we will define our rows to be an array of `AchUploads` (listing 6.25).

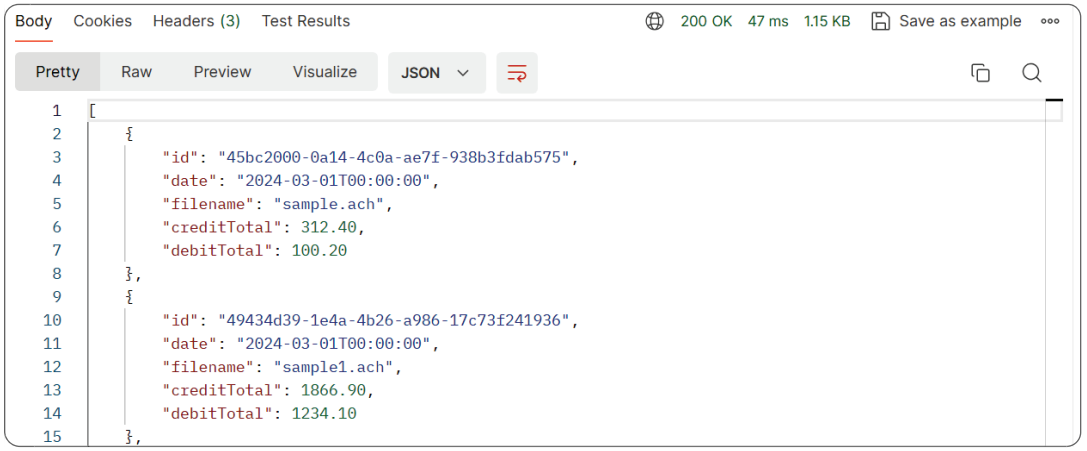


Figure 6.4 Response from our `/api/v1/files` endpoint as viewed in Postman

Listing 6.25 Preparing our component to receive data

```
…
import {useState} from "react";
…
interface AchUpload {
    id: number;
    date: string;
    name: string;
    creditAmount: number;
    debitAmount: number;
}

export default function RecentAchUploads() {

    const [rows, setRows] = useState<AchUpload[]>([]);
…
```

Defines a structure for our data so we can type it appropriately

Our rows data will be stored in an array of type ACHUpload.

Now, we can make our request to populate data.

Listing 6.26 Making an API call with axios

```
useEffect(() => {
   axios.get('http://localhost:8081/api/v1/files')
   .then(response => {
      console.log(`Response data
➥ ${JSON.stringify(response.data)}`);
         setRows(response.data);
      })
```

useEffect is a perfect React function for making API calls.

Uses axios to get our files listing

Handles the returned Promise by displaying the data and setting rows to the returned data

```
    .catch(error => {
          console.log(error);
    });
}, []);
```

Logs any problems to the console

The empty array [] tells React to execute this only once.

Usually, we recommend that these API calls are offloaded and called from a single location instead of being embedded in the components, but for now, it is okay to have them inside the component. For instance, we often create `api` folder or equivalent where these functions can be imported from. This allows the API calls to be easily accessed from multiple locations.

However, our more immediate concern is that we have a hardcoded URL pointed to the localhost (http://localhost:8081), which will certainly change when we deploy the Dashboard to Docker. So for the time being, let's take advantage of Next.js to store our variables, and then, we'll be able to configure them for Docker easily. Thus, we create an .env.local file.

Listing 6.27 Our .env.local file

```
NEXT_PUBLIC_API_URL=http://localhost:8081/api/v1
```

We can now update our code to use the environment variable in our URL.

Listing 6.28 Updated API call

Returns the environment variable or sets it to the empty string when undefined

```
useEffect(() => {
    const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
    axios.get(`${apiUrl}/files`)
```

Updates the axios call to use the new apiUrl

With this code, we can test our API and still see the sample files we are returning from WireMock. We are now able to change the URL our APIs hit when we are building our Docker containers.

6.6 Expanding the UI with an ACH batch view

Now that you understand these pieces, let's take a look at combining Next.js, Recharts, Axios, and WireMock to create a view of ACH batches. On the dashboard, we would like to be able to click one of the file names presented to view the batches for that file. The view will display a scatterplot and provide the ability to display more details about the batches for each company in the ACH file, as shown in figure 6.5. Providing information on the characteristics of the batches (in this case their size based on their amount and number of entries) can help us start putting controls in place for tasks such as fraud detection and risk management in the future as we expand the

application. The scatterplot allows us to easily spot batches with larger dollar amounts or with higher item counts. Eventually, if we were to mix in historical data, we could then analyze patterns for these companies and highlight unusual activity patterns.

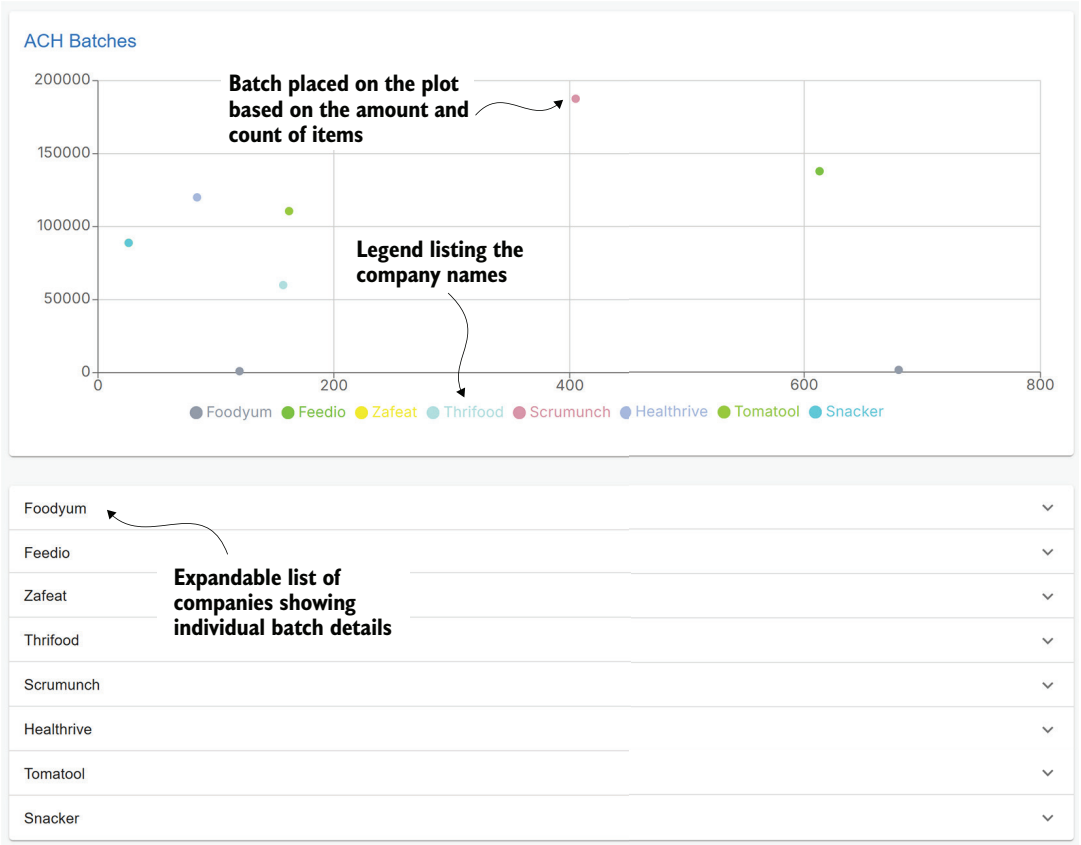


Figure 6.5 An ACH batch view

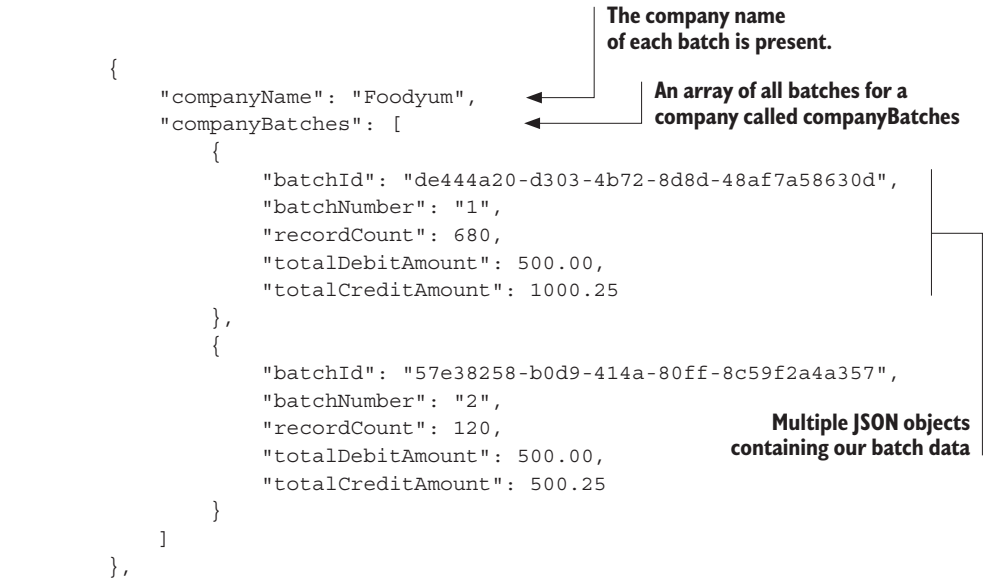
6.6.1 *Defining our WireMock data for the ACH batch view*

Our JSON data will essentially be an array of arrays. We also know that there can be multiple batches in an ACH file. Each batch has an associated company, and the company may have multiple batches in a file. We want to have the batches returned and grouped by the company. So, something like the following listing should work.

Listing 6.29 Sample ACH batch data

```
{
    "batches" : [
```

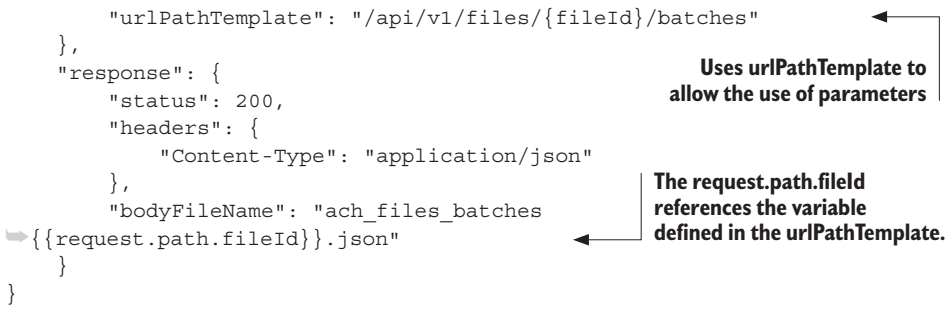
We return an array of all batches in a file grouped by company.




```
        {
            "companyName": "Foodyum",
            "companyBatches": [
                {
                    "batchId": "de444a20-d303-4b72-8d8d-48af7a58630d",
                    "batchNumber": "1",
                    "recordCount": 680,
                    "totalDebitAmount": 500.00,
                    "totalCreditAmount": 1000.25
                },
                {
                    "batchId": "57e38258-b0d9-414a-80ff-8c59f2a4a357",
                    "batchNumber": "2",
                    "recordCount": 120,
                    "totalDebitAmount": 500.00,
                    "totalCreditAmount": 500.25
                }
            ]
        },
...
```

We define some batches we are going to associate with the stubbed data we used for the file. Specifically, we want to return this data when someone queries an API endpoint such as `/api/v1/files/{fileId}/batches`. As we saw in WireMock, we could define hardcoded paths for all the `fileId`s we were interested in. However, that would result in a lot of mappings. Instead, we can dynamically match these fields using templating and return specific responses for each. We define ach_files_batches.json in the following listing. Note that we use `urlPathTemplate` and specify `{fileId}`. Then we also use `{{request.path.fileId}}` in the `bodyFileName` of the response, which allows us to extract the dynamic UUID we expect to receive and use it in a response we would then define in the `__files` directory.

Listing 6.30 WireMock templating example




```
{
    "request": {
        "method": "GET",
        "urlPathTemplate": "/api/v1/files/{fileId}/batches"
    },
    "response": {
        "status": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "bodyFileName": "ach_files_batches
➥{{request.path.fileId}}.json"
    }
}
```

With all said, we would have the ability to define specific responses to API calls. Figure 6.6 illustrates this concept better. Note that we are not actually saving anything into WireMock. We are using canned responses we have prepopulated in the WireMock `__files` directory.

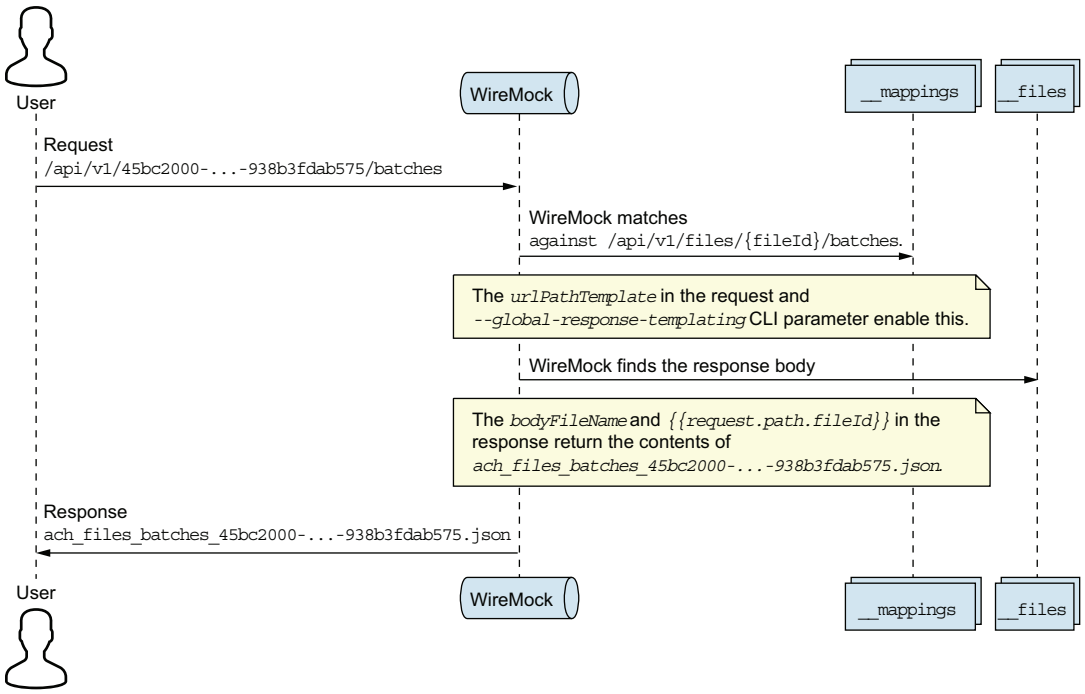


Figure 6.6 WireMock sequence

Now that we are able to return data, we can define a component that can display it.

6.6.2 Updating the dashboard to take us to the batches screen

We update the RecentAchUploads.tsx so that the filename becomes a link.

Listing 6.31 Updated `RecentAchUploads`

```
<TableCell>
   <Link onClick={() =>
      route.push(`/fileDetails/${row.id}`)}>
         {row.filename}
   </Link>
</TableCell>
```

Updates the text to be a link that will use the file ID

The `fileDetails` screen takes the UUID of the file, which will help drive the resulting queries. Next.js uses file-based routing that lets us define the URL using a simple folder

structure. In this case, including the UUID as part of the URL is denoted by a directory in square brackets. The following listing is the folder structure we need to create.

Listing 6.32 Folder structure for the file details page

```
src/
├─ app/
|  ├─ fileDetails/
|  |  ├─ [id]/
|  |  |  ├─ page.tsx
```

The square brackets denote a parameter on the URL we can access.

The `[id]` is a dynamic parameter in our URL. For example, with this structure, we should be able to navigate to URLs such as `/fileDetails/0854b018-c5ce-4e00-8a4f-d268a1a85aa2` or `/fileDetails/d11fb49c-4a07-4b96-88cc-db1fc7277bfa`, assuming those were valid UUIDs for our ACH files.

6.6.3 Displaying batch details

The page.tsx will contain the logic to retrieve the batch information and display it.

Listing 6.33 Batch details

```
…
export interface AchBatchInfo {
    companyName: string;
    companyBatches: AchBatchDetails[];
}
…
export interface AchBatchDetails {
    batchId: string;
    companyName: string;
    recordCount: number;
    totalDebitAmount: number;
    totalCreditAmount: number;
}
…
interface AchBatchInfoResponse {
    batches: AchBatchInfo[];
}
…
export default function FileDetails({params}: any) {

    const id: string | string[] = params.id;
    const [batches, setBatches] =
➥ useState<AchBatchInfo[]>([]);

    useEffect(() => {
        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        axios.get<AchBatchInfoResponse>(`${apiUrl}/files/${id}/batches`)
            .then(response => {
                console.log(`Response data
➥ ${JSON.stringify(response.data.batches)}`);
```

Defines our JSON structure so that we can strongly type the response

Allows the URL parameters to be accessed

References the URL parameter by the same name we defined in square brackets

Stores the batches with useState like we did in other components

Uses useEffect to gather the data from the API

```
                setBatches(response.data.batches);
            })
            .catch(error => {
                console.log(error);
            });
    }, []);
…
                    {batches.map((item, index) => (
                            <Accordion key={item.companyName}>
                                <AccordionSummary
➥ expandIcon={<ExpandMoreIcon />}>
                                    <Typography>{item.companyName}</
    Typography>
                                </AccordionSummary>
                                <AccordionDetails>
{item.companyBatches.map((companyBatch, index) => (
                                    <>
                                        <Typography>
                                    Batch ID:
➥ {companyBatch.batchId}<br />
                                    Record Count:
➥ {companyBatch.recordCount}<br />
                                    Total Debit Amount:
➥ {formatCurrency(companyBatch.totalDebitAmount)}<br />
                                    Total Credit Amount:
➥ {formatCurrency(companyBatch.totalCreditAmount)}
                                        </Typography>
                                        <Link>View Batch</Link>
                                        <Divider />
                                    </>
                                )
                                )}
                                </AccordionDetails>
                            </Accordion>
                        ))}
…
```

Maps the returned batches to display the UI elements

This component presents the accordion view showed earlier in the chapter as part of figure 6.5 for the companies in the ACH file. An important takeaway from this component is seeing how we can dynamically create layout elements on our page. In other components, we will use DataGrid views that provide a table structure for viewing dynamic data. However, it is helpful to know that we can create layouts with dynamic data manually, as well when the need arises.

6.7 Creating a ScatterPlot component

Finally, we want to add a scatterplot to the page that shows the items in the batch versus the total amount of debits and credits. The following listing shows the directory structure we will be creating. You may create another subdirectory to store specific scatterplot-related files. Since we have not reached the point where this directory contains a lot of files, we will leave this as is for the time being.

Listing 6.34 Directory structure for our charts

```
src/
├─ app/
|  ├─ components/
|  |  ├─ charts/
|  |  |  ├─ AchBatchesScatterPlot.tsx
|  |  |  ├─ AchBatchesScatterTooltip.tsx
|  |  |  ├─ CurrencyTick.tsx
```

6.7.1 Scatterplot component

A scatterplot is useful for showing the possible correlation between variables. In our case, we may not necessarily be interested in the analytic aspect. However, the scatterplot also provides a way to identify batches with a high number of items or with a large dollar amount, which may be interesting to our users. Certainly, there may also be an opportunity for us to display trends in the batches for a particular company, which may also help our users perform analysis.

In the meantime, this component provides an opportunity to explore using colors, custom tooltips, and custom ticks on the axis. These features are discussed in subsequent sections. The initial code for the scatterplot is shown in listing 6.35, with the actual component rendering shown in listing 6.36.

The component defines an interface called `AchBatchesScatterPlotProps` so that it can set the types properly. In addition, it is important that this component accepts the batches as a read-only parameter rather than making any API calls inside the component. We could have just as easily made the API calls inside the component. However, if we had multiple components that wanted to make use of this data, we would end up with repeated API calls to gather the same data. We also marked the data as read-only to help enforce the concept of the component only utilizing the data and not introducing unintended consequences by being used. However, note that `Readonly` is a compile-time feature for TypeScript, and the resulting JavaScript will not have it. Finally, since we are interested in the total dollar amount of the batch, we add the credit and debit amounts together and store them into `totalCreditAndDebits`.

Listing 6.35 Scatterplot component

```
...
interface AchBatchesScatterPlotProps {
    batches: AchBatchInfo[];
}

export default function AchBatchesScatterPlot({batches}:
➥ Readonly<AchBatchesScatterPlotProps>) {

    const plotData = batches.map(batch => {
        const updatedCompanyBatches =
➥ batch.companyBatches.map(companyBatch => {
```

Defines an interface for the component properties, so we can strongly type them

Accepts the properties as Readonly so we do not modify them

```
            return {
                ...companyBatch,
                totalCreditAndDebits:
➥ companyBatch.totalCreditAmount + companyBatch.totalDebitAmount
            };
        });
        return {
            ...batch,
            companyBatches: updatedCompanyBatches
        };
    });

    return (
    # Refer to AchBatchesScatterPlot.tsx source code for this logic
    );
}
```

Spreads the contents of the companyBatch object

Adds a new property to the object named totalCreditAndDebits

Spreads the content of the batch object

Replaces the existing companyBatches with the new batches containing the totalCreditAndDebits

In the following listing, we have the component that renders the actual scatterplot. Here we take advantage of the flexibility of recharts by defining custom tooltip content by passing a component called `AchBatchesScatterTooltip` and by defining a custom tick mark component for the `YAxis` called `CurrencyTick`. Furthermore, we assign a different color to each company when placing the point on the plot (more about these custom components in the following sections).

Lisitng 6.36 Scatterplot render

```
<>
   <Title>ACH Batches</Title>
   <ResponsiveContainer>
      <ScatterChart
         width={400}
         height={400}
         margin={{
            top: 20,
            right: 20,
            bottom: 20,
            left: 20,
         }}>
      <Legend />
      <CartesianGrid />
      <XAxis type="number" dataKey="recordCount" name="Count" unit="" />
      <YAxis type="number"
             tick={<CurrencyTick/>}
             dataKey="totalCreditAndDebits" name="Total Amount" unit="" />
      <Tooltip cursor={{ strokeDasharray: '3 3' }}
         content={<AchBatchesScatterTooltip />} />
         {plotData.map((entry, index) => (
            <Scatter key={`${entry.companyName}`}
                     name={`${entry.companyName}`}
                     data={entry.companyBatches}
                     fill={stringToColor(entry.companyName)}>
```

Custom component for the ticks on the Y-axis

Custom tooltip when hovering over a data point

Custom color depending on the company

```
                <Cell
                   key={`cell-${entry.companyName}-${index}`}
                            fill={stringToColor(entry.companyName)} />
                      </Scatter>
                   ))}
      </ScatterChart>
   </ResponsiveContainer>
</>
```

6.7.2 Scatterplot custom tooltip

After defining our custom tooltip component in the next listing, we can customize the displayed tooltip. We kept it simple by using the components available in Material UI. We can also reuse the `formatCurrency` method that we had defined to ensure the correct display of data in the tooltip.

Listing 6.37 Custom tooltip for scatterplot

```
export function AchBatchesScatterTooltip
➥ ({ active, payload, label }: TooltipProps<string, string>)  {
    if (active && payload && payload.length) {
        const value = payload[1].value ?? '0.0';
        return (
            <Tooltip title={<Typography>{`Value: ${label}`}
➥</Typography>} arrow>
                <Paper sx={{ p: { xs: 1, sm: 2, md: 3 } }}>
                    <Typography variant="h6">Total Debits and
➥ Credits</Typography>
                        {formatCurrency(Number.parseFloat(value))}
                </Paper>
            </Tooltip>
        );
    }

    return null;
}
```

Reuses our formatCurrency routine to ensure consistency in displaying amounts

6.7.3 Scatterplot custom ticks

Another way to customize charts in recharts is by defining a custom tick. Here we want our ticks to be formatted as currency instead of leaving them unformatted and adding a label to say "in dollars" or something similar. We define an interface that contains the passed data and then again use our `formatCurrency` utility (listing 6.38).

This component uses both `Readonly` (to ensure that the component does not modify any parameters) and `Partial`. The `Partial` type is used because `recharts` is in control of the parameters passed. Furthermore, rather than marking each parameter as optional by using `?`, we can mark all parameters in `CurrencyTickProps` as optional by using `Partial<CurrencyTickProps>`.

Listing 6.38 Custom currency tick

```
interface CurrencyTickProps {
    x: number;
    y: number;
    payload: {
        value: string;
    };
}

export function CurrencyTick({ x, y, payload }:
➥ Readonly<Partial<CurrencyTickProps>>): React.JSX.Element {
    const value = payload?.value ?? '0.0';
    return (
        <g transform={`translate(${x},${y})`}>
            <text x={0} y={0} dy={16} textAnchor="end"
➥ fill="#666" transform="rotate(-35)">
                {formatCurrency(Number.parseFloat(value))}
            </text>
        </g>
    );
```

Defines our properties so that they are strongly typed

Properties are marked as Readonly and Partial.

If the payload does not exist, substitute 0.0.

Uses our handy formatCurrency function

Obviously, the ability to customize our charts further using custom ticks is a powerful tool. Even if we chose to use the standard ticks, it is helpful to keep this type of customization in our back pocket for when the business makes a more specific request. In the next section, we will look at customizing colors for our charts.

6.7.4 Using colors for the chart

Colors are a very important part of any visualization. In our case, we do not limit the number of points in the scatterplot, so we do not necessarily know the number of companies that will have to be identified. We could always limit the visualization by saying it is a scatterplot of the "Top 10 companies" or the "Top 10 batches" so that we do not have to enumerate additional colors to handle an undetermined number of companies.

The next listing shows how we may enumerate the colors and then the code will loop through these colors and reuse them as necessary as we are doing `COLORS[index % COLORS.length]`. This approach will work but also has the problem of repeating colors when there are more companies than colors defined, and that will obviously make the visualization difficult to understand from a user's perspective.

Listing 6.39 Enumerating colors

```
const COLORS = ['#0088FE', '#00C49F', '#FFBB28', '#FF8042', 'red', 'blue'];
…
   <Scatter key={`${entry.companyName}`}
            name={`${entry.companyName}`}
            data={entry.companyBatches}
            fill={COLORS[index % COLORS.length]}>
```

```
      <Cell key={`cell-${entry.companyName}-${index}`}
            fill={COLORS[index % COLORS.length]} />
    </Scatter>
```

Another option we went with was computing the colors based on the company name, which provided us with a better opportunity to handle many companies in our visualization (although there will be a threshold where it is no longer valuable). However, as it stands, this approach also does not allow us to use names that may hash to the same value, nor does it necessarily give us the best colors in a visualization. Later, we may decide that showing the top 10 companies is the best way to handle the visualization.

Listing 6.40 shows the code used to compute a hex color code from the company name (or any string). In this case, the implementation details on how we create a color code from a string are not so important. Rather, we want to understand different ways we may use when approaching a problem and try to understand the potential tradeoffs in each approach. In this case, we have lost control over the exact colors being used, which also means the colors may be indistinguishable or conflict with the rest of the visualization. However, this approach allows us to show many companies, also providing consistency in color for the company. So, the color could be consistent in other visualizations or just change every day.

Listing 6.40 Using the company name to determine the color

```
export function stringToColor(str: string): string {
    let hash = 0;
    for (let i = 0; i < str.length; i++) {
        hash = str.charCodeAt(i) + ((hash << 5) - hash);
    }

    let color = '#';
    for (let i = 0; i < 3; i++) {
        const value = (hash >> (i * 8)) & 0xFF;
        color += ('00' + value.toString(16)).slice(-2);
    }
    return color;
}
```

How we display data is not only a matter of preference—our choice of layout, formatting, and colors can influence data interpretation. For instance,

- Using a red/green color scheme makes it harder for individuals with color blindness to distinguish the colors in different categories.
- 3D bar charts can be visually appealing but can lead to inaccurate interpretations of the data because the perspective can make bars appear larger/smaller than they are.
- In cherry-picked timeframes, data that does not fit a specific narrative is hidden or minimized.

For additional information on this topic, see *Everyday Data Visualization* (2024, Manning) by Desireé Abbott, especially if you want to take your visualizations to the next level.

6.8 Deploying the dashboard in Docker

As a final step, we want to deploy our code in Docker to tie all the pieces of our system together. The next chapter will focus on implementing the needed APIs, making sure components can communicate, and general housekeeping. For now, we only want to ensure the container is built correctly and can communicate with our WireMock instance.

As you may recall, our API calls use the `NEXT_PUBLIC_API_URL`, which allows us to configure the base URL needed to communicate with the backend. Listing 6.41 shows the service implementation for our docker-compose.yml. We now introduce the use of the `args` parameter to pass in arguments to our build process. We have set the URL to the same endpoint that we had in our `.env.local` project setup. This is because the code will be running on the client side in the browser, which means we will need to use `localhost:8081` and not the internal reference to `wiremock:8080`, as you may initially imagine.

Listing 6.41 Adding the dashboard to Docker

```
…
  dashboard:
    build:
      context: ./dashboard
      dockerfile: Dockerfile
      args:
        - NEXT_PUBLIC_API_URL=http://localhost:8081/api/v1
    ports:
      - 4000:3000
```

Defines variables to pass to the build

In the following listing, we have the Dockerfile referenced in listing 6.41. This code takes care of setting our environment variable, and building and running the application. In this Dockerfile, we utilize the `NEXT_PUBLIC_API_URL` by showing how we may set a default value, as well as how to set it in the container.

Listing 6.42 The Dockerfile for the dashboard

```
FROM node:alpine

ARG NEXT_PUBLIC_API_URL=
➥'http://override-me-in-docker-compose'
ENV NEXT_PUBLIC_API_URL $NEXT_PUBLIC_API_URL

WORKDIR /usr/src/app

COPY app/package*.json ./
```

Defines an argument that can be used or defaulted

Sets an environment variable that will contain our API URL

```
RUN npm install

COPY app/. .

RUN npm run build

CMD [ "npm", "start" ]
```

At this point, we should be able to build the containers and start them with `docker -compose down; docker-compose up --build` like before. Then, after navigating to http://localhost:4000, we should see our dashboard come up, displaying our WireMock data.

Summary

- The dashboard serves as the user interface for file uploads and data views, powered by APIs, and it can be developed using various technologies, with a current preference for Next.js.
- Next.js supports building interactive web applications with React by using HTML, CSS, and DOM principles and offering file-based routing, CSR, SSR, and multiple styling methods.
- Starting with Next.js involves setting up `Node.js`, creating a project using npx, and running a default application to verify setup correctness.
- The dashboard can be initialized using templates such as Material UI, and dependencies should be installed to support components and API calls.
- Incorporating TypeScript enhances maintainability by adding explicit types and catching errors during development.
- Jest is used for unit testing the UI, ensuring page functionality and accessibility through ARIA roles.
- Refactoring of navigation components warrants better code organization and inclusion in the upload page.
- Hardcoded data is initially used for UI development and then replaced with mocked API calls using tools such as WireMock for parallel development.
- Mocking APIs with WireMock involves setting up Docker, configuring stubs, and refreshing for changes to take effect, facilitating UI testing without complete backend availability.
- Deployment of the dashboard in Docker involves configuring environment variables and ensuring communication with mocked APIs for dynamic data display.

7 Our minimum viable product

This chapter covers

- The purpose of minimum viable product (MVP)
- Putting everything together into an MVP
- Security concerns
- Testing with Playwright

Over the previous sprints, we worked on spike stories examining our project's various components. You were introduced to generative AI and different tools that can expedite software development. We also reviewed specific technologies, such as Python/FastAPI, Postgres, Docker, and Next.js. Now, the line of business would like to see the fruit of our labor. This chapter focuses on putting everything together into a minimum viable product (MVP), which allows a user to

- Upload an ACH file
- Parse an ACH file and store it in the database
- Visualize the results in our dashboard

The functionality will not expand significantly from what was developed in the previous sprints. Instead, we want to ensure those individual components are now integrated seamlessly. Along the way, we will explore what happens in the system demo and validate our MVP using Playwright. Having the MVP will allow us to release something to our customers and gather feedback, which we'll use to enhance our project in the coming chapters.

7.1 Which minimum are we talking about?

During your career, you have probably heard about at least a few different minimums, such as

- *Minimum lovable product (MLP)*—Similar to an MVP but focuses on delivering something that customers love from the start
- *Minimum marketable product (MMP)*—Focuses on delivering the smallest amount of functionality that can be sold to our customers
- *Minimum marketable feature (MMF)*—Focuses on delivering software features that are of value to our customers

The MVP concept was first introduced by Frank Robinson and popularized by Eric Ries in his book *The Lean Startup*. Ries defines MVP as "that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort."

Let's take a moment to break this statement down. Obviously, our new product represents our modernization of the ACH dashboard, but what about "validated learning"? With validated learning, we are gaining knowledge about just how close we are to progressing toward a product that satisfies our customers. Validated learning should be evidence based and actionable, giving us something that we can use to improve our product in the next iteration. The "least effort" part does not mean producing something subpar but rather something achievable in the short term and with minimal investment.

Our goal with the MVP of the ACH dashboard is to provide our customers with a product that begins to address their concerns about the existing dashboard but ultimately helps us learn about the final product. This approach lets us transform the product from something that may have been developed in relative isolation to something usable by the customer (in larger companies, customer requests may have been filtered down to us through multiple layers, with each request putting the company's preferences on the product specs). Once our customers have the product in hand, we can use their feedback to help drive the direction of the product and provide the required features quickly. Compare this to other approaches where the entire product may be developed and delivered without meaningful feedback from the client base. We may end up with a product that is disliked because it is missing a feature customers may deem important, or it may not even be usable in a customer's environment due to their unique needs. Indeed, we have seen fully developed products that had to be shelved

because basic concerns about the problem at hand were not addressed or the environment in which they were needed to run was not considered. Table 7.1 provides a quick breakdown of key concepts regarding MVP, MLP, MMP, and MMF.

Table 7.1 MVP, MLP, MMP, and MMF key concepts

	MVP	MMP	MLP	MMF
Purpose	Tests and validates core assumptions	Creates a sellable version of the product	Delivers a product customers can connect with and love	Provides features that add value and can be marketed separately
Development approach	Prioritizes testing and learning	Balances functionality with market needs	Focuses on design and emotional engagement	Focuses on a single effective feature
Customer value	Provides core value to early adopters	Provides a complete solution to target market needs	Creates an emotional connection and stands out in the market	Provides incremental improvement to the existing product
Successful outcome	Validation of core assumptions	Sales and market acceptance	Customer love and brand loyalty	Adoption rate and user engagement

7.2 Preparing for the MVP

In the following sections, we ensure that all our previous work is integrated into the MVP. But before embarking on that journey, we review the individual pieces and parts that will make up our MVP—ACH parsing, database design, and UI—which were discussed in previous chapters. Figure 7.1 shows the components necessary for this part of the project.

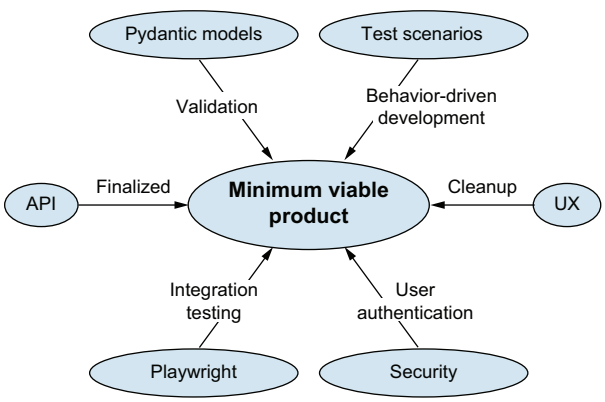


Figure 7.1 The pieces and parts fed into the MVP

If our initial work is done properly, the building of the MVP should be relatively straightforward. After all, we've done a fair amount of upfront work to get various components to this point. Remember, our previous research spikes built out the following components:

- An ACH parser in Python
- APIs written in Python using FastAPI that generated OpenAPI documentation
- Design of a Postgres database to store the ACH data
- UI written in Next.js

In the worst-case scenario, we may come across the proverbial "square peg into a round hole" type of situation where we did not quite document (or adhere to documented functionality), which may require some rework to get those pieces to fit correctly. We also have the opportunity to include some final touches such as using Playwright for integration/system testing, which will allow us to ensure the product functions from an end-user's perspective. Finally, we explore some additional security we may want to include in the project.

7.3 Building out the /files APIs

In chapter 4, you learned how to build APIs in Python using FastAPI. Then, we used Next.js and WireMock to create mock API responses that could be returned to the UI with the needed data. At this point, we want to verify that the backend and the APIs align properly.

First, we bring up the different components through our JetBrains IDE (PyCharm), which allows us to start our current database in Docker, API, and the Dashboard UI. By navigating to http://localhost:8000/docs (OpenAPI documentation), we can take advantage of the `POST` request we had built during our research (it may not be perfect but should load the file). Figure 7.2 shows the sample `POST` request that will also let us choose our sample.ach file.

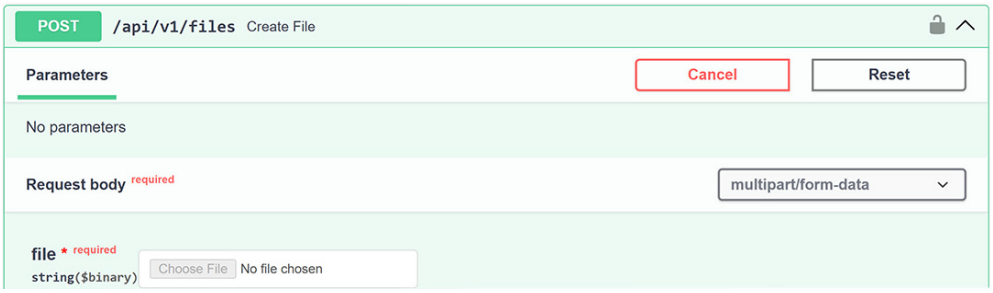


Figure 7.2 The sample `POST` request

If we refresh our page, the Recent ACH Uploads shows NaN (not a number) for many fields (see figure 7.3). There could be several reasons for getting NaN:

- *Parsing errors*—Parsing non-numeric data such as `parseInt("abcd")`
- *Undefined or null values*—Using a field from the API that does not exist
- *Arithmetic errors*—Dividing by zero

Let's see if we can pinpoint this particular "square peg into a round hole" problem.

Figure 7.3 Data being returned but displayed incorrectly

We can explore the data returned by the `/files` API by using the OpenAPI documentation page (or a tool such as Postman). The following listing shows the data being returned from the API.

Listing 7.1 Data being returned from `/files` endpoint

```
[
  {
    "ach_files_id": "dcb8e6a4-79e9-4cf7-a138-491e58c1ebb8",
    "file_name": "sample.ach",
    "file_hash": "application/octet-stream",
    "created_at": "2024-02-18T01:05:37.513025"
  }
]
```

If we compare this code to the interface for `AchUpload` from RecentAchUploads.tsx, we can see the differences. Not only are we missing fields such as `creditTotal` and `debitTotal`, but we also need to map some of the field names.

Listing 7.2 The `AchUpload` interface

```
interface AchUpload {
    id: number;
    date: string;
    filename: string;
    creditTotal: number;
    debitTotal: number;
}
```

We should also review how the database is configured and change the amount fields from the file control record (type 9) to be `NUMERIC` instead of `VARCHAR`. This is a somewhat personal choice as arguments may be made to have a more flexible character

value since the fields themselves can be formatted improperly. Making this type of change can be beneficial because the database will now enforce the data to be numeric. However, we may run into errors with existing functionality that worked before. Why? It's because we could have unknowingly been putting spaces or other non-numeric data into that field. Of course, hopefully, our unit tests would have prevented this or, at the very least, would have quickly pointed out possible adjustments.

In our case, we have tried to create a database structure that handles both unparsed records (for those times when the records are simply not parsable) and more detailed parsed records. The following listing shows the updated constraint for the numeric field. Note that the fields are 12 characters long, with a precision of 2.

Listing 7.3 ACH table for the file control record

```
total_debit_entry_dollar_amount NUMERIC(12, 2) NOT NULL,
total_credit_entry_dollar_amount NUMERIC(12, 2) NOT NULL,
```

The parsing program needs to be updated as well (see listing 7.4). We use the `Decimal` datatype since it is a bit safer than using a floating point (https://docs.python.org/3/library/decimal.html). Note that we pass the fields as a string to `Decimal` instead of converting it to an `int/float` to avoid accidental conversion. For instance, the results of `Decimal('0.1')` and `Decimal(0.1)` are `Decimal('0.1')` and `Decimal('0.1000000000000000055511151231257827021181583404541015625')`, respectively, so we still have to be careful.

Listing 7.4 Converting the string to a number that includes a decimal point

```
total_debit_entry_dollar_amount=
   Decimal(f"{record[31:41]}.{record[41:43]}"),
total_credit_entry_dollar_amount=
   Decimal(f"{record[43:53]}.{record[53:55]}"),
```

Parses the fields using a Decimal object and f-strings

Having to update our database to convert from storing a string to numeric where appropriate adds extra work. We could have cut down some of the work had we tackled it in our previous research spikes when working on the database. In addition, we may be asking ourselves whether it is worth the effort to convert these fields to numeric for an MVP. That is a valid question. While at the very least we get the benefit of the database enforcing the formatting of our data, the tradeoff is increased development time.

Unfortunately, there is no right answer. Depending on the project, requirements, and timeframe, you may not have the extra time to convert the fields and may choose to do it later. In this case, we convert the fields since we need to update the API to return these fields anyway. We can make the changes, and based on the interface defined in the UI (listing 7.2), we know that a numeric was expected.

Now that we have the data stored correctly, we need to return the fields as part of the query. That brings us to another area that has to be refactored. We created schemas and used them in both the database and to represent the data being returned from

our APIs. We would like to keep the models separate because this separation of concerns provides us with more flexibility, better security, and data integrity. We refactor our schemas directory to have folders for `api` and `database`, and we move all existing schemas to the database folder, which results in the structure presented in the following listing (note that not all files are shown).

Listing 7.5 New schemas directory structure

```
schemas/
├─ api/
├─ database/
|  ├─ ach_record/
|  |  ├─ ach_record_base_schema.py
|  |  ├─ ...
|  ├─ ach_file_schema.py
|  ├─ ...
```

With the updated structure, we are free to add our API responses. We can now continue with updating the API as we wanted. If you remember, we wanted to return a response with the file totals. Therefore, we created ach_files_response.py that contains the `AchFilesResponse`, as shown in the following listing. We also take the time to use more of Pydantic by providing documentation to the fields.

Listing 7.6 The `AchFilesResponse` class

```
from decimal import Decimal
from datetime import datetime
from pydantic import BaseModel, UUID4, Field


class AchFilesResponse(BaseModel):
    id: UUID4 = Field(...,
        description="Unique identifier for the ACH file",
        title="ID")
    date: datetime = Field(...,
        description="The date and time the ACH file was uploaded.",
        title="Date")
    filename: str = Field(...,
        title="Filename",
        description="The name of the file the ACH file was loaded from.",
        max_length=255,
    )
    credit_total: Decimal = Field(
        ...,
        description="The total amount of credit
➥ transactions in the ACH file.",
        ge=0,
        title="Credit Total",
    )
    debit_total: Decimal = Field(
        ...,
```

By using Field from Pydantic, we can specify both constraints and documentation.

A description of our field shown as part of the API doc

A title for the field shown in our API doc

A constraint on the field; strings longer than max_length will cause validation errors.

Constrains the field to be ≥0

```
        description="The total amount of debit
➥transactions in the ACH file.",
        ge=0,
        title="Debit Total",
    )

    class Config:
        json_schema_extra = {
            "example": {
                "id": "123e4567-e89b-12d3-a456-426614174000",
                "date": "2024-01-01T12:00:00",
                "filename": "ACH_20240101_123.ach",
                "creditTotal": "1000.00",
                "debitTotal": "5000.23",
            }
        }
```

We have previously defined a simple query to return all the fields from the `ach_files` table, but now that we want to include the credit and debit totals in the response, we need to pull them from the parsed type 9 record. To work out the query, we can use CloudBeaver to run queries against our database and ensure the results are correct. Once we start pulling the required data, we can create a new method to return the data retrieved.

Listing 7.7 Updating SQL query

```
def get_files_response(
    self, limit: Optional[int] = None, offset: Optional[int] = None
) -> list[AchFilesResponse]:
    with get_db_connection(row_factory=dict_row) as conn:
        result = conn.execute(
            """
            SELECT af.ach_files_id AS id,
                   af.file_name AS filename,
                   af.created_at AS date,
                   afcr.total_debit_entry_dollar_amount AS debit_total,
                   afcr.total_credit_entry_dollar_amount AS credit_total
            FROM ach_files AS af
            INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
            INNER JOIN ach_records_type_9 AS art9
               USING (ach_records_type_1_id)
            INNER JOIN ach_file_control_records AS afcr
               USING (ach_records_type_9_id)
            ORDER BY af.created_at DESC
            LIMIT %s
            OFFSET %s
            """,
            [limit, offset],
        )
        return result.fetchall()
```

Fields are renamed to match the AchFilesResponse.

When joining tables on columns that are named the same in both tables, we can employ USING.

Sorts our results by the created_at date in descending order

Although not used at the moment, we can limit the results and allow for paging through them when we have large amounts of data.

Finally, we update our route not only to call our new `get_files_response` but also to provide more documentation for our end-user, as shown in the following listing. This documentation step is important because it makes the API more accessible to nontechnical stakeholders, such as the QA department, who may be helping us with testing the APIs, or technical writers, who may be helping us with documentation. Furthermore, this change can benefit the API consumers who do not have the advantage of viewing the underlying code.

Listing 7.8 Updated API route

```
@router.get(
    "",
    response_model=list[AchFilesResponse],
    summary="Retrieve Uploaded ACH Files",
    description="Retrieve the details of an ACH
➥ file including credit/debit totals.",
    response_description="The details of the
➥ requested ACH file.",
)
async def read_files() -> list[AchFilesResponse]:
    return AchFileSql().get_files_response()
```

We can use Pydantic again to document our fields.

With this completed, we have a usable API with documentation. Figure 7.4 shows the results of some of the documentation we obtained using Pydantic. So, in addition to being able to assist us with data validation, parsing, and type safety, Pydantic also provides these documentation benefits.

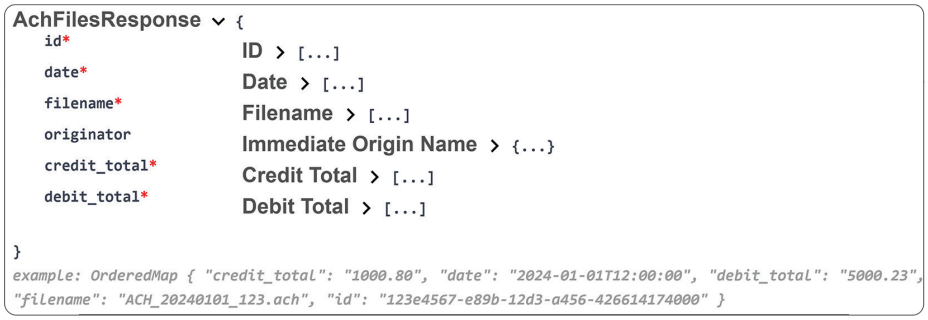


Figure 7.4 Our API now has documented fields and examples

We are almost there—just a few finishing touches on the UI side, and everything will be hooked together.

In the RecentAchUploads.tsx, we named our interface `AchUploads`. We would prefer to name that something closer to what the API is returning. We also have a mapping problem where we have fields defined as `creditTotal`, but they are being returned as `credit_total` from the API. While Python uses `snake_case` for variable names,

TypeScript prefers `camelCase`. So, we rename `AchUploads` to `AchFiles` and leave the fields as is. Then, we create a new interface—`AchFilesResponse`.

Listing 7.9 The `AchFilesResponse` interface

```
interface AchFilesResponse {
    id: number;
    date: string;
    filename: string;
    credit_total: number;
    debit_total: number;
}
```

Next, we update the call to the `/files` endpoint in the component to transform the response from `AchFilesResponse` to `AchFiles`. We do so to maintain consistent naming standards for the language we are using. However, this can also yield another benefit—providing a level of abstraction between the UI and API. This benefit can be valuable if we change the vendor of our API (not likely in this case), or something unexpectedly changes within the API (where the API did not maintain backward compatibility, or unannounced changes were made). The following listing shows the updated `axios` call.

Listing 7.10 Updated `axios` call

```
axios.get<AchFilesResponse[]>(`${apiUrl}/files`)
   .then(response => {
      console.log(`Response data
➥ ${JSON.stringify(response.data)}`);
      const transformedData: AchFiles[] =
➥ response.data.map((row: AchFilesResponse) => ({
         id: row.id,
         date: row.date,
         filename: row.filename,
         creditTotal: row.credit_total,
         debitTotal: row.debit_total
      }));
      setRows(transformedData);
   })
```

A simple debug message to see the data coming back

For each object returned, maps the fields to their new names

Saves the response so we can use it

We should now have a valid response from the API and no longer see NaN for our fields when viewing the page (figure 7.5).

Figure 7.5 Updated component for Recent ACH Uploads

When securing the ability to upload files, we worked through a few problems and did some manual testing to ensure the data was being returned as expected. We also took an opportunity to ensure our APIs were expanded with documentation and constraints where appropriate. In the next section, we take some time to explore supplementing our manual testing with some additional automated testing.

7.3.1 Testing the API

Since at this point our API is working already, let's take a look at how to incorporate some behavior-driven development (BDD) testing. Remember, this type of testing works best when the business, QA, and development all work together to create the testing scenarios. In our case, we are interested in creating a few scenarios that other groups may be able to use in their testing. We can create our features and scenarios as shown in listing 7.11.

The benefit of these types of tests is that other departments can plug in file names, along with credit and debit amounts, to test a variety of files. We would treat these tests as first-class citizens, meaning they take the same priority as the production code. Therefore, requests to expand the testing functionality are handled the same as production bugs and enhancement requests. For example, the tests employed here only assess a small fraction of required scenarios. We should expect to see requests to get the total number of records processed or the number of batches, check for exceptions, and so on.

Listing 7.11 BDD-style scenarios

```
Feature: The /files endpoint
  Test the functionality of the /files endpoint

  Scenario: I want to get a list that contains a single file
    Given that I have a clean database
    And that I have posted the file "sample.ach"
    When I request a list of files
    Then I should have a single file named "sample.ach"
    And it should have a total credit amount of "1146114.80"
    And it should have a total debit amount of "78.25"
    And there should be no exceptions

  Scenario: I want to get a list of files
    Given that I have a clean database
    And that I have posted the file "sample.ach"
    And that I have posted the file "sample1.ach"
    When I request a list of files
    Then I should have a file that includes the file "sample.ach"
    And I should have a file that includes the file "sample1.ach"
    And there should be no exceptions
```

For now, let's see how to power these examples in Python code. Listing 7.12 shows some setups for these scenarios. We use the scenarios method to pull features we want to test (obviously, they should be making use of the grammar we have created). We define an

instance of the `AchFileProcessor` because we are going to use it directly when parsing the file. We could have possibly gained some more functionality from using `POST` on the file, but since we are testing a single API, we did not necessarily want to muddy the water by relying on another API to work and parse the file. Note that even though we used `And` in the previous feature (e.g., `And that I have posted the file`), we use `Given` when annotating the method. This is because the `And` acts as syntactic sugar and will pick up the previous `Given/When/Then` clause. You will see that in play again when we look at the `Then` clauses.

Listing 7.12 Python code for BDD

```
...
client = TestClient(app)

scenarios("../features/ach_files_endpoint.feature")

@pytest.fixture
def ach_file_processor() -> AchFileProcessor:
    return AchFileProcessor()

@pytest.fixture
def api_response():
    return {}

@given("that I have a clean database")
def truncate_database():
    SqlUtils.truncate_all()

@given(parsers.re(r'that I have posted the file
➥ "(?P<ach_file>.*)"'))
def parse_the_given_file(ach_file):
    dir_path = os.path.dirname(os.path.realpath
➥(__file__))
    file_path = os.path.join(dir_path, "../data",
➥ ach_file)
    parser = AchFileProcessor()
    ach_files_id = SqlUtils.create_ach_file_record(ach_file,
➥ str(randint(1, 99999999)))
    parser.parse(ach_files_id, file_path)
...
```

Imported from starlette.testclient, the TestClient is needed for making HTTP requests.

Loads scenarios associated with these step definitions

Creates the AchFileProcessor as a pytest.fixture; we can use it as needed.

api_response provides a way for us to store our results between steps.

This @given statement employs our SQL utility to clear all tables.

Here we take the desired filename from disk and parse it.

To maintain our database integrity, we need to create an ach_file_record and use it as a reference in the parse routine.

In the following listing, we execute the API and save the response. The response will be needed later when checking it against the conditions we defined with `Then`.

Listing 7.13 `When` BDD code

```
...
@when("I request a list of files")
```

No need to do anything special for hardcoded strings

```
def request_files(api_response):
    response = client.get("/api/v1/files")
    assert response.status_code == 200, response.text
    print(response.json())
    api_response["response"] = response.json()
…
```

We include the api_response fixture to store our results.

The client is used to retrieve the data from the endpoint.

We assert that we received a normal response code.

Prints the JSON response just for a sanity check

Stores the response to be used when we evaluate our @then statements

Finally, we can see our defined `Then` conditions, which validates our testing. We use a mixture of parsers depending on the situation. For the amounts, it is possible to use the `parsers.parse`. For more complicated tasks (such as parsing the filename), we can use `parsers.re`, which supports regular expressions. However, we generally recommend keeping things simple.

All of these methods use simple `assert` statements to accomplish the needed validation. The following listing shows how to confirm the filename was returned by looking at all the fields in the API response to find a field named `filename` and then ensure (`assert`) it matches the `filename` specified in the BDD test.

Listing 7.14 `Then` BDD code

```
@then(parsers.re('I should have a file that
➥ includes the file "(?P<filename>.*)"'))
def then_file_check(filename, api_response):
    assert any(
        response["filename"] == filename for response
            in api_response["response"]
    ), (f"Expected {filename} in"
➥f" {api_response['response']}")

@then(parsers.parse('it should have a total credit
➥ amount of "{credit_amount}"'))
def then_total_credit(credit_amount, api_response):
    response = api_response["response"][0]
➥["credit_total"]
    assert credit_amount == response, f"Expected
➥ {credit_amount} in {response}"
```

Using a regular expression to extract the filename

The filename we extracted and the pytest fixture are parameters.

Checks whether the desired filename is in the array

An example of using parse to get the credit_amount

The credit_amount and pytest fixture are parameters.

Gets credit_total for the first and only file in the response

Ensures the credit total from the response matches the expected amount

You should now have a better idea of how to incorporate this type of testing into your projects. Giving everyone the ability to contribute to testing will not necessarily make it more fun, but it will certainly help improve the robustness of our test suite. It takes

little knowledge about the inner workings of testing and allows testers to focus on creating tests in a natural way.

At this point, the `/files` APIs for the MVP are completed, and we move on to building other required APIs in the next section.

7.4 ACH batches APIs

When we built the UI, we also created an API to retrieve batch information for a particular file. However, that endpoint was not something we created during our initial research on FastAPI. Instead, we found a need for it when performing our UI research and building the data with WireMock. This is something that we find in projects often—we need multiple perspectives to obtain a successful result. From subject matter experts to developers completing the work, everyone has different priorities and perspectives on how to complete a project. At this point, we want to investigate how to build an API to retrieve that data.

7.4.1 Adding the /files/id/batches API

With our `/files` API, we took an opportunity to revisit the BDD-type syntax for testing our API because the API has already been built. Here, we start with some tests to illustrate the TDD approach (albeit, a somewhat loose approach). First, we build a simple test class.

Listing 7.15 Test class for the `/files/id/batches`

```
class TestAchBatchesApi:
    client: TestClient = TestClient(app)
    ach_files_id: Optional[str] = None

    def setup_method(self, method: Callable) -> None:
        ach_file = "../data/sample.ach"
        SqlUtils.truncate_all()
        self.ach_files_id = SqlUtils.create_ach_file_record(
            ach_file, str(randint(1, 99999999))
        )
        AchFileProcessor().parse(self.ach_files_id,
➥ "data/sample.ach")

    def test_get_batches_api(self):
        response =   self.client.get("/api/v1/files/{self.
➥ach_files_id}/batches")
        assert response.status_code == 200,
➥ response.text

    def teardown_method(self, method: Callable)
➥ -> None:
        print(f"\nTeardown for {method.__name__}
➥ test method execution")
```

Defines the test client and a variable to hold the file_id

This setup method parses the given ACH file, and the database should be ready.

At this point, we are simply looking to ensure we get a valid response from the API.

For completeness' sake, this method executes after the test is run.

Can you guess what the status code will be? The answer is a `404`, because we have not built the API yet. Running into `404 errors` is typical during a development cycle where

we may not have endpoints defined (as is the case here), or systems are in flux (perhaps another team changed their endpoint, or the machine is temporarily offline). The point is that during development, `404 errors` are somewhat innocuous but are a lot more concerning in production, when something that was supposed to be there is not.

In our case, it should be simple to define the endpoint in our files.py and return a hardcoded string. The following listing shows the results of that minimal code.

Listing 7.16 Defining the `/files/id/batches` endpoint

```
@router.get("/{file_id}/batches")
async def read_batches_from_file(file_id: UUID) -> str:
    return f"File {file_id} batches"
```

A simple method to return a string confirming we received the file_id

With that, we can rerun our test and should be receiving a `200`-response code. We can now venture on and build the rest of the code. Since the other endpoints make use of our `AchFileSql` class, we can add another method called `get_batches`, as shown in the following listing. Adding another method helps keep our code focused on the task at hand, which provides better maintainability and scalability going forward. Compare this to an approach where maybe we have one massive SQL query to gather all the data we could ever want about our SQL file every time, even though we only needed a tiny fragment of the data.

Listing 7.17 Updating our endpoint to call `get_batches`

```
@router.get("/{file_id}/batches")
async def read_batches_from_file(file_id: UUID) -> list[AchBatchesResponse]:
    return AchFileSql().get_batches(file_id)
```

Calls the method to get batches based on our file_id

Updates the return from a string to an AchBatchesResponse (which we have not created yet)

Next, we create the `AchBatchesResponse`. When we create this layout, we will also need to review the database tables associated with this data. Namely, the ACH catch control record (type 5). When building this response, we can take a look at the ACH batch control record layout and see how a correct file should be formatted. This obviously affects how we build the `AchBatchesResponse` class.

Listing 7.18 The `AchBatchesResponse`

```
from decimal import Decimal
from pydantic import BaseModel, UUID4, Field


class AchBatchesResponse(BaseModel):
    id: UUID4 = Field(
```

Creates a class inheriting from the BaseModel provided by Pydantic

```
        ...,
        description="Unique identifier for the ACH batch",
        title="ID"
    )
    company_name: str = Field(
        None,
        description="The name of the company.",
        title="Company Name"
    )
    batch_number: int = Field(
        ...,
        title="Batch Number",
        description="The number associated with the batch.",
        ge=0,
    )
    credit_total: Decimal = Field(
        ...,
        description="The total amount of credit
➥ transactions in the ACH batch.",
        title="Credit Total",
    )
    debit_total: Decimal = Field(
        ...,
        description="The total amount of debit
➥ transactions in the ACH batch.",
        title="Debit Total",
    )
    entry_addenda_count: int = Field(
        ...,
        description="The number of addenda records in the ACH batch.",
        title="Entry Addenda Count",
        ge=0,
    )

    class Config:
        json_schema_extra = {
            "example": {
                "id": "123e4567-e89b-12d3-a456-426614174000",
                "company_name": "Company Name",
                "batch_number": 1,
                "credit_total": "100.00",
                "debit_total": "100.00",
                "entry_addenda_count": 2,
            }
        }
```

Creates fields as necessary using integers where appropriate

Creates decimal fields for monetary values

Updates our example accordingly; notice the monetary values are maintained as strings to avoid floating point errors.

As shown in the following listing, there is a need to update the record description to better align with both the `AchBatchesResponse` from listing 7.18 and with the ACH record layout in general.

Listing 7.19 Updated batch control SQL table

```
CREATE TABLE ach_batch_control_records (
    ach_records_type_8_id UUID UNIQUE NOT
```

```
➥ NULL REFERENCES ach_records_type_8(ach_records_type_8_id)
➥ ON DELETE CASCADE ON UPDATE CASCADE,
    record_type_code VARCHAR(1) NOT NULL,
    service_class_code NUMERIC(3) NOT NULL,
    entry_addenda_count NUMERIC(6) NOT NULL,
    entry_hash NUMERIC(10) NOT NULL,
    total_debit_entry_dollar_amount NUMERIC(12,2) NOT NULL,
    total_credit_entry_dollar_amount NUMERIC(12,2) NOT NULL,
    company_identification VARCHAR(10) NOT NULL,
    message_authentication_code VARCHAR(19) NOT NULL,
    reserved VARCHAR(6) NOT NULL,
    originating_dfi_identification VARCHAR(8) NOT NULL,
    batch_number NUMERIC(7) NOT NULL #A
);
```

Updates the table to support the numeric values

Figure 7.6 shows a trimmed-down version of the OpenAPI documentation reflecting these updates.

Figure 7.6 OpenAPI documentation for retrieving ACH batches

Remember, this page not only provides a great way to document our APIs but also to test and validate them. We can use this web page to `POST` an ACH file, `GET` the list of files, and then `GET` the batches. While we have already implemented unit tests to validate the APIs, this is a great way to do some more integration and manual testing of our APIs. After loading our standard sample.ach file, we should see data being returned, as in the following listing. It is worth noting again that having sample data as defined in our sample.ach file is beneficial for many reasons. Sample data can help us test edge cases and allows more robust test scenarios. Another important benefit is that it is *test* data—we are not worried about exposing nonpublic information (NPI) or personally identifiable information (PII).

Listing 7.20 Sample batch data being returned

```
[
  {
    "id": "e89511cc-9791-4318-b3b7-76406c191442",
    "company_name": "",
    "batch_number": 4,
    "credit_total": "114609873.00",
    "debit_total": "0.00",
    "entry_addenda_count": 32
  },
  {
    "id": "92ece8a1-e9b8-4bf6-b3df-416d7efa2909",
    "company_name": "CCD coname",
    "batch_number": 2,
    "credit_total": "201.00",
    "debit_total": "404.00",
    "entry_addenda_count": 6
  },
…
```

It is important to note that the changes to the database allow us to rely on the database (which is our source of truth) and not necessarily the business rules we defined in Python with Pydantic. We will find this helpful as our project grows, and the carefully laid out business rules that we have in one module are bypassed when another project chooses to use its own path. Both code and technical reviews are needed to ensure that, for example, the requirement that all code must access the database through a particular module will be met.

Tools such as `pylint` allow us to build a plugin that can be used to detect these situations. While perhaps a bit beyond the scope of the book, we would like to refer you to listing 7.21, which provides the basics of a plugin that looks for imports with `psycopg`. While this is not comprehensive detection, hopefully, it will fire our imagination and give us something to dive into and research further uses of this type of technique. While code and technical reviews are extremely useful, it is nice to have guardrails that can keep us on track before we get to a final code review, only to find out we need to rework something. For more information on creating a `pylint` plugin, visit https://mng.bz/MDjn.

Listing 7.21 A simple `pylint` plugin to detect `psycopg` use

```
import os
from typing import TYPE_CHECKING
import pdb
from pylint.checkers import BaseChecker

ALLOWED_MODULES = ["ach_file_sql"]
ALLOWED_DIRECTORIES = ["ach_processor/database"]

if TYPE_CHECKING:
    from pylint.lint import PyLinter
```

The necessary imports for our plugin

Defines either modules that we allow psycopg to be referenced from or entire directories

```
class PsycopgImportChecker(BaseChecker):

    name = "psycopg-import-checker"
    msgs = {
        "E9999": (
            "psycopg usage only allowed in specific modules",
            "psycopg-disallowed-import",
            "Used when psycopg is imported outside
➥ designated modules.",
        ),
    }
    options = ()

    def visit_import(self, node):
        if any(modname == "psycopg" for
➥ modname, _ in node.names):
            self._check_psycopg_usage(node)
        elif any(modname.startswith("psycopg")
➥ for modname, _ in node.names):
            self._check_psycopg_usage(node)

    def visit_importfrom(self, node):
        if node.modname.startswith("psycopg"):
            self._check_psycopg_usage(node)

    def _check_psycopg_usage(self, node):
        module_name = node.root().name
        module_file = node.root().file
        if module_name in ALLOWED_MODULES:
            return  # Allow usage in specific modules

        for allowed_dir in ALLOWED_DIRECTORIES:
            # Ensure paths are absolute or relative to
➥ the project root
            allowed_path = os.path.normpath(allowed_dir)
            module_path = os.path.normpath(module_file)
            if allowed_path in module_path:
                return

        self.add_message("psycopg-disallowed-import",
➥ node=node)

def register(linter):
    linter.register_checker
➥(PsycopgImportChecker(linter))
```

Defines the pylint error that we will produce

Methods executed when "import" and "import from" statements are encountered; here, they call our _check_psycopg_usage method.

A method that gets the name and file we are working in and checks whether it has been whitelisted (found in ALLOWED_MODULES or ALLOWED_DIRECTORIES)

Adds the error message If the method is not found

Registers our module

While not strictly necessary for our project, a mantra of "Hey, I can automate that" can help your growth as a developer. Integrating this check with `pylint` ensures we can prevent the misuse of the module (in terms of where it is imported) throughout our code base.

With that, we are ready to switch gears and move over to the UI side to ensure we can access our endpoint.

7.4.2 Supporting ACH batches in the dashboard

Now that we have an API capable of returning ACH batches, we can integrate it into our UI. Remember, when mocking out the API, we created a data structure similar to what is presented in the following listing. However, now that we actually have the data being returned, we need to update the UI to accommodate the new format. Note that if you were to start the UI, upload a file, and try to drill down into it, because of these format changes, you would receive an error message similar to `TypeError: Cannot read properties of undefined (reading 'map')` on the `batches.map` statement. We will resolve that problem in this section.

Listing 7.22 Format for batch information

```
{
   "batches" : [
       {
          "companyName": "example name",
          "companyBatches": [
             {
                "batchId": "value"
                "batchNumber": "value"
                "recordCount": "value"
                "totalDebitAmount": "value"
                "totalCreditAmount": "value"
             }
          ]
        }
}
```

While we could have returned the data the same way we mocked it up, it may make our lives easier to return a simpler structure from our API that could then be manipulated with the results. This is all part of putting an MVP together. We have not invested too much into the solution, and it is not too late to adjust the game plan. A bit of rework early on can ensure we stay on the right path and prevent headaches down the road.

Our choice of SQL, Python, or TypeScript will likely not matter to the end-user. For now, we opt for the simple SQL query, and format the data on the UI side. We update our interfaces to have one dedicated to the API response, and we use the rest throughout the code.

Listing 7.23 Updating interfaces for ACH batches

```
export interface AchCompanyBatchInfo {
    companyName: string;
    companyBatches: AchBatchInfo[];
}

export interface AchBatchInfo {
    id: string;
    batchNumber: number;
```

The ACH batch info interface that we will use throughout the UI

```
    debitTotal: Decimal;
    creditTotal: Decimal;
    recordCount: number;
}

export interface AchBatchInfoResponse {
    id: string;
    company_name: string;
    batch_number: number;
    debit_total: string;
    credit_total: string;
    entry_addenda_count: number;
}
```

The ACH batch info interface that we will use throughout the UI

The response from the API that we will rework

We also need to add the code that will condense the response into the `AchCompanyBatchInfo`, which mimics our original structure. As shown in the following listing, we organize our records by company name, creating an array of batches for each company.

Listing 7.24 Transforming the code into the desired structure

```
function createAchCompanyBatchInfoRecords
➥(response: AchBatchInfoResponse[]): AchCompanyBatchInfo[] {
    const condensedRecords: Record<string,
➥ AchCompanyBatchInfo> = {};

    response.forEach((record: AchBatchInfoResponse) => {
        if (!condensedRecords[record.company_id]) {
            condensedRecords[record.company_id] = {
                companyId: record.company_id,
                companyBatches: []
            };
        }

        condensedRecords[record.company_name]
➥.companyBatches.push({
            id: record.id,
        companyId: record.company_id,
            companyName: record.company_name,
            batchNumber: record.batch_number,
            debitTotal: new Decimal(record.debit_total),
            creditTotal: new Decimal(record.credit_total),
            recordCount: record.entry_addenda_count,
        });
    });

    return Object.values(condensedRecords);
```

Defines a simple Record object to hold our condensed records by Company ID

Checks whether we have an entry with the key of company_id; if not, it creates one with an empty array for companyBatches.

Adds the batch information for the company

Our logic for the scatterplot we created remains largely the same with just an update to the names of the interfaces and a couple of minor tweaks. Preserving consistency in naming helps with code maintenance and readability, especially as other developers may make enhancements. The most important change we need to keep in mind is

that now that we are using `Decimal`, we have to update the way in which we create our field `totalCreditAndDebits`. As shown in the following listing, we need to call the `add` method instead of using the standard `+` (plus) operator.

Listing 7.25 Updating the computation for `totalCreditAndDebits`

```
totalCreditAndDebits: companyBatch.creditTotal.add(companyBatch.debitTotal)
```

7.4.3 Uploading files

So far, we have been using our API documentation to upload files. The final functionality that we want to look at is being able to upload from our dashboard. We have already built the UI capable of dragging a file into the browser and obtaining the file name. We also had a button that did not do anything when clicked. Now we will provide code that opens a window where the user can select a file to upload.

We know there will be at least two paths to uploading a file—dropping a file into the browser and clicking the upload button—which means we need to create a function to handle the uploads. If we can centralize the function, it will prevent duplicating code and maintenance in the future

Listing 7.26 The `uploadFile` function

```
...
    const [isLoading, setIsLoading] =
➥useState<boolean>(false);
...
    const uploadFile = (file: File) => {
        const formData = new FormData();
        formData.append('file', file);

        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        setIsLoading(true);
        axios.post(`${apiUrl}/files`, formData, {
            headers: {
                'Content-Type': 'multipart/form-data',
            },
        })
            .then((response) => {
                console.log('File uploaded successfully', response);
            })
            .catch((error) => {
                console.error('Error uploading file', error);
            }).finally(() => {
            setIsLoading(false);
            route.replace('/');
        });
    }
```

Defines a state to track whether the file is loading

Gets the file from the form

Uses our URL

Sets loading to true so that we can update the page to display an activity indicator

Sends a post request with our file to the endpoint

When the file is uploaded, sets loading back to false and immediately goes back to the dashboard

This feature allows us to easily update the previous code we had for `handleDrop` when a user drops in the file. As soon as the file is dropped, we call `uploadFile` with the file,

and when the file is uploaded, we will route the user back to the dashboard. In this simple case, the dashboard will be automatically updated with the new file. However, we have the following limitations:

- We only upload one file, although a user could conceivably pick more.
- We assume the file was loaded successfully.

Either way, we do not necessarily provide any feedback to our users. We won't focus on these problems because we are working on an MVP, and we do not need to have all the bells and whistles, except in a finished production-ready product. While it may be tempting to add this functionality, we want to avoid scope creep (when our project goes beyond the original requirements), which can cause delays in rolling out the product. For now, let's add the call to our `handleDrop` function.

Listing 7.27 The `handleDrop` function

```
    const handleDrop = useCallback((event:
➥ DragEvent<HTMLDivElement>) => {
…
        if (files.length > 0 && files[0].name !== '') {
            console.log(files);
            setFileInfo(files[0]);
            uploadFile(files[0]);
        }

    }, []);
```

Updates the handleDrop code to call our routine to upload the file

We also call this code from the button click as well, which takes a little more work.

Listing 7.28 Uploading files with our button

```
…
    const fileInputRef = useRef<HTMLInputElement>
➥(null);
…
const handleFileChange = (event:
➥ChangeEvent<HTMLInputElement>) => {
        if (event.target.files &&
➥event.target.files.length > 0) {
            const file = event.target.files[0];
            console.log(file);
            uploadFile(file);
        }
    };
…
    const handleButtonClick = () => {
        fileInputRef.current?.click();
    };
…
         {isLoading ? <CircularProgress/> : <Typography
```

Defines a reference to an HTMLInputElement, which allows us to interact with the element in our code

Handling of the file changing

Executes the click event, which implements a click on the "input" HTML element, making the File Chooser open

```
            variant="h6">
               {fileInfo?.name ?
➥fileInfo.name : 'Drag and drop files here'}
            </Typography>}
…
    <Button onClick={handleButtonClick}
            startIcon={<CloudUpload/>}>
       Upload
    </Button>
    <input
       type="file"
       ref={fileInputRef}
       onChange={handleFileChange}
       style={{display: 'none'}}
    />
```

Specifies our handleFileChange method when the input changes

With this code, we can now upload files through our dashboard.

7.5 Putting the puzzle together

We now have the MVP up and running in our IDE environment. The final step is to move everything into the Docker environment and ensure things work.

From our previous preparations, we already know that all the pieces have been operating in Docker. This means we can use our previous Docker configuration with just a few tweaks, and we should be good to go.

Figure 7.7 shows the Docker layout, with containers and ports labeled to give us the basic idea of our layout. CloudBeaver is not necessary in our MVP—we only include it in the setup so we can work with queries and explore the database. With our configuration, we can start Docker as usual, with `docker-compose down; docker-compose up --build`.

7.5.1 Cleaning up the configuration files

When we begin getting our code ready for a production environment (or Docker), some additional housekeeping will be required. First, we can take a look at main.py in the `app` folder. Here, we specified some CORS settings. The allowed origins are likely going to be the biggest problem, and we want to ensure that requests are coming from where we expect them (in this case, `localhost:4000`). We also limit our methods to those expected (`GET` and `POST`). We should not need to allow `OPTIONS`, which is sent by the browser as part of a Preflight CORS request, but keep in mind that different frameworks/versions may require it.

Listing 7.29 Updates for CORS

```
origins = [
    "http://localhost:4000",  # For local development
    # "*" # Allow all, but not recommended for production
]
…
app.add_middleware(
```

```
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["GET", "POST"],
    allow_headers=["*"],
)
…
```

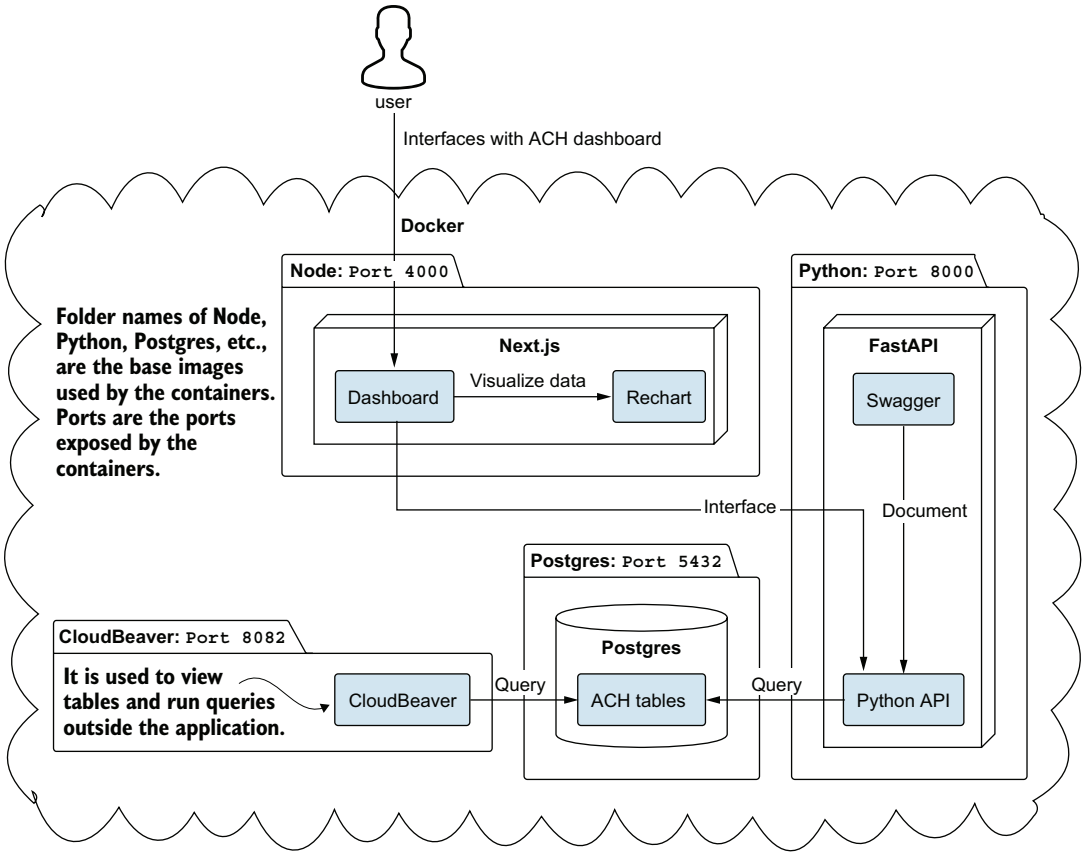


Figure 7.7 Docker setup for the ACH dashboard MVP

Another part of the Python configuration we may want to clean up is the requirements .txt. We have dependencies that are specific to our development, such as `pytest` and `pytest-bdd`. In our production code, there should be no reason to include these. Attackers cannot exploit dependencies that are not present, so it is important to keep unnecessary or unused requirements out of production. We create a dev-requirements .txt, which we can use for development environments as needed, and try to leave a requirements.txt that is production ready. If your IDE (or other tooling) maintains a

requirements.txt, you may find it easier to do the reverse and keep a prod-requirements .txt. This way, you can move any requirements you install over there immediately and do not necessarily need to guess. The following listing shows an example of a dev -requirements.txt.

Listing 7.30 A sample dev-requirements.txt file

```
-r requirements.txt
pytest==8.0.1
pytest-bdd==7.0.1
iniconfig==2.0.0
pluggy==1.4.0
```

We clean up the UI configuration files similarly by splitting tsconfig.json into tsconfig .dev.json and tsconfig.prod.json. This is because we do not need references to Jest and other development options such as `sourceMap` set in our production environment. The following listing shows how the tsconfig.prod.json has been updated to remove entries for Jest and expand to exclude any tests from the production build. We would not expect any tests to make it to the packaging steps, so excluding the *.test.ts and *.spec.ts files is mostly precautionary.

Listing 7.31 Updating the tsconfig.prod.json file

```
{
  "compilerOptions": {
…
    "sourceMap": false,
    "types": [],
…
  "exclude": ["**/*.test.ts", "**/*.spec.ts", "node_modules"]
}
```

In addition to configuration files for the components, we also want to take a minute to ensure that the `NEXT_PUBLIC_API_URL` is pointing to the correct location. The following listing shows the configuration for the dashboard service from docker-compose.yml.

Listing 7.32 Configuration for the dashboard service from docker-compose.yml

```
  dashboard:
    build:
      context: ./dashboard
      dockerfile: Dockerfile
      args:
        - NEXT_PUBLIC_API_URL=http://localhost:8000/api/v1
    ports:
      - 4000:3000
```

With that, we should be able to bring our project up in Docker.

7.5.2 Ensuring APIs are accessible

We can run the general check of our containers with `docker ps --format "table {{.Names}}\t{{.Status}}"` (we use the format option just for clarity; `docker-compose ps` will work as well, and it is shorter) and see that our containers are up and running.

Listing 7.33 Containers up and running

```
NAMES                     STATUS
dockermvp-dashboard-1     Up 58 minutes
dockermvp-api-1           Up 58 minutes
dockermvp-cloudbeaver-1   Up 58 minutes
dockermvp-postgres-1      Up 58 minutes (healthy)
```

The postgres container is the only showing as healthy because it is the only container we implemented a health check for.

Now that everything is working, note that we only have a health check defined for Postgres, so we see that it is healthy, but the other containers have no health checks, so we do not see the `(healthy)` text. Since all our containers appear to be up and running, the next thing we do is navigate to http://localhost:8000/docs and verify our APIs are available (see figure 7.8).

Figure 7.8 Our list of defined APIs

7.5.3 Programming challenges

The process itself should hopefully be clear by now—we identify desired functionality, create an API to support it, backend code to enable it, and frontend code to consume and display it (figure 7.9)

We still have several items to discuss in this chapter, but now is a good time to put out some challenges you could work on:

- *Top files today*—Provide a visualization of the largest files loaded today.
- *Today's totals*—Provide a component showing credit/debit totals for today.

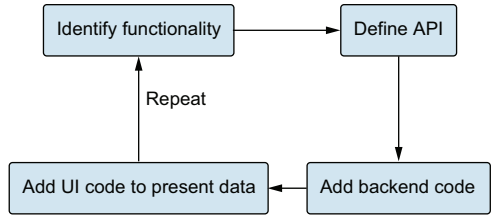


Figure 7.9 Process of adding functionality to our MVP

- *Drill down*—Allow the user to click a file, view batch information, and drill into the batch to see entries.
- *Transaction code breakdown*—When viewing the entries, provide a visualization of the breakdown of transaction codes (debits, credits, etc.) used.

7.5.4 Exploring the dashboard

At this point, we should have all the components up and running in Docker, and we can navigate to the dashboard. This would be the time to click everything to ensure it functions as intended. Load some ACH files, and make sure you can view the files, batches, and the charts are working. Then take a minute to congratulate yourself—the MVP is just the start of our journey, but we put some effort into getting to this point. The pace of development and the timeline for projects is often frantic. It is important that we take the time to recognize our achievements, as well as those of our team.

7.6 Gotta have more files

During development, there will come a time when we get to a point where the ACH files provided by the business are inadequate for our testing. This may be because we have specific scenarios we want to test, or we need to slim down files so we can test simple scenarios to ensure our data is working. Since ACH files are just text files, nothing is stopping us from simply copying and opening them to make the necessary updates. Fortunately (or unfortunately), we are programmers, and we cannot bother with editing files manually (except in rare cases). We can use BDD-style testing to develop syntax and create files.

Using Python and `pytest-bdd` to create our ACH files gives us not only a chance to work on our Python skills but also a chance to dive into ACH files and some of the requirements. By creating files programmatically and then turning around to load them into our dashboard, we are back to working in short development cycles. As we create and load files, we may find problems with the generation, loading/parsing, APIs, and UI. We can create an `ach_file_creation.feature` and generate a syntax for ACH file creation. The following listing shows a sample of the first test. Remember that chapters 2 and 5 reviewed some of the characteristics of ACH files. If you need additional

information on required fields and other important information, refer to the ACH File Overview at https://achdevguide.nacha.org/ach-file-overview.

Listing 7.34 BDD-style ACH file creation

```
  Scenario: Create an ACH file with a single batch and a single debit
    Given I want to create an ACH file named "ppd-single-debit.ach"
    And I want to have an immediate destination of "123456789"
    And I want to have an immediate origin of "987654321"
    And I want to have 1 batch with ACH debits only
➥  and a standard entry class code of "PPD"
    And I want 1 entries per batch with random amounts between 100 and 100
    And I want to use individual names of "John Doe"
    And I want to have company name "My Company" and
➥  company id "1234567890"
    When my ACH is created
    Then I should have a file named "ppd-single-debit.ach"
    And there should be 1 batch in the file
    And there should be 1 entries in the file
```

This code gives us some flexibility in the number of batches and entries a file will contain, as well as amounts used. It does not necessarily reflect/enforce the correct business rules for producing ACH files. We simply took an existing file and began working backward to replace fields as we needed some flexibility in our file creation. The file exists as a "test" within our tests folder for our project, but it could very well be split out into its own project to become its own utility others will use. We'll come back to this topic and work on it further as we continue with the project and find additional requirements.

Most of the code for supporting the grammar is centered around collecting the necessary information. In listing 7.35, we set it so that we are only expecting debits, and therefore, we can also set the appropriate service class and transaction codes. We could have also allowed users to specify transaction codes and then determined which service class code to use or vice versa. We can always approach problems differently and need to weigh the tradeoffs. Keeping in mind that nontechnical users may want to use this feature, abstracting the details of a correct service class and transaction code seems to be a good choice. Of course, later, we may want to introduce files produced with an error condition to create various scenarios and therefore need to introduce finer-grained control.

Listing 7.35 Sample for creating files

```
@given(
    parsers.re(
        r'I want to have (?P<batch_count>\d+)
➥  batch with ACH debits only and a standard entry
 class code of "(?P<standard_entry_class>.*)"'
    )
)
```

```
def set_number_of_debit_batches_to_create(
    setup_info, batch_count, standard_entry_class
):
    setup_info["batch_count"] = int(batch_count)
    setup_info["service_class_code"] = "225"
    setup_info["transaction_code"] = ["27"]
    setup_info["standard_entry_class"] = standard_entry_class
```

Gets the batch count, converts it to an integer, and stores it for later

Gets the standard_entry_ class and saves it for later

We are working with debits only, so we set the fields accordingly.

Similarly, the validation routine is straightforward. In this case, we know that we have a set number of records, and therefore, we want to simply count the records that are type 6. We perform a similar check for the number of batches. Opening, reading the entire file, and closing the file each time may not be the most efficient way to do things but is certainly sufficient for now.

Listing 7.36 Validating our entries

```
@then(parsers.parse("there should be
➥ {expected_entry_count:d} entries in the file"))
def validate_entry_count(setup_info, expected_entry_count):
    count = 0
    with open(f"../output/{setup_info['filename']}",
➥ "r", encoding="utf8") as f:
        for line in f:
            if line.startswith("6"):
                count += 1
    assert (
        count == expected_entry_count
    ), f"Expected {expected_entry_count}, but got {count}"
```

Opens the file for reading

Counts lines that are a type 6 record

Asserts that the count is correct by comparing it to what was expected

Finally, we show some of the work needed to create a record in the file. In the following listing, we format amounts and pad them as necessary before finally plugging them into the returned string.

Listing 7.37 Creating a file

```
def create_file_control(setup_info, total_debits_in_file,
➥ total_credits_in_file):
    total_credits_in_file =
➥ str(total_credits_in_file).rjust(12, "0")
    total_debits_in_file =
➥ str(total_debits_in_file).rjust(12, "0")
    batch_count =
➥ str(setup_info["batch_count"]).rjust(6, "0")
```

Fields are right justified with a max length and padded with zeroes.

```
    entry_count =
➥ str(setup_info["entry_count"] *
➥setup_info["batch_count"]).rjust(
        8, "0"
    )
    return return f"9{batch_count}000010{entry_count}0198019800
➥{total_debits_in_file}{total_credits_in_file}".ljust(94," ")
```

We may have invested more time in putting all this together, but we still believe these examples can improve our development skills and streamline our work later.

7.7 Adding a basic login page

The dashboard is a great way to get us accustomed to loading and visualizing ACH files. However, we do not want anyone to have access to the upload page. Whether the dashboard is available as a web interface to the outside world or is for internal use only, we still want to limit users who can work with files. Later, this may facilitate functionality such as alerts when a file is successfully loaded or rejected. In addition, an audit trail of someone who loaded a file or performed some meaningful activity to the file (such as deleting batches and their associated transactions) is important in case we have disgruntled or malicious users.

Thus, this section covers adding a basic login page using NextAuth.js, which is an open source authentication solution for Next.js. While we will have hardcoded values, you can extend the solution to use other providers such as Auth0, Apple, and Google, to name just a few.

We can set up the NextAuth.js with a few files and supporting code. We create `middleware.ts`, `auth.ts`, and `auth.config.ts`, which support the authentication.

Listing 7.38 Setting up our middleware

```
export default NextAuth(authConfig).auth;

export const config = {
    matcher: ['/uploads'],
};
```

We provide basic configuration that supports callbacks for checking whether a user is authorized and redirecting them as necessary.

Listing 7.39 `Auth` callbacks

```
import type { NextAuthConfig } from 'next-auth';

export const authConfig = {
    debug: true,
    pages: {
        signIn: '/login',
    },
```

We set debug to true while in development to view additional log messages.

Sets a custom sign-in page to override the unbranded default page; other pages that can be set are signOut, error, verifyRequest, and newUser.

```
    callbacks: {
        authorized({ auth, request: { nextUrl } }) {
            const isLoggedIn = !!auth?.user;
            const isOnDashboard =
               nextUrl.pathname.startsWith('/uploads');
            if (isOnDashboard) {
                return isLoggedIn;
            } else if (isLoggedIn) {
                return Response.redirect(new URL('/login', nextUrl));
            }
            return true;
        },
        async redirect({ url, baseUrl }) {
            const queryParams = new URL(url).searchParams;
            const callbackUrl = queryParams.get('callbackUrl');
            if (callbackUrl) {
                return callbackUrl;
            } else {
                return url.startsWith(baseUrl) ? url : baseUrl;
            }
        }
    },
    providers: [],
} satisfies NextAuthConfig;
```

Checks whether the path starts with /uploads

Redirects to the desired page if available

Providers are services that can be used to sign in a user.

To power our login page, we use the credential provider in which we hardcode a couple of different users. The following listing shows how we parse the given credentials and compare them against our known values. Note that hardcoding the values for our username/passwords is never a good idea, but this example only shows the concept of adding authentication to the project.

Listing 7.40 `Auth` credentials configuration

```
import NextAuth from 'next-auth';
import Credentials from 'next-auth/providers/credentials';
import { z } from 'zod';
import { authConfig } from './auth.config';

const validUsers = [
    { email: 'admin@futuristicfintech.com', password: 'password'},
    { email: 'user@futuristicfintech.com', password: 'password'}
    ]

export const { auth, signIn, signOut } = NextAuth({
    ...authConfig,
    providers: [
        Credentials({
            async authorize(credentials) {
                const parsedCredentials = z.object({
                    email: z.string(),
                    password: z.string(),
                }).safeParse(credentials);
```

Defines an array of emails and passwords to use

An existing configuration is included.

Parses the email and password

```
                if (!parsedCredentials.success) {
                    return null;
                }

                const { email, password } = parsedCredentials.data;

                const user = validUsers.find(user => user.email
➥ === email && user.password === password);

                if (!user) {
                    return null;
                }

                return { email: user.email };
            },
        }),
    ],
});
```

Checks for a successful parse

Retrieves the parsed data

Finds a valid user

If there is no valid user, returns null

Returns the email for the user

The login page will be displayed with the dashboard. We also asked ChatGPT to create a logo for our company. Interestingly, we made several attempts, and while the graphics always looked pretty good, ChatGPT could never quite get the spelling right (figure 7.10)!

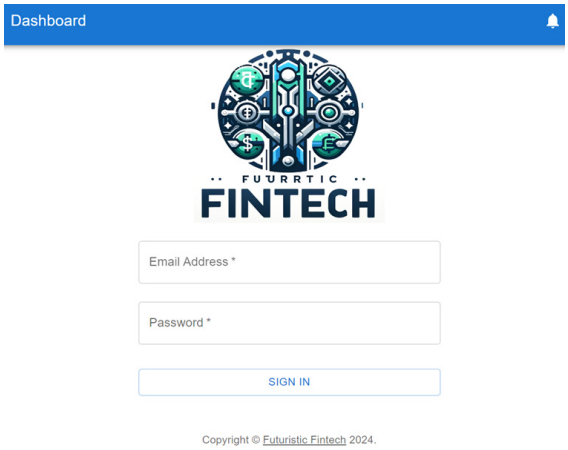


Figure 7.10 Login sample page

The login page is a standard form, and much of the code is in support of the layout, tooltips, and button clicks.

Listing 7.41 A login form showing NextAuth.js incorporated into our login page

```
...
    const [errorMessage, dispatch] = useFormState(authenticate, undefined);
    const [showErrorDialog, setShowErrorDialog] = useState<boolean>(false);
```

```
    const route = useRouter();
…
    const handleSubmit = async (event:
➥ React.FormEvent<HTMLFormElement>) => {

        event.preventDefault();
        const formData = new FormData(event.currentTarget);

        dispatch(formData);
    };

    useLayoutEffect(() => {
        if (errorMessage && errorMessage !== 'success') {
            setShowErrorDialog(true);
        } else if (errorMessage === 'success') {
            route.push('/uploads');
            setShowErrorDialog(false);
        }
    }, [errorMessage, route]);

    return (
…
                    {showErrorDialog && <AlertMessage
                        open={showErrorDialog}
                        setOpen={() => setShowErrorDialog(false)}
                        title={"Error Signing In"}
                        message={"Either your email or
➥ password is incorrect. Please try again."}
                    />}
…
    <Box
                    component={"img"}
…
                    alt="Company Logo"
                    src="/images/logo-light.png"
  …
                    <Box component="form" onSubmit={handleSubmit}
➥ noValidate sx={{ mt: 1 }}>
```

These changes prevent access to the upload page. However, we have not done anything to prevent access to the actual API behind the UI that powers the processing. For that, we need to incorporate access tokens and ensure these are being passed and validated on the server side as well. When we are ready to take that step to ensure the APIs are secure, our UI would have to make a `POST` request and include an `Authorization` header in it.

Listing 7.42 Including the `Authorization` header in a `POST` request

```
        axios.post(`${apiUrl}/files`, formData, {
            headers: {
                'Content-Type': 'multipart/form-data',
```

```
                'Authorization': `Bearer
token_string_that_is_assigned_after_login`,
            },
        })
```

Checking would be done similarly on the server with FastAPI.

Listing 7.43 Creating a function to validate the token

```
async def verify_token(Authorization: str = Header(...)):
    if Authorization != "Bearer secret_token":
        raise HTTPException(status_code=400, detail="Invalid token")
    return Authorization
```

Listing 7.44 Adding the dependency

```
@router.post("", status_code=status.HTTP_201_CREATED,
➥ tags=["ACH Files"], dependencies=[Depends(verify_token)])
```

With all of that in place, when navigating to http://localhost:8000/docs, you will see a new Authorize button that will allow you to specify a token to be passed (figure 7.11). This approach provides a simple way to test our application via the OpenAPI documentation.

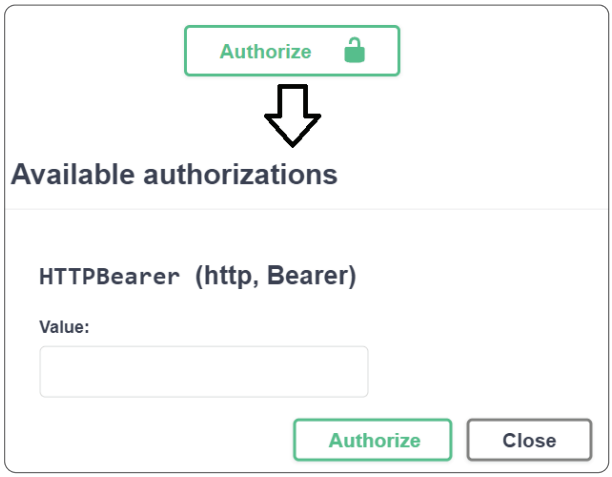


Figure 7.11 Showing the Authorize button and popup

Here, we have simply used a hardcoded value, but it is possible to use a more robust authentication flow. To learn more about authentication flows such as OpenID Connect, check out their site at https://openid.net/.

7.8 Adding TLS to our dashboard

The previous section covered how to secure our dashboard's UI and APIs. However, there is another layer of security that we want you to be aware of—the communication or network layer. As we have seen throughout the book, our URLs have been using HTTP and not HTTPS. While this is not a problem for a development environment, we explore some of the steps involved in securing the network layer. Depending on the size of the company we work for (and of course our role), we may never need to worry about this. However, understanding some of the processes in setting this up can be helpful. It can be especially beneficial when having to work through any problems we may encounter as we gain valuable troubleshooting knowledge.

We will keep this relatively simple by just adding an `nginx` instance to our Docker configuration that we can use. We will not worry about removing access to the existing HTTP ports from the outside as that is not necessary for our development environment. We simply want to see how to use Docker to understand how this process works.

The following listing shows how to add a simple service to our existing docker-compose.yml file. We define the build information and ports. We expose both the standard HTTPS `port 443` and HTTP `port 80` because we will be redirecting any HTTP traffic to HTTPS. If we do not expose the port, the users will receive an error when trying to connect to the web site.

Listing 7.45 The `nginx` endpoint for Docker to support TLS

```
nginx:
  build:
    context: ./nginx
    dockerfile: Dockerfile
  ports:
    - 443:443
    - 80:80
```

A Dockerfile to build from

We expose both the HTTPS 443 and HTTP 80 ports for this container.

Next, as shown in the following listing, we specify the Dockerfile that needs to be in place. We are using the latest image from Nginx, ensuring packages are up to date and then installing OpenSSL because we will be using a self-signed certificate. Another option is using certbot (https://certbot.eff.org/), but a self-signed certificate will be enough for our needs. In a production environment, we would have to obtain the certificate from a trusted certificate authority (CA). We use the `/etc/ssl` directories to store our certificates, but that may differ depending on the OS.

Listing 7.46 Dockerfile to build the container

```
FROM nginx:latest

WORKDIR /etc/nginx

RUN apt-get update && \
    apt-get install -y openssl
```

Uses the latest Nginx

Sets the work directory to /etc/nginx (not strictly necessary)

Updates and then installs OpenSSL

```
RUN mkdir -p /etc/ssl/certs
RUN mkdir -p /etc/ssl/private
RUN chmod 700 /etc/ssl/private
```

Preps directories to store certificates and keys in

```
RUN openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -keyout /etc/ssl/private/localhost.key -out
➥ /etc/ssl/certs/localhost.crt \
    -subj ➥
"/C=US/ST=YourState/L=YourCity/
➥O=YourOrganization/OU=YourUnit/CN=localhost"
```

OpenSSL command creating our certificate; we make use of the directories we just created to store certificates.

```
COPY ./dashboard.conf /etc/nginx/conf.d/default.conf

CMD ["nginx", "-g", "daemon off;"]
```

Copies our server configuration over to the container

The Dockerfile copies a dashboard.conf to default.conf. Let's look at what that configuration should look like.

Listing 7.47 The dashboard.conf

Defines the port we are listening on; 443 is a standard SSL port, but we still must include the ssl parameter to enable SSL.

The name of the server; we are mapping localhost and 127.0.0.1.

```
server {
    listen 443 ssl;
    server_name localhost 127.0.0.1;

    ssl_certificate /etc/ssl/certs/localhost.crt;
    ssl_certificate_key /etc/ssl/private/localhost.key;

    location / {
        proxy_pass http://dashboard:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
➥ $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Specifies the cert and key for SSL

Any traffic coming to this server is redirected to the dashboard on port 3000. We also set additional headers to provide information to the receiving server, which helps indicate where this forwarded request came from.

Sets up port 80

Again, we specify the localhost.

```
server {
    listen 80;
    server_name localhost 127.0.0.1;

    location / {
        return 301 https://localhost:443$request_uri;
    }
}
```

We redirect the browser to the SSL port.

We have now secured the dashboard using TLS. While doing so is required for production environments, it is not strictly necessary for our development. However, having a bit of knowledge about some of the basic setup can help you troubleshoot problems

when environments are migrated or otherwise changed, and your application breaks. For instance, in the previous listing, headers such as `X-Real-IP` were set. If the application depends on these to be present but someone removes them from the configuration, we could end up with a troublesome bug. While uncommon, running into similar bugs is possible.

7.9 Testing with Playwright

You are now familiar with a few approaches to testing our software, specifically TDD and BDD. We have also looked at building tests with `pytest`, when working in Python, and Jest, when using TypeScript. With those tools, we have focused mostly on smaller unit testing. Because our MVP is completed, we are going to use end-to-end testing now to validate our work before it gets shipped off. For that, we use a tool called Playwright. Playwright is a cross-browser, cross-platform, and cross-language testing tool. So, regardless of our environment, chances are we can use Playwright. We chose Playwright with Python because we are currently using this combination for our APIs and file parsing.

Getting started with Playwright is simply a matter of installing `pytest-playwright` with `pip install pytest-playwright` and then running `playwright install` to ensure browser extensions are installed as well. Once that is complete, we can start scripting. A powerful feature of Playwright is using the interactive mode from a Python console. After launching a browser with `headless=False`, we can see the results of our commands in real-time.

Listing 7.48 Simple interactive Playwright

```
from playwright.sync_api import sync_playwright
playwright = sync_playwright().start()
browser = playwright.chromium.launch(headless=False)
page = browser.new_page()
```

Specifies headless=False, which allows us to view and interact with the browser

We will not see the browser window until we call the new_page() method.

When we have a handle on using Playwright to navigate to URLs and select elements, we can begin writing tests. The following tests show that we rely heavily on the `get_by_role` method when selecting elements. The `get_by_role` method follows the W3C specifications for ARIA (Accessible Rich Internet Applications), which can aid in developing pages that help conform to standards used by assistive technology (e.g., screen readers).

Listing 7.49 Test to populate the login form

```
def test_login_screen(page: Page):
    page.goto("http://localhost:3000/uploads")
    email = page.get_by_role("textbox", name="Email")
    expect(email).to_have_id("email")
```

Uses get_by_role to help ensure ARIA is supported

```
email.fill("admin@futuristicfintech.com")
password = page.get_by_role("textbox", name="Password")
expect(password).to_have_id("password")
password.fill("password")
page.screenshot(path="screenshots/login_screen.png")
```

Uses the fill method to populate text

Creating screenshots is easy.

As we saw, populating fields is straightforward, and clicking buttons is just as easy. Let's see whether we can cause a login exception by not populating any fields and clicking the button to sign us in. Notice how we use the `wait_for_selector` to ensure the alert is visible. If we do not use this, our error message may be translucent as it may not have finished rendering correctly.

Listing 7.50 Capturing the exception message

```
def test_login_exception_message(page: Page):
    page.goto("http://localhost:3000/uploads")
    page.expect_navigation(wait_until="load")
    expect(page).to_have_url(re.compile
➥("http://localhost:3000/login.*"))
    signin_button = page.get_by_role("button", name="Sign In")
    expect(signin_button).to_have_id("signin")
    signin_button.click()
    alert = page.get_by_role("heading")
    expect(alert).to_have_id("alert-message")
    expect(alert).to_have_text("Error Signing In")
    page.wait_for_selector("#alert-message",
➥ state="visible")
    page.screenshot(path=
➥"screenshots/login_exception.png")
```

Verifies that we were redirected to the login page when we initially go to /uploads

Clicking buttons is simple.

Gets the alert message that should appear and verifies the text

Uses the wait_for_selector before taking a screenshot

If `wait_for_selector` is not used, we may end up with an error message that is not completely rendered, as shown in figure 7.12 on the left. This can be a common challenge in automated UI testing—the speed at which tests execute can lead to this scenario. Often, as we step through the code to verify the test is working, we won't encounter this situation. It is generally possible only when the test is run as part of our automation or build pipeline.

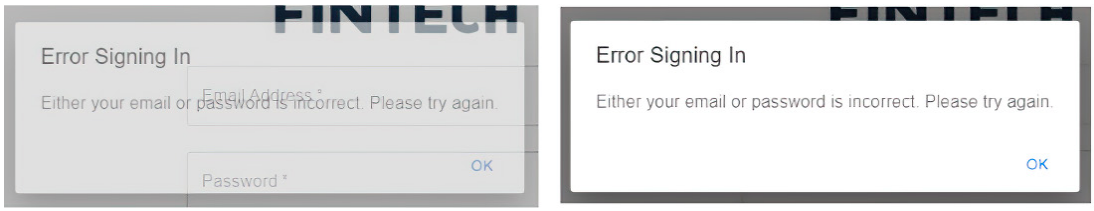


Figure 7.12 Screen not rendered completely when we do not wait

Here, we have only touched on some basics of using Playwright to perform end-to-end testing on our dashboard by automating a browser instance. You should be able to see the benefits of running these types of tests on a variety of browsers. Our listings showed capturing a screenshot as part of the test, but that was mostly for validation purposes since we are running in a headless (no browser window) mode, and we do not actually see it running. In practice, we may only capture a screenshot of an exception coding for further troubleshooting and research. Maybe we could even combine Playwright with our BDD-style syntax to create browser automation tools for our QA department to use! Combining these into a CI/CD pipeline along with our current unit tests can further ensure our software is ready to ship.

7.10 MVP wrap-up

At this point, the MVP of our dashboard is complete. We have built the necessary remaining APIs using manual, unit, and integration testing to make sure the dashboard will work flawlessly once delivered. We also worked on improving the security of our application by using TLS and user authentication.

Certainly, there are tweaks we could make to the UI and additional functionality. Make notes of those items as they are great to work on when you are capable of accepting more story points during a future sprint. As mentioned before, we must balance the functionality of the app with a delivery timeframe. Thus, it is important not to get caught up in adding all these bells and whistles.

We covered a lot of ground in this chapter. Hopefully, our previous research spikes provided us with enough background to help us stay on target. The next section addresses some of the problems we may run into when putting the MVP together.

7.10.1 Troubleshooting the MVP

Inevitably, we run into problems as we work through our projects. While these can be frustrating, being able to resolve them is a necessity not limited to software development.

One of the most common problems that we run into is a CORS error that usually says something like `Access to XMLHttpRequest at some_location from origin some_server_name has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource`. It can be triggered by a number of problems, such as

- Request went to an unexpected location. In one instance, our Docker container had not shut down as intended and was still attempting to handle requests without our CORS changes in place.
- Pydantic was throwing an error because we had specified a required field named `account_number_last_4` but had mislabeled it as `account_number` in the SQL query. This was not too difficult to track down because Pydantic provided us with a nice error that looked similar to `{'type': 'missing', 'loc': ('response', 0, 'account_number_last_4'), 'msg': 'Field required', 'input': {'id': UUID`

`('29159e96-c2b3-452e-be1b-b4258c5e77aa'), 'transaction_code': '22', 'amount':'0000000201','account_number':'*************0034','addenda_count': 1}, 'url': 'https://errors.pydantic.dev/2.6/v/missing'}`. Pydantic has specified the type as missing and provided our object with both fields and values. Sure enough, we do not see the missing field but the `account_number` instead.

Other errors that we ran into were also Pydantic related because we failed to update the field definitions as we were going through the code. So, a message such as `1 validation error for AchEntryPpdDetailsSchema` also showed `[type=string_type, input_value=Decimal('1.00'), input_type=Decimal]`, which was a simple fix to ensure that all our data types were matching.

Another data-type-related problem was the mix-up between strings and numerics because we were inconsistent or forgetting to update the parsing program to handle the cents when converting from a string to a numeric. So, $1.00 was showing incorrectly as $100.00. Interestingly, when reworking the code to go from strings to numerics, Copilot was not very helpful. It kept assuming that we wanted the next two positions in the string instead of breaking the current field up.

We also ran into problems with SQL when we specified `INNER JOIN` instead of `LEFT JOIN`. We wanted to capture the addenda counts for a particular entry. `INNER JOIN` worked great when there were addenda records, but without them, the row was dropped, and that was not what we wanted.

When testing with Playwright, we needed to ensure that names/IDs were included in our elements. Playwright also has great error messages. When running in strict mode, and there are multiple matches, it enumerates the possibilities and provides appropriate examples such as `get_by_role("banner").get_by_role("button")`, `locator("button").nth(2)`, and `get_by_role("button", name="Dashboard")`.

Our final problem came from Docker, where we accidentally deleted a Dockerfile when updating our code. This resulted in the message `failed to solve: failed to read dockerfile: open /var/lib/docker/tmp/buildkit-mount39300809/Dockerfile: no such file or directory`. In another case, when updating the code for Docker, we simply copy/pasted existing code without clearing out the existing files in Docker, which triggered a problem in Next.js because we had different parameter names (slugs) for the same dynamic path. The error was clear enough: `Error: You cannot use different slug names for the same dynamic path ('fileId' !== 'id')`. However, it was also confusing because it was not a code problem but rather a directory structure problem (and procedural as well). Next, we had several dependency problems due to mismanagement on our part. These only appeared when we tried to build the solution in our Docker container. We ended up with errors such as `Starlette was listed in the requirements.txt`. The conflict was caused by

```
The user requested starlette==0.37.1
fastapi 0.110.0 depends on starlette<0.37.0 and >=0.36.3
```

Finally, because we did not expose `port 443` in `docker-compose`, the connection was refused when trying to access the HTTPS endpoint. That error looked similar to `Error: connect ECONNREFUSED 127.0.0.1:443`.

As we've experienced, being a full stack developer means we also get to see errors from many different areas, which makes our lives exciting!

Summary

- Previous sprints explored using generative AI and technologies such as Python/FastAPI, Postgres, Docker, and Next.js to expedite software development for our ACH dashboard MVP.
- The MVP allows users to upload and parse ACH files, store them in a database, and visualize the results on a dashboard.
- Different minimum concepts include MVP, MLP, MMP, and MMF, each serving unique purposes—from testing assumptions to delivering features.
- Preparing for an MVP involves integrating components such as the ACH parser, APIs, database design, and UI to function cohesively.
- Developing the `/files` APIs ensures alignment between the backend and UI, focusing on data accuracy and seamless functionality.
- BDD and TDD approaches help integrate APIs with the UI, ensuring accurate data responses and interactions.
- Refactoring interfaces and schemas improves code maintainability, readability, and flexibility when consuming APIs.
- Supporting the API integration of ACH batches into the dashboard requires updating data structures and transforming responses to match UI needs.
- The upload functionality from the dashboard simplifies user interaction but highlights development considerations, such as code duplication and user feedback upon file upload.
- Deployment in Docker involves cleaning up configurations for production readiness, ensuring seamless communication between containers, and addressing common troubleshooting problems.

Part 3

Beyond the minimum viable product

With the minimum viable product (MVP) in the hands of the stakeholders, you will start receiving feedback and consider further development possibilities, expanding the functionality of the MVP in exciting new directions .

Chapter 8 enhances the ACH processing by expanding the exception handling of the system. This vital aspect of software is often not glamorous, but still crucial for robust and user-friendly software.

Chapter 9 expands the ACH dashboard by providing the ability to search and create an audit trail to track user interactions with the APIs.

Chapter 10 adds company information to the dashboard. This activity will let you explore several additional areas, opening new directions for enhancing the dashboard.

Chapter 11 dives into international ACH transactions and the regulatory effects they have on our system.

Chapter 12 wraps everything up by providing ideas on how to take this project even further.

After reading the book, you will be familiar with a wide variety of technologies, tools, and techniques. We hope you have found some interesting areas to dive further into. Moreover, you will have a pretty robust ACH dashboard and a better understanding of ACH processing!

8 Exceptions

This chapter covers

- Initial feedback from the MVP
- Planning and addressing user requests
- Identifying different types of exceptions
- Updating the dashboard
- Ensuring our UI is responsive

In chapter 7, we successfully launched our minimum viable product (MVP) to our user base to help solicit feedback and drive enhancements to the product. The feedback varied significantly—from users who loved the new direction of the dashboard to those who expressed their dissatisfaction, feeling it was missing too much functionality although it was not meant to be a full replacement yet. The business now wants us to both capitalize on what went well and address some of the missing functionality the users are currently clamoring for.

By far, the biggest need recognized in our MVP was having a wider variety of ACH-related exceptions and making their handling possible within the user interface. This chapter identifies different categories of exceptions and ways to identify and potentially recover from them.

8.1 Planning enhancements

The benefit of providing our target audience with an MVP is that we can get feedback from them and then use it to determine which features need to be added to the software. This advantage allows us to concentrate on items that deliver the most value. In this instance, we want to focus on providing a way to view ACH exceptions in the UI, as well as to search through our loaded files, which will help us find a much-needed and practical enhancement to the application as users must be aware of exceptions that prevent further processing of ACH files. This is especially valuable because we focus on Prearranged Payment and Deposit (PPD) ACH transactions, meaning that payroll and bill payments are at risk of not being processed.

Depending on our circumstances, we may have a normal sprint to complete our work, or if the need is dire enough, we may be expected to turn around changes and fixes sooner. We will also likely see user stories written for these items, such as

- As an operator, I want to be able to view processing exceptions and be aware of potential ACH problems that need to be resolved so that transactions are not delayed in posting to a customer's account.
- As an operator, I want to be able to search for specific transactions so I can research items for customer inquiries.

In either case, providing stakeholders with a timeline or Gantt chart for the project can be beneficial. These visualizations are especially helpful if we find ourselves with a hard deadline. We often work backward from that deadline, trying to complete the expected tasks, while fitting them all within the allotted time. Of course, the required timeframe may not be possible, and having it laid out for management is a great way to help make our case for more time should we need it. Again, we rely on PlantUML to provide us with a simple Gantt chart we can share with the stakeholders. The following listing shows the required syntax.

Listing 8.1 PlantUML syntax for a Gantt chart

```
@startgantt
saturday are closed
sunday are closed

header ACH Dashboard Enhancements - Round 1

Project starts 2024-08-01
-- Exceptions Enhancement --
[Database Design] starts 2024-08-01 and lasts 2 days
[Exception API Design] starts at [Database Design]'s
➥ end and lasts 2 days
[ACH Processor] starts at [Exception API Design]'s
➥ end and lasts 3 days
[UI Exception Changes] starts at [ACH Processor]'s
➥ end and lasts 3 days

-- UI Enhancement –
```

Excludes weekends from our chart because we do not want to work overtime

A header for the chart

Creates a task with a start time and duration

Creates a visual break in the chart

Tasks can be dependent on other tasks.

```
[Update fields screen] starts at 2024-08-01 and lasts 3 days
[Update batch screen] starts at [Update fields screen]'s
➥ end and lasts 3 days
[Sprint Complete] happens at [UI Exception Changes]'s end
@endgantt
```

This listing produces the chart shown in figure 8.1. As we mentioned, such charts are helpful for visualizing the work and are often appreciated by project stakeholders.

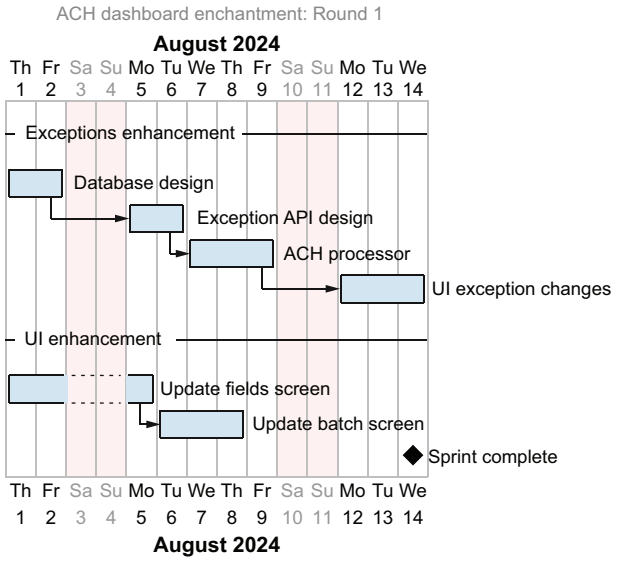


Figure 8.1 A Gantt Chart for our enhancements

Now that we have outlined the initial work for this sprint, we can start tackling the steps needed to add exceptions to our application.

8.2 Exceptions and warnings

Remember that we prefer working here in short development cycles, as shown in figure 8.2. In step 1, we need to create an ACH file with our desired exception condition, which means we will be updating the `ach_file_creation.feature` and associated test_create_ach_files.py to support creating files with our desired exception. In step 2, we support the new exception by updating the ach_file_processor.py, either by diving right in and starting to work on it or by creating unit tests in a more test-driven development (TDD) fashion. Finally, in step 3, we update the actual UI for the dashboard to support viewing and displaying of the exceptions.

Although the steps are going to be covered in separate sections, note that we would likely be working on each individual exception through these general steps. And as we work through each exception, we will notice improvements/refinements that can be made.

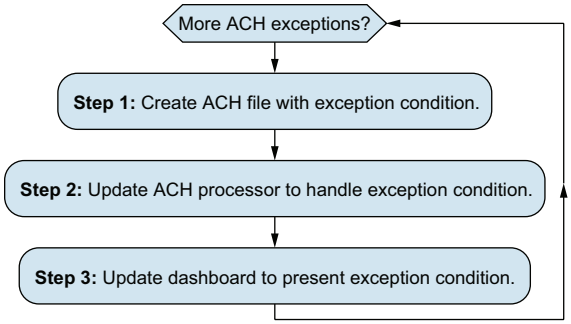


Figure 8.2 Flowchart representing our short development cycle

First, we need to define exactly what we mean when we use the terms "exception," "error," and "warning." They may seem somewhat interchangeable, but for our purposes, we would like to provide some background information on what we have in mind when talking about each. Please bear in mind that these are our own definitions and not a reflection of any fintech or Nacha standards.

- *Error*—A condition that prevents processing from taking place. We can think of errors as conditions such as an incomplete file upload or a bug in our software that causes processing to halt. Some conditions would cause an error, such as having a record with a record type code that does not exist in the ACH standard (e.g., record type code 2) because we do not have a place for this record in the database.
- *Exception*—A business condition that would prevent file processing, which could include an improperly formatted file, invalid data in the file, or conditions the business has defined that should stop processing. Examples of these business conditions may be transactions over certain amounts, flagged customers, or other transaction activity that may cause concern.
- *Warning*—A business condition that requires notification but does not necessarily prevent the file from being processed. Warnings may allow us to show leniency in enforcing ACH rules for files or customers. For instance, if the count on a control record is incorrect, the business may allow this discrepancy and process the file anyway.

As we can see, there are slight differences in the conditions and responses for these different categories. We should also have the recovery or research steps documented clearly in case an error is encountered. We have been involved in supporting software in the middle of the night where an explicit check was done within the code but without any indication of why the condition may occur, how to resolve it, or whether it is important. More than once, it was not important enough to stop the processing, and the code would be commented out so the check could be bypassed. This story illustrates the need to determine the severity of exceptions, and when and where they should be reported. Figure 8.3 shows a flowchart that can help you when deciding whether an exception needs to halt processing or it can be handled differently.

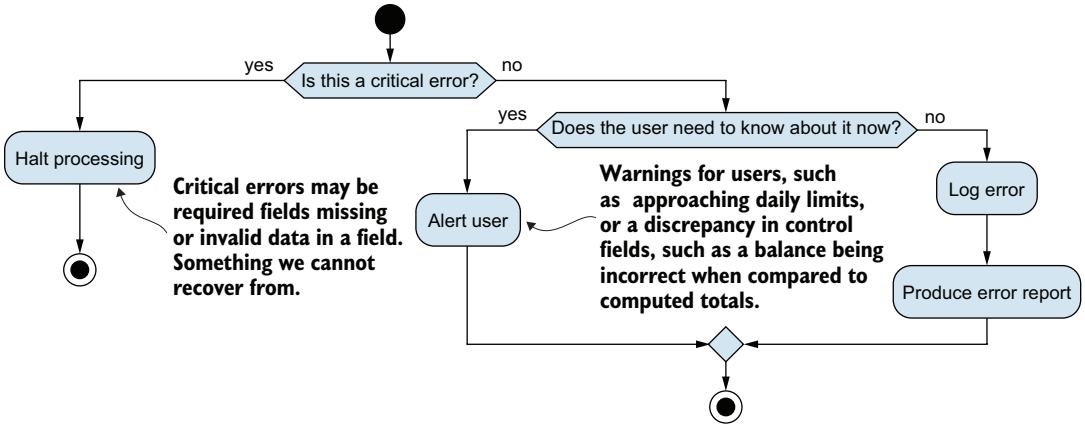


Figure 8.3 Flow chart for deciding how to handle errors

To summarize, we should be thinking about

- Whether this is a problem that requires halting processing
- Whether it can be reported to the user in a timely manner
- Whether this is something that can wait but does need to be resolved eventually

Too often, we may create alerts, messages, and reports that are ignored and dismissed by users before corrective action is taken. It can be a balancing act as different products or business needs require different approaches.

From our software and development perspective, we must consider these different types of errors and warnings. It is also important to understand that the Federal Reserve has their own way of looking at a file's ACH status. These statuses can be `accept`, `pend`, or `reject` and are exactly what you might expect:

- `accept`—The file is good and has been accepted by the Federal Reserve.
- `pend`—The financial institution must confirm that the file should be accepted or rejected. This usually happens when there is some potential exception that may occur due to the formatting or contents of the file.
- `reject`—A condition/error causes the file to be rejected. This is an error state we may also end up with when the financial institution chooses to reject a file that was previously in a `pend` state or when the Federal Reserve flat-out rejects a file.

It should be noted that when files are corrected, there should be an audit trail. If a file is modified, the originator of the file may need to be notified. Otherwise, these changes may lead to confusion when attempting to research problems. We can imagine trying to speak with someone about a file where the file ID was corrected from a numeric to a letter without any indication it was done. It could lead to some frustrating interactions, with someone at the financial institution claiming not to see a file although the user clearly sees the transaction hitting their account.

8.3 Creating ACH files with exceptions

As part of the MVP in chapter 7, we created the `ach_file_creation.feature` that allowed us to easily create well-formatted files using the BDD-style syntax (known as Gherkin). Recall that it allowed us to create files as shown:

```
  Scenario: Create an ACH file with a single batch and a single debit
    Given I want to create an ACH file named "ppd-single-debit.ach"
    And I want to have an immediate destination of "123456789"
    And I want to have an immediate origin of "987654321"
    And I want to have 1 batch with ACH debits only
➥ and a standard entry class code of "PPD"
    And I want 1 entries per batch with random amounts between 100 and 100
    And I want to use individual names of "John Doe"
    And I want to have company name "My Company"
➥ and company id "1234567890"
    When my ACH is created
    Then I should have a file named "ppd-single-debit.ach"
    And there should be 1 batch in the file
    And there should be 1 entries in the file
```

This works well for that happy path that we often start with, but now we need to expand our code to address some possible exceptions so we can work on building the required functionality. In some cases, our syntax may help enforce proper formatting of the ACH file, and that's obviously a good thing. We want the creation process to be easier than creating something by hand, but now we also want to add some flexibility to the files. Eventually, we may accept CSV or other formatted files that would make customizing the fields easy. For now, we can look to expand and hopefully improve our syntax.

8.3.1 Creating ACH files with file-level exceptions

File-level errors and warnings will have to do with either the file header (type 1) or the file trailer (type 9) records. Many of the exceptions we might encounter stem from formatting problems with the file, and we would like to be able to introduce the following exceptions:

- *File ID is not an uppercase letter in the range from A to Z or a number.* If it is a lowercase letter, the file could be rejected, or the financial institution could decide to fix it since it is a minor problem and easily corrected.
- *The immediate destination should be our bank's routing number.* This is important because we obviously want to ensure we are processing the correct file for the correct bank. If we were to receive a file that was intended for another institution, it could be categorized as information leakage, unauthorized disclosure, or inadvertent data sharing. Regardless of what we call it, there are serious implications in terms of customer privacy, and we may have legal and regulatory obligations. While a seemingly innocuous problem (e.g., the field didn't match), it would be an all-hands-on-deck type event. In FinTech, there are all kinds of places where this may occur, especially when dealing with multitenant environments. For

example, it could occur when storing/retrieving images, displaying transactions in a mobile or internet banking application, sending statements, or communicating to incorrect or unauthorized addresses. This change also requires us to maintain a list of routing numbers associated with the bank since we need to be able to validate them. Because of bank mergers or acquisitions, it may be possible for banks to have more than one routing number.

- *Blocking factor should never change but would be an error if it did.* This gives us a chance to update our syntax to support setting fields and using the `get(field, default value)` so we do not need to have a lot of different syntax, which could then be expanded to support `padding-left/right`, `fill value`, and `length`.
- *Record size is similar to the blocking factor, and we always expect a 094.* In theory, this field could be used to change the record format length, but in practice, systems typically just assume the 94 bytes and do not use the record size to read different-sized records. This does allow us to expand our checks to become more generic, exchanging constraints imposed by specific contexts and allowing more flexibility, but with that greater power comes more responsibility in knowing how to use it. For instance, we are adjusting the field offset for what is needed by Python. We could have left it starting at 1, so it aligned better with the ACH documentation and changed offsets within the code.
- *File totals.* The file trailer record should match the computed totals for the file
- *File hashing.* Each batch control has a hash value that is added up, truncating the result as necessary to keep the 10 lower-order digits (rightmost digits).

8.3.2 Creating ACH files with batch-level exceptions

As we parse more of the file, we need to consider batch-level exceptions. These are specific to the batch records, and while they share some similarities to other exceptions, we may find they do not necessarily cause the entire file to be rejected. Instead, the batch itself could potentially be rejected by the system. In turn, that could cause the entire file to be rejected, or the financial institution could simply require the originator to send a new file. Some examples of batch-level exceptions are the following:

- An invalid effective date would cause a batch error as this is the necessary information for correctly posting the file. So, whether it is missing or formatted incorrectly, there is no way to recover it, and a new file is needed.
- Invalid batch totals on the control record would cause a warning. In practice, this usually causes a file to be rejected, but it would also be possible to correct the given file so that the totals match the entries for the file to be accepted.
- Bad Standard Entry Codes (SEC) on a batch will cause an error and the batch to be rejected. There is a set list of viable SEC codes that can be used, and some of them change the parsing of the records. We have been focused on the SEC code of PPD (used in payrolls, bill payments, pensions, etc.) and will continue to do so in this chapter.

8.3.3 Creating ACH files with entry-level exceptions

Entry-level exceptions are interesting because aside from the usual formatting exception, they can be related more to the account in our system. Covering these exceptions requires more functionality in the system, but to give you a better idea of what we mean, let's cover some of the common entry exceptions:

- *Invalid account number*—It could be related to formatting or simply not found on the system.
- *Invalid amount*—The amounts are formatted (e.g., such as including the decimal or containing spaces).
- *Insufficient funds (NSF)*—An account does not have funding for the transaction. This opens a lot of interesting possibilities for our project, from maintaining customer and account tables to assessing fees for transactions that cannot be processed.

8.3.4 Clean-up of our file creation syntax

With the need to create additional files, we will be revisiting the BDD-style syntax we used to create our files. Often, this is a great place to gain some experience in refactoring code and finding little improvements to make the syntax or code more robust. Because these are normally internal facing, we can gain experience by reworking existing code and seeing the potential effects it may have.

For instance, we had the line `@then(parsers.re('I should have a file named "(?P<expected_file_name>.*)"'))`, but there is no need to specify the filename again. Not only does it cause potential problems if we copy/paste a scenario to create a similar one, but it is just unnecessary typing. Instead, since we already have the filename stored, we can update it to become `@then("I should have a file of the same name")`. Then the code can pull in the stored filename from `setup_info['filename']` to perform the check.

Another improvement we can look at is cleaning up the way we parsed the strings from some of our statements. We originally used regular expressions to capture the strings being used, and while that is not necessarily a problem, it does add a layer of potential confusion. We are reminded about the famous quote from Jamie Zawinski: "Some people when confronted with a problem think, 'I know, I'll use regular expressions.' Now they have two problems." To fix this, we can replace our regular expressions such as `@given(parsers.re(r'I want to create an ACH file named "(?P<filename>.*)"'))` with a far simpler `@given(parsers.parse('I want to create an ACH file named \"{filename}\"'))`. Keeping our code simple ensures it's more maintainable and less error-prone to updates when we return to it in a year or two to add more improvements.

What other improvements can be made? Remember that we should look to balance functionality and need. We can probably add a lot of bells and whistles, but if they go unused, was that really worth the extra work? That balance is especially important when

talking about internal tools that do not necessarily add value that clients can see. Our stakeholders are unlikely to be impressed if we have a full-function ACH creation utility but nothing for them to actually process those files with!

8.4 Expanding the ACH processing for exceptions

When adding the exceptions to the parser, we'll follow a TDD approach by creating a test that will validate the intended exception has been added to the database. With the TDD approach, we expect this test to fail, and then we'll work on adding the required logic to have a passing test. This approach may not be for everyone, but we would encourage you to try it out, if nothing else, just for a different perspective on development.

8.4.1 ACH exceptions in the database

Originally, we stored some exception text in the database. However, we may want some finer controls and less redundant data. Therefore, we create a database type called `ach_exception_severity` to limit the entries that can be used when creating exceptions. The use of an `ENUM` allows us to ensure consistency in the type and reduce redundancy in our database. The downside is that `ENUM`s are not part of the SQL standard, and therefore, they may not be implemented in your relational database management system (RDBMS). While these examples assume the use of a Postgres database (which supports `ENUM`s), it is also possible to mimic this functionality with a separate table that provides the values we would use for the `ENUM` and a foreign key relationship in that table.

The following listing shows the creation of the `ENUM` type and our new `ach_exception_codes` table we use to store extended descriptions of the errors.

Listing 8.2 Creating the database `ENUM` and exception code table

```
CREATE TYPE ach_exception_severity AS ENUM
➥ ('error', 'warning', 'info');

CREATE TABLE ach_exception_codes (
    ach_exception_codes_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    exception_code VARCHAR(3) NOT NULL UNIQUE,
    exception_severity ACH_EXCEPTION_SEVERITY NOT NULL,
    exception_description VARCHAR(255) NOT NULL
);
```

Creates the ach_exception_severity type

Uses our type as the datatype for a field

Next, we create an exceptions table to keep track of our exceptions. There are a few approaches to this, where we have multiple tables for each specific entry (similar to having multiple ACH record types) and then tie them together with a view. However, here, we demonstrate using a single table that can hold references to other tables which contain the record we could look up the exception with. The following listing contains the `CREATE TABLE` statement. The important difference from our previous tables is that we use foreign key constraints, which allow `NULL` for the `ach_records_type_5_id` and

ach_records_type_6_id fields because this information will only be populated when we have a record of that type with an exception. Note that every exception will be associated with a file, so we expect ach_files_id to be populated every time; therefore, we still have a NOT NULL constraint.

Listing 8.3 Creating the exceptions table

```
CREATE TABLE ach_exceptions (
    ach_exceptions_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    ach_files_id UUID NOT NULL
       REFERENCES ach_files(ach_files_id)
       ON DELETE CASCADE ON UPDATE CASCADE,
    ach_records_type_5_id UUID
       REFERENCES ach_records_type_5(ach_records_type_5_id)
       ON DELETE CASCADE ON UPDATE CASCADE,
    ach_records_type_6_id UUID REFERENCES
       ach_records_type_6(ach_records_type_6_id)
       ON DELETE CASCADE ON UPDATE CASCADE,
    record_number NUMERIC NOT NULL,
    exception_code VARCHAR(3) NOT NULL REFERENCES
    ach_exception_codes(exception_code)
                                ON UPDATE CASCADE
);
```

These fields are not always used and can be NULL.

The exception code details are referenced in ach_exception_codes.

Adding data to our table is part of the database startup. We do this because these are standard errors we can consider being shipped with the product. Often, we find these types of tables or settings that are part of the setup for a financial institution.

Listing 8.4 Adding data to the exceptions table

```
INSERT INTO ach_exception_codes (exception_code,
  exception_severity, exception_description)
VALUES ('001', 'error', 'Record length is not 94 characters');
INSERT INTO ach_exception_codes
 (exception_code, exception_severity, exception_description)
 VALUES ('002', 'error', 'Record type was an unexpected value');
…
```

Each exception needs to be inserted into the database.

Adding this type of persistent data is necessary for the dashboard; however, it does add a level of complexity. Our tests become dependent on these tables being defined and the data populated. We also ran into a problem because our unit tests were truncating all tables to ensure the database was clean. The following listing shows the changes to a portion of the truncate script so that we can understand the needed changes to the SqlUtils.truncate_all(). We may want to consider updating the name (or at the very least the documentation) so that it is understood that truncate_all is not actually truncating all data anymore.

Listing 8.5 Updating the `truncate` script

```
EXECUTE (
   SELECT 'TRUNCATE TABLE ' ||
          string_agg(quote_ident(table_name), ', ') ||
          ' CASCADE'
   FROM information_schema.tables
   WHERE table_schema = 'public'
   AND table_type = 'BASE TABLE'
   AND table_name NOT IN ( 'ach_exception_codes',
                           'ach_recovery_options',
                           'bank_routing_numbers')
);
```

We have tables that need to be skipped from our truncate logic as they contain hardcoded values.

Although we may not be overly burdened by this approach now, there will likely come a time when we'll have to manage the test database a bit better. A popular option is to use Testcontainers (https://testcontainers.com/), which allows us to create temporary Docker containers that contain our database and will not persist beyond the lifespan of our test. This provides obvious advantages when dealing with unit testing, especially when it comes to tests run within the CI/CD pipeline.

8.4.2 Recovery options

Before updating the ACH parser to support our new exceptions, let's take a moment to consider the flip side of logging exceptions—what to do when they are encountered! We want to consider exactly how users should recover from a problem. This can be especially true when dealing with loading transactions, which the dashboard could eventually be involved with. Since an ACH file is used to move money around, if our file fails to load or otherwise has error conditions, we may need to consider implications to user's accounts. In other words, it may not be enough to just load a file and try again. We need to be able to ensure transactions will not be posted to the account again, resulting in customers being either credited or debited multiple times. At this point, our concerns do not need to be centered around those details, but it is important to provide some help to the user.

One scenario that comes to mind is encountering an exception being thrown, which involved comparing totals from two parts of the system. If the totals did not match, the processing halted, resulting in a call to a developer at 2 am. The concern was not an exception making us work at odd hours. If a critical processing exception occurs, we can expect occasional problems. Rather, the concern was about the recovery steps that should be taken when the error was encountered—there were none! The result was multiple developers having to get involved to figure out what needed to be done. It was determined the check could be safely bypassed, and we could follow up and investigate further the following day. Eventually, it was found the audit records were not written out as expected, which resulted in the totals being off, but the problem recurred several times before that was discovered.

While we do not have such concerns with our project right now, it would certainly be nice to provide users with some additional information. After all, the dashboard may eventually require its own internal help system, or perhaps a look at the exception will offer some advice on what actions can be taken. The following listing shows the creation of an `ach_recovery_options` table tied back to the `ach_exception_codes` table, allowing multiple entries to be associated with an exception code.

Listing 8.6 ACH recovery options table

```
CREATE TABLE ach_recovery_options (
    ach_recovery_options_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    exception_code VARCHAR(3) NOT NULL REFERENCES
       ach_exception_codes(exception_code) ON UPDATE CASCADE,
    recovery_option VARCHAR NOT NULL
);
```

Each recovery option should reference a specific exception.

The recovery option is simply text that the users can review for help.

We can then prepopulate the table with our recovery options as part of our startup. The following listing shows some possible recovery options being inserted into the database.

Listing 8.7 Prepopulating the recovery options

```
INSERT INTO ach_recovery_options (exception_code, recovery_option)
➥ VALUES ('001', 'Request a corrected file from the originator');
INSERT INTO ach_recovery_options (exception_code, recovery_option)
➥ VALUES ('002', 'Request a corrected file from the originator');
INSERT INTO ach_recovery_options (exception_code, recovery_option)
➥ VALUES ('004', 'Specify a new File ID Modifier');
```

The recovery options could certainly be expanded. However, at this point, we just want to ensure recovery and help is a consideration in our development. We should all be aware of the importance of clean code and comments to help us understand the code when we have to dive back into it in 6 to 12 months. Just as important is making sure that our software is usable by the end-user. Otherwise, we may find ourselves having to provide support for problems that users should be able to fix themselves. Defining help within the application and considering more extensive documentation and troubleshooting in a tool such as Confluence can be helpful to end-users and support staff. In the end, it will free you up to spend more time developing, and that is what we are all here for!

8.5 Updating the ACH parser

With the database capable of supporting better exception handling, we can now work on updating the ACH parsing to write our exceptions. Many exceptions that we might encounter from a formatting perspective can be handled through Pydantic. We can

continue to add constraints and validation to our Pydantic definitions, which will take many of the more tedious tasks out of our hands. Other exceptions may rely on examining more than just the fields of our record. For instance, ensuring that trace numbers are ascending is something that we would prefer to do outside of Pydantic.

8.5.1 Creating unit tests for the exceptions

We have already shown how to update our BDD-style syntax to create files with the needed errors. Then we can create the basic layout, as shown in the following listing. This approach should work just fine for testing, regardless of whether an exception was encountered.

Listing 8.8 Test for invalid immediate destination

```
def test_incorrect_immediate_destination(parser):
    ach_filename = "invalid_immediate_destination.ach"
    dir_path = os.path.dirname(os.path.realpath(__file__))
    file_path = os.path.join(dir_path, "data", "bad_files",
                                           ach_filename)

    expected_exceptions_count: int = 1
    expected_exceptions_code: str =
       AchExceptions.INVALID_IMMEDIATE_DESTINATION.value

    ach_files_id =
       SqlUtils.create_ach_file_record(filename, "123456789")

    parser.parse(ach_files_id, file_path)
    exceptions = SqlUtils.get_exceptions()

    assert (
        len(exceptions) == expected_exceptions_count
    ), f"Expected {expected_exceptions_count},
➥ but got {len(exceptions)}"

    assert (
        exceptions[0] == expected_exceptions_code
    ), f"Expected {expected_exceptions_code},
➥ but got {exceptions[0]}"
```

We use an ENUM value so that the exception codes remain flexible.

Defines initial variables so we can obtain a file to load

Inserts a needed dummy record for our insert

Parses the file which loads it into the database

Gets the exceptions

Asserts that we have our one expected exception

Considering the previous sample, we may notice that much of that code will be the same for each test. This is a perfect opportunity to recall that refactoring is one of the TDD phases. In this instance, we can refactor the code so that the processing of the file is generalized to one routine, and we call that with the expected values, as shown in the following listing. This approach still works well for these basic tests where we

just want to confirm the exception has been caught. Since the `assert` raises an exception, we still see our error thrown from within the `test_incorrect_file_id_modifier` method.

Listing 8.9 Refactored test

```
def test_incorrect_file_id_modifier(parser):
    process_file_and_verify_exception(
       ach_filename = "invalid_file_id.ach",
       expected_exceptions_count = 1,
       expected_exceptions_code =
          AchExceptions.INVALID_FILE_ID_MODIFIER.value,
       parser = parser)

def process_file_and_verify_exception(ach_filename: str,
   expected_exceptions_count: int,
   expected_exceptions_code: str,

parser: AchFileProcessor):
…
```

We create a function and pass the needed variables to it.

The called function contains the same logic of parsing and asserting the desired variables.

With these basic tests, we can easily start adding exceptions to our code. Of course, more extensive testing will be done at some point, but this is a great way to get moving in the right direction.

8.5.2 Supporting the exceptions in our parser

To begin writing our exceptions, we have to add a schema that can hold the exception information. By providing the UUID for the file, batch, and entry, we have flexibility in reporting the exception at different levels. For instance, when an amount is invalid, we can drill down directly to the record causing the problem because we have access to the UUID for the file, batch, and entry. Compare that with if we only had the `ach_files_id` that caused the exception. If we had thousands of records in the file, we would have our work cut out for us, determining the offending record. The following listing shows our ACH exception schema. We have excluded the field definitions to keep things simple.

Listing 8.10 The ACH exception schema

```
class AchExceptionSchema(BaseModel):
    ach_exceptions_id: Optional[UUID4] = Field(…)
    ach_files_id: UUID4 = Field(…)
    ach_batch_id: Optional[UUID4] = Field(…)
    ach_entry_id: Optional[UUID4] = Field(…)
    record_number: int = Field(…)
    exception_code: str = Field(…)
```

We always associate exceptions with a particular file.

The record number helps locate the unparsed record.

The exception code helps locate the error details.

We then continue creating our `AchExceptionsSql` class that has the usual `insert_record` and `get_record` methods. We could have named these `insert_exception` and `get_exception`, but to stay consistent with the other methods, we keep it similar to our existing code.

Listing 8.11 ACH exceptions SQL

```
class AchExceptionsSql:
    @staticmethod
    def insert_record(ach_file_exception: AchExceptionSchema) -> UUID:
        with get_db_connection() as conn:
            result = conn.execute(
                """
            INSERT INTO ach_exceptions
➥ (ach_files_id, record_number,
              exception_code)
            VALUES (%(ach_files_id)s,
➥%(record_number)s, %(exception_code)s)
            RETURNING ach_exceptions_id
             """,
                 ach_file_exception.model_dump()
             )

        return result.fetchone()[0]

    @staticmethod
    def get_record(ach_exception_id: UUID) -> AchExceptionSchema:
        with get_db_connection
➥(row_factory=class_row(AchExceptionSchema)) as conn:
            result = conn.execute(
                """
                SELECT * FROM ach_exceptions
                WHERE ach_exceptions_id = %s
                """,
                [ach_exception_id]
            )

            return result.fetchone()
```

The exception class containing any SQL calls

Standard SQL to insert and return the newly added ID

Returns the first field from our results, which is the ID

Standard SQL to read the exceptions

Now that we can create exception records, we can simply call an `_add_exception` routine from ach_file_processor.py wherever we need to write out exceptions. Listing 8.12 shows this exception routine. Notice that we take the `AchExceptionSchema` and the unparsed line as parameters. We chose to take the line as an optional parameter and write it out to an invalid record table. This is because we split out the individual record types in our database earlier. To accommodate conditions where we may have a record that could not be parsed because it was not a valid type, we still want to store it somewhere.

Listing 8.12 Adding an exception method

```
    @staticmethod
    def _add_exception(exception: AchExceptionSchema,
➥ unparsed_record=None) -> None:
```

Our function to call when we have an exception during processing

```
        sql = AchExceptionsSql()
        sql.insert_record(exception)
        if unparsed_record is not None:
            sql_invalid_rec = AchRecordsSqlTypeInvalid()
            sql_invalid_rec.insert_record
➥(AchRecordTypeInvalidSchema(
                ach_files_id=exception.ach_files_id,
                unparsed_record=unparsed_record,
                sequence_number=exception.record_number,
            ))
```

When a record has been passed, we need to write it to a special database table since we could not determine the correct table for it (e.g., a record type that does not exist).

With the ability to create exceptions, we need to update our parsing methods to return a custom validation error. The individual field parsing does not change, but we want to capture the validation error and wrap it with a custom error.

Listing 8.13 Updating parse routine

```
    @staticmethod
    def parse_file_header(
            ach_records_type_1_id: UUID, record: str
    ) -> AchFileHeaderSchema:
        try:
            return AchFileHeaderSchema(
…
            )
        except ValidationError as e:
            raise AchParsingValidationError(
   message='Error parsing file header',
   validation_errors=e.errors())
```

Tries to parse the header with Pydantic

Catches when a validation error is encountered

Creates a custom error that stores our ACH related information using the raise keyword

The custom `AchParsingValidationError` is shown in the following listing. The purpose of the custom error is that we can translate the encountered errors to their equivalent `AchException` codes, which facilitates writing them to the database.

Listing 8.14 Custom ACH parsing error

Our error is a subclass of Exception.

Processes each validation from the array self.validation_errors

Defines a function to get a list of exceptions

Translates the field having a problem to one of our exception codes

```
class AchParsingValidationError(Exception):
…
    def get_exception_codes(self) -> list[str]:
        exception_codes = []
        for error in self.validation_errors:
            if error['loc'][0] == 'file_id_modifier':
exception_codes.append(AchExceptions.
➥INVALID_FILE_ID_MODIFIER.value)
            elif error['loc'][0] == 'immediate_destination':
exception_codes.append(AchExceptions.
➥INVALID_IMMEDIATE_DESTINATION.value)
        return exception_codes
```

Then we can update the individual parsing of the record to add the exceptions when encountered.

Listing 8.15 Updates to handle validation errors

```
case "1":
   ach_record = AchRecordType1Schema(
      ach_files_id=ach_file_id,
      unparsed_record=line,
      sequence_number=sequence_number)
   ach_record_id = AchRecordsSqlType1().insert_record(ach_record)
   current_file_header_id = ach_record_id
   try:
      self._parse_file_header(ach_record_id, line)
   except AchParsingValidationError as e:
      for exception_code in e.get_exception_codes():
         self._add_exception(AchExceptionSchema(
            ach_files_id=ach_file_id,
            record_number=sequence_number,
            exception_code=exception_code,
            ))
```

We try to parse the record.

For each error we find, it writes out an exception.

That provides a basic outline for handling of exceptions in the ACH parser. The same formula applies to the remaining record types.

8.5.3 Validation in Pydantic

Much of field-level validation will happen in Pydantic. Let's take a look at some of the exceptions and how we can do them in Pydantic.

First, we would like to address the file ID modifier on the type 1 record. This should be an uppercase letter from A to Z or a numeric value from 0 to 9, which we can easily specify in Pydantic.

Listing 8.16 Pydantic validation of the file ID modifier

```
file_id_modifier: Annotated[str,
   StringConstraints(
      pattern=r'^[A-Z0-9]$',
      min_length=1,
      max_length=1)]
```

We define a regular expression pattern to validate against.

We also limit the field size, although the pattern itself should be enough.

Another condition to cover is the immediate destination. We know that this should match the routing number of our bank. We do not necessarily want to hardcode these values into our code because they can change (although not often), but as previously mentioned, banks that go through acquisitions and mergers may gain (or lose) routing numbers. Therefore, we want a way to have Pydantic validate the immediate destination based on a list of values known at runtime.

This task is accomplished through a `field_validator` annotation in Pydantic. For our purposes, we will hook this into a database call, which lets us validate the field against a list of allowed values in our database.

Listing 8.17 Pydantic field validator for immediate destination

```
@field_validator('immediate_destination')
@classmethod
def validate_immediate_destination(cls, v):
    try:
        if BankRoutingSql().valid_bank_routing_number(v):
            return v
        else:
            raise ValueError('immediate_destination
➥ is not a valid routing number')
    except Exception:
        raise ValueError('Unable to validate
➥ immediate_destination')
```

Uses the field_validator and classmethod annotations

Uses SQL to check the bank routing number and return it if valid

Otherwise, raises a ValueError as it was not found

Handles any unexpected errors with a ValueError as well

To support this, we have to create a simple table that contains the routing numbers we want to consider as our bank routing numbers. Depending on the number of records we expect to validate, having to perform a database query for each record (and possibly multiple fields) may be inefficient. In our case, there should only be a single ACH file header record in any file that we load. So, we are not overly concerned about the effects on performance, but these types of potential problems should be kept in mind.

Once we start validating the immediate origin, we need to go through and correct many of our tests where we had a number that was valid but did not exist in our database. There will be an additional clean-up of unit tests to handle the exceptions and expanding of parameters passed to the routines so that we can log the exceptions. When these methods start getting too complex, it will be time to split them into their own record-type classes away from the parser itself.

8.6 Updating the APIs

After passing the unit tests with the ACH parsing and handling our errors, we can jump over to the dashboard to see things in action. Unfortunately, if you try to load one of the exception files, you will not see it on the dashboard. What happened?

We already have BDD syntax for the API endpoint that supports loading a file and checking the results, so no new code will be needed to support the test shown in the next listing. Having the flexibility to create new tests without having to write any new code is one of the benefits of BDD and a flexible syntax that powers it. Running this test does indeed show no records being returned.

Listing 8.18 Testing our API with an invalid file

```
Scenario: I should get a file even when
➥ it has an exception
```

Defines a scenario

```
Given that I have a clean database
And that I have posted the file "invalid_file_id.ach"
When I request a list of files
Then I should have a file that includes the
➥ file "invalid_file_id.ach"
```

Sets up the database and processes an ACH file

Executes the API

Checks for our included file

Another great benefit of having tests is that we can simply add a breakpoint to the endpoint and rerun the test in a debug mode, which then allows us to step through the code. If you take the time to do this, you should see that no results are being returned from the query.

The queries are not able to handle the parsed record (`ach_file_headers` or `ach_file_control_records`) not being found. Because we used `INNER JOIN`, the record must exist in the database to be returned.

Listing 8.19 Using `INNER JOIN` in our original query

```
SELECT af.ach_files_id AS id,
                      af.file_name AS filename,
                      af.created_at AS date,
                      afh.immediate_origin_name AS originator,
                      afcr.total_debit_entry_dollar_amount AS debit_total,
                      afcr.total_credit_entry_dollar_amount
➥ AS credit_total,
               FROM ach_files AS af
               INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
               INNER JOIN ach_records_type_9 AS art9 USING
                                    (ach_records_type_1_id)
               INNER JOIN ach_file_headers AS afh USING
                                    (ach_records_type_1_id)
               INNER JOIN ach_file_control_records AS afcr USING
                                         (ach_records_type_9_id)
               ORDER BY af.created_at DESC
```

INNER JOINs on these files may cause records not to be returned when exceptions occur.

Both the `ach_file_headers` and `ach_file_control_records` are needed in the query. Now that we have started adding our exceptions and not writing out parsed records, we will also need to be more aware of these queries. Since there is no guarantee that this feature will be present, we should use a `LEFT JOIN` to return a result. The following listing shows updating both parsed versions of our query to use the `LEFT JOIN`. Depending on the workflow we use to work through this problem, we may not want to make changes to both tables until we have tests to verify them.

Listing 8.20 Updating our queries to `LEFT JOIN`

```
INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
INNER JOIN ach_records_type_9 AS art9 USING (ach_records_type_1_id)sd
```

```
LEFT JOIN ach_file_control_records AS afcr
➥ USING (ach_records_type_9_id)
LEFT JOIN ach_file_headers AS afh
➥ USING (ach_records_type_1_id)
ORDER BY af.created_at DESC
```

Updating our queries to LEFT JOIN resolves the problem by keeping the data in the left table.

With the updates to the query, we now see our tests passing. We may also look at these queries and think that we have spent all the previous sections adding exceptions, but we neither have any APIs handling them nor have we updated any of our existing APIs to consider them as well.

When thinking about including exceptions in our dashboard, we will likely want APIs that can help us with the following:

- *Return all exceptions*—Regardless of whether we are looking for all exceptions ever or just exceptions encountered today, we need some insight into the exceptions that have been encountered. Also, if we are interested in allowing users to perform maintenance on the exceptions (change exception levels), this is a good starting point.
- *Return exceptions for a file*—There will certainly be a need to get exceptions for a specific file.

There are more use cases for our APIs, but these should be sufficient to get us moving with some of the APIs. One design choice we will have to make is the endpoint layout. Do we create a new endpoint so that our exceptions are like our existing `/files` endpoint (something similar to `/api/v1/exceptions`), or are exceptions part of our files (something similar to `/api/v1/files/exceptions`). Will we use a combination of the two? Since the exceptions will be associated with the loaded files, we keep our exceptions associated with the `/files` endpoints.

Next, we want to ensure that our exception APIs are returning the expected data. For instance, when we load the invalid_file_id.ach, we expect an error code and message that will be returned from the database. We can use most of our existing code in the test_file_api.py and only include a few other steps. Starting with this, we can create a test we then verify against. We created a new file named ach_files_exception_endpoint .feature that stores our exception features.

Listing 8.21 Testing our exceptions API

Sets up a scenario for the exceptions API

```
Feature: Exceptions returned for files
  Test the functionality of the exception returned for loaded ACH files

  Scenario: I get a list of exceptions for a file
    Given that I have a clean database
    And that I have posted the file "invalid_file_id.ach"
    When I request a list of exceptions for the
➥ file "invalid_file_id.ach"
    Then I should receive an error of "004" and a
➥ message of "Invalid File ID Modifier"
```

Ensures an empty database and loads the file

Calls the API and validates the exception

To support the new `When` clause, we must add code that will request a list of files so that we can determine the UUID. After that, we use the obtained ID to make a request to the exception endpoint. To avoid any unexpected errors, we could also ensure that we have one record returned from the response, as well as a valid response code. However, since we have cleared the database and only loaded a single file, it should be an exceptional condition that causes this to break.

Listing 8.22 Adding a new `When` clause

```
@when(parsers.parse('I request a list of exceptions
➥ for the file \"{filename}\"'))
def get_list_of_exceptions_for_file(api_response, filename):
    response = client.get("/api/v1/files/")
    assert response.status_code == 200, response.text
    file_id = response.json()[0]["id"]
    response = client.get
➥(f"/api/v1/files/{file_id}/exceptions")
    assert response.status_code == 200, response.text
    api_response["response"] = response.json()
```

Gets a list of files and ensures that it worked

Extracts the file_id from the response

Saves the response for use in other steps

Uses the file_id we just extracted from the response, makes another API call to our exceptions API, and validates the HTTP response

We have our stored API response, and now we want to ensure that we are getting the expected error code and message from our response. Why are we validating both the error code and the message text coming back? Should the error code not be the determining factor? On the one hand, the consumer of the API should rely on the error code and not the error message description. On the other hand, we want to ensure that both the error code and description match our expectations. Furthermore, we may want to identify when the description changes as maybe some consumers of our API work directly with the description rather than with the error code. Perhaps the API was incorrect at some point and returned the same error code for different errors.

To work around the error, the consumers may have directly interrogated the description, and changes to it may break the API. Of course, if we make a change to that API, we may choose to change the version so that consumers can move to the updated API when they are ready. With all that said, we just need to be aware that changes can affect end-users in unexpected ways. We have seen instances where customers have implemented their own scraping of log messages for their own purposes. When those messages (even though they were internal to the application) changed, the infrastructure broke. In the end, the messages needed to be put back in place. So, be very careful—seemingly innocuous changes may cause problems.

Listing 8.23 Adding a new `Then` clause

```
@then(parsers.parse('I should receive an error
➥ of \"{error_code}\" and a message of \"{error_message}\"'))
def check_error_code_exists_on_response(error_code,
➥ error_message, api_response):
    assert any(
        error_code == response["exception_code"]
➥ for response in
api_response["response"]
    ), f"Expected {error_code} in"
➥ f" {api_response['response']}"
    assert any(
        error_message == response["description"]
➥ for response in api_response["response"]
    ), f"Expected {error_message} in" f" {api_response['response']}"
```

Using any, we can search for a field in our response. This tactic is convenient when dealing with an array of values and we do not know the order of the response.

We now have tests for these APIs to determine whether they are returning data successfully. As a programming challenge, can you create the logic behind these APIs and get the tests to pass?

We create the endpoint for the exceptions `/{file_id}/exceptions` as shown in the following listing.

Listing 8.24 An exceptions endpoint for a file

```
@router.get(
    path="/{file_id}/exceptions",
    response_model=list[AchExceptionsResponse],
    summary="Retrieve ACH File Exceptions",
    description="Retrieve the exceptions in an ACH file.",
    response_description=
➥"The exceptions in the requested ACH file.",
    tags=["ACH Files"],
)
async def read_exceptions_for_file(file_id: UUID) ->
list[AchExceptionsResponse]:
    return AchExceptionsSql()
➥.get_exceptions_response(file_id)
```

Our API details that will be used for OpenAPI documentation

The actual function

Next, we create the `get_exceptions_response` method.

Listing 8.25 The `get_exceptions_response` method

```
    @staticmethod
    def get_exceptions_response(ach_files_id: UUID = None) ->
     list[AchExceptionsResponse]:
        query_parameters = []
        sql = """
                    SELECT
                        afe.ach_exceptions_id AS id,
                        afe.ach_files_id AS file_id,
```

Standard SELECT statement with our desired fields

```
            afe.ach_records_type_5_id AS batch_id,
            afe.ach_records_type_6_id AS entry_id,
            afe.record_number AS record_number,
            afe.exception_code AS exception_code,
            aec.exception_description AS description
        FROM ach_exceptions AS afe
        INNER JOIN ach_exception_codes AS aec
                USING (exception_code)
    """
    if ach_files_id is not None:
        sql += " WHERE afe.ach_files_id = %s"
        query_parameters.append(ach_files_id)

    with get_db_connection(row_factory=class_row(AchExceptionsResponse))
as conn:
        result = conn.execute(sql, query_parameters)
        return result.fetchall()
```

Our initial table is the ach_exceptions.

Joins our exceptions with the exception codes for more details

If we received a specific ach_files_id, then use that to filter our results by appending a WHERE clause.

Executes the query and returns the results

Our test should be passing. If we step through the code and examine the response that we receive, we should see the `batch_id` and `entry_id` are both set to `None`, as shown in the example response.

Listing 8.26 Example response

```
[{…, 'batch_id': None, 'entry_id': None,
➥ 'record_number': 1, 'exception_code': '004',…}]
```

We may consider updating our SQL query to set those returned values to empty strings by using `COALESCE`—for example, `COALESCE(afe.ach_records_type_5_id, '') AS batch_id`. However, because we have defined our Pydantic models to expect a UUID, and the empty string is not a valid UUID, we will receive an `InvalidTextRepresentation` error. Because we are returning JSON, a better alternative may be to drop this value from the response. Note that this may not always be the best option. For instance, if we wanted to note the explicit absence of the value, it may be desirable to return something to indicate the value is missing. However, for our purpose, dropping the value works just fine. The following listing shows how to drop the `None` value from our response. Since we are not interested in it, we will see that our UI will handle the missing value without our help.

Listing 8.27 Dropping `None` values from our model

```
@router.get(
    path="/{file_id}/exceptions",
    response_model=list[AchExceptionsResponse],
    summary="Retrieve ACH File Exceptions",
    description="Retrieve the exceptions in an ACH file.",
    response_description="The exceptions in the requested ACH file.",
    response_model_exclude_none=True,
    tags=["ACH Files"],
)
```

Drops the fields containing None

With the `response_model_exclude_none` set to `True`, we should be ready to move on to the UI to start taking advantage of our exception API. Now that we have declared several endpoints within files.py, we want to ensure that we are paying attention to the order we declare them in. We make sure that we are going from general endpoints such as `/files` and `/exceptions` to more specific, such as `/{file_id}/exceptions`. If we encounter a `422 error`, it could be an indication that FastAPI has picked up the wrong endpoint to use.

8.6.1 Retrieving an unparsed record

Another piece of functionality we need for the dashboard is the ability to retrieve an unparsed record from the database. As we have seen, when there is a parsing exception, we are not writing the parsed record out. It will be helpful to an end-user if they can see what the record causing the exception looked like. Of course, they could pull up the file themselves and look at it, but that is not very user friendly. We would also recommend passing the record back all the time (which would make this request more straightforward) because the unparsed record could contain nonpublic information (NPI) data. By creating this as a separate endpoint that returns the record, say when a button or icon is clicked, we can call out to the specific endpoint to retrieve the data, which allows us to take preventative measures such as

- Hiding the button to display the record when the user is not authenticated.
- Creating an audit trail when the information is viewed
- Preventing needless data from being returned since users will not always need to see the unparsed record

We can start by creating a scenario to test this approach.

Listing 8.28 Scenario for an unparsed record

```
Scenario: I get the unparsed record for a
➥ specific exception                                  <-- Scenario for an unparsed records API
    Given that I have a clean database
    And that I have posted the file "invalid_file_id.ach"
    When I request the unparsed record for the exception code "004"
    Then I should receive an
➥ unparsed record of "1…i…XXXXX"                      <-- Contains the entire unparsed record
```

The following code creates a step that helps support the retrieval of the unparsed record for a given exception. We assume that we are only getting one file returned and that the only exception is the one we want. Later, we may need to make the code more robust, but there is no need for now.

Listing 8.29 Step definition for retrieving an unparsed record

```
@when('I request the unparsed record for the exception code "004"')
def request_unparsed_record(api_response):
```

```
    response = client.get("/api/v1/files/")
    assert response.status_code == 200, response.text
    file_id = response.json()[0]["id"]
    response = client
➥.get(f"/api/v1/files/{file_id}/exceptions")
    assert response.status_code == 200, response.text
    exception_id = response.json()[0]["id"]
    response =
 client.get(f"/api/v1/files/{file_id}
➥/exceptions/{exception_id}")
    assert response.status_code == 200, response.text
    api_response["response"] = response.json()
```

Similarly to previous API tests, we must make several calls to drill down to the data we are interested in. With each API call, we must validate a successful 200 response code before moving on to prevent errors in subsequent calls.

We store the results of our final API call for further use.

If we run this code, we are likely to get a `404 error` since we have not defined an endpoint. You should see one of the `client.get` calls failing with the error:

```
Expected :200
Actual   :404
```

What steps need to be taken to fix this error?

- We need to create a new route to get the unparsed record for a given `exception_id` in the files.py.
- We update the ach_exceptions_sql.py to return the `unparsed_record` from the database.

We will cover this code in chapter 9 when talking about auditing. For now, feel free to take a shot at completing it.

8.7 Insights into exceptions with the UI

In previous sections, we have built the ability to create exception files, identify those exceptions, log them appropriately, and finally return the information from an API. Now, we can look at how to incorporate viewing the exceptions in our dashboard. We likely want to provide the users with successively more specific views that drill down into the exceptions. For example,

- Exceptions for all files are useful for operations to get a sense of the exceptions in the system and for analyzing the number/types of exceptions being encountered.
- Exceptions for a specific file could be useful for determining if the file needs to be removed and reprocessed.
- Exceptions for a specific batch are useful for determining if a particular company is having a problem. Also, a file could still be processed with only a specific batch being rejected.
- Exceptions for a specific entry may be useful to work with a client or company to resolve what caused an exception.

Is it necessary to provide all these views? Certainly, the customers will have their input, and the business will provide us with direction. In addition, the business will take our input as well. Some of these may be easier or more efficient, depending on how things are set up. In our case, we can easily provide exceptions for all files and specific files, so we will start there.

8.7.1 Showing all the exceptions

The easiest way to get insight into all exceptions is to provide a navigation button on the left-hand side of the screen to view all our exceptions for the day (we say "day," but we are going to leave filtering by day as an exercise for you). Remember that, eventually, to provide a better UX, we need to offer users the ability to search and filter. Multiple ACH files will be loaded daily and users will soon become overwhelmed, or the dashboard will become unresponsive if we are not careful with the amount of data we are pushing to the user. When we get to the point where the number of exceptions starts causing problems with the performance and UX, we will need to look at limiting the data by providing pagination, searching capabilities, and other strategies to limit the data being retrieved at a time.

With that in mind, we can review updating NavButtons.tsx to navigate to the exceptions page.

Listing 8.30 Adding the error navigation to the dashboard

```
<ListItemButton onClick={() => route.push("/exceptions")}>
   <ListItemIcon>
      <Error/>
   </ListItemIcon>
   <ListItemText primary="Exceptions"/>
</ListItemButton>
```

Once we have the button, we will need to create the actual page. The page should meet the following requirements:

- Displays a list of all exceptions
- Allows viewing the record causing the exception
- Enables navigation to the file when the row is clicked

Those are some straightforward requirements. Let's see how to accomplish them. First, we want to display the exceptions using a DataGrid element as we did for some of the other data on our pages. Recall that the DataGrid element from Material UI (MUI) allows us to sort, search, and filter, making it fully functional with very little work on our part. The following listing shows the basic API call used to retrieve exceptions from http://localhost:8000/api/v1/files/exceptions and populate them to our DataGrid.

Listing 8.31 API Calls to populate the DataGrid

```
…
    const [entries, setEntries] = useState<AchExceptionsResponse[]>([]);
…
    useEffect(() => {
        const apiUrl = process.env
➥.NEXT_PUBLIC_API_URL ?? '';
        axios.get<AchExceptionsResponse[]>
➥(`${apiUrl}/files/exceptions`,
            {
            headers: {
                'Content-Type': 'application/json'
            }
        })
            .then(response => {
                console.log(`Response data
${JSON.stringify(response.data)}`);
                setEntries(response.data);
            })
            .catch(error => {
                console.log(error);
            });
    }, []);
…
```

The URL for our API

Axios call to get all exceptions

With DataGrid, we are taking another approach to show additional functionality (listing 8.32). We define a column consisting of an info icon and use `renderCell` to add the icon. This can be a handy way to control the contents of the cell, especially in the case where we are not mapping something directly to our table. We also make use of another function—the `valueGetter`—to convert the date we receive using our `convertDateFormat` we have defined previously.

Listing 8.32 Defining the DataGrid

```
…
    const columns: GridColDef[] = [
        {field: 'view', headerName: 'View',
➥ sortable: false, width: 10, renderCell: (params) => (
                <IconButton
                    onClick={(e) => {
                        e.preventDefault();
                        setIsOpen(true);
                    }}
                    color="primary"
                    ▸
                    <InfoIcon />
                </IconButton>
            )},
        {field: 'file_name', headerName: 'Filename', width: 150},
        {field: 'created_at', headerName: 'Date',
➥ width: 150, valueGetter: (params) => convertDateFormat(params.value)},
```

Defines a clickable icon that can be used to show the unparsed record

```
        {field: 'record_number', headerName: 'Record Number', width: 150},
        {field: 'exception_code', headerName: 'Code', width: 10},
        {field: 'description', headerName: 'Description', width: 300},
    ]
…
<DataGrid columns={columns} rows={entries}/>
…
            <AlertMessage open={isOpen} setOpen={setIsOpen}
      message="Exception Record" title="Record Exception" />
…
```

The grid of exceptions uses the MUI DataGrid component.

Reuses our AlertMessage component and shows the exception record

With our API and DataGrid complete, we should now be able to upload a file with an exception, navigate to this page, and see our exception listed. When the info icon is clicked, we render a static message as a placeholder. Instead of that static message, let's use an API call instead to retrieve the actual record so we can display the unparsed record. Why not return the unparsed record when we pass the list of exceptions back? We could certainly do that; however, there are a few reasons to avoid it:

1. *Passing back unnecessary data*—Since it is unlikely that a user is going to click on every exception in the list, a lot of that data will remain unused.
2. *Separation of duties*—We want this API to function as an overview of our exceptions. Returning the data using a separate API call helps keep the code for the API and SQL simpler.
3. *Security considerations*—The unparsed records could have account numbers and other sensitive NPI data that we do not want to be passed back all the time. By making it a separate API call, we can control who may be able to view those details and also log exactly when someone views that data.

With the understanding that showing the unparsed record could open up showing NPI data, we can make a call to get the unparsed record and display it in the alert box. Of course, we may eventually want to make use of the other details we have for the exception and create a better-formatted window. For now, simply displaying the record in question is enough. We can use the `params.row` to retrieve the needed information from the row that was clicked. The following listing shows the updated code that goes into the icon button.

Listing 8.33 The updated icon button

```
<IconButton
   onClick={(e) => {
      e.preventDefault();
      const fileId = params.row.file_id;
      const exceptionId = params.row.id;
      const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
      axios.get<AchExceptionDetailsResponse>(
         `${apiUrl}/files/${fileId}/exceptions/
➥${exceptionId}`, {
```

Retrieves the IDs for both the file and exception for the row

Our API URL

Axios call to get the unparsed record

```
          headers: {
             'Content-Type': 'application/json'
          }
       })
       .then(response => {
          setUnparsedRecord(response.data.unparsed_record);
          setIsOpen(true);
       })
       .catch(error => {
          setUnparsedRecord(error.message)
          setIsOpen(true);
       });
    }}
color="primary">
…
</IconButton>
```

Upon response, sets the unparsed record and the Boolean for the dialog

If there is an error, shows it in the dialog instead

We now have a way to view exceptions and the record causing the error. That should be useful to our end-users. However, should they click on exceptions whenever an ACH file is uploaded? Let's update our initial Recent ACH Uploads screen to have an indicator next to the filename when it is a file that has exceptions.

8.7.2 A visual cue for exceptions

When we first view our dashboard, there should be some indication that a loaded file has exceptions. We do this by updating our dashboard to include an icon whenever a file contains exceptions. We make the following updates:

1. SQL query needs to return a Boolean value indicating a file has exceptions.
2. API needs to return the new field as `has_exceptions`.
3. UI needs to check this field to display an icon when it is true.

We first create a couple of test scenarios so that we can work on the first two items on our list. The following listing shows the newly added scenarios.

Listing 8.34 Scenarios testing `has_exceptions`

```
Scenario: I should not have any exceptions
  Given that I have a clean database
  And that I have posted the file "sample.ach"
  When I request a list of files
  Then the has_exceptions field should be False

Scenario: I should get a file even when it has an exception
  Given that I have a clean database
  And that I have posted the file "invalid_file_id.ach"
  When I request a list of files
  Then I should have a response that includes
➥ the file "invalid_file_id.ach"
  And the has_exceptions field should be True
```

The `get_files_response` can be updated to include the `has_exceptions` field. The following listing shows how we use a Common Table Expression (CTE) to get a list of unique `ach_files_id` values and then convert those to a Boolean value to use in `has_exceptions`.

Listing 8.35 Updating SQL for `get_files_response`

```
WITH exceptions AS (
   SELECT DISTINCT(afe.ach_files_id)
   FROM ach_exceptions AS afe
)
SELECT af.ach_files_id AS id,
       af.file_name AS filename,
       af.created_at AS date,
       afh.immediate_origin_name AS originator,
       afcr.total_debit_entry_dollar_amount AS debit_total,
       afcr.total_credit_entry_dollar_amount AS credit_total,
       CASE
          WHEN exceptions.ach_files_id IS NOT NULL THEN TRUE
       ELSE FALSE
       END AS has_exceptions
       FROM ach_files AS af
INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
INNER JOIN ach_records_type_9 AS art9 USING (ach_records_type_1_id)
LEFT JOIN ach_file_headers AS afh USING (ach_records_type_1_id)
LEFT JOIN ach_file_control_records AS afcr USING (ach_records_type_9_id)
LEFT JOIN exceptions USING (ach_files_id)
ORDER BY af.created_at DESC
LIMIT %s
OFFSET %s
```

Creates a CTE to get the distinct file IDs

Uses a case statement to set a Boolean field called has_exceptions

Since not every file has exceptions, uses LEFT JOIN

And now we can update the UI to support the new `has_exceptions` field as well.

Listing 8.36 Updated API mapping

```
const transformedData: AchFiles[] = response.data.map
➥((row: AchFilesResponse) => ({
   id: row.id,
   date: row.date,
   filename: row.filename,
   originator: row.originator,
   creditTotal: new Decimal(row.credit_total),
   debitTotal: new Decimal(row.debit_total),
   hasExceptions: row.has_exceptions
}));
```

The new field to determine if a file has exceptions

And the updated RecentAchUploads.tsx

Listing 8.37 Displaying an error icon

```
import {ErrorOutlineOutlined} from "@mui/icons-material";
…
```

```
<TableCell>
   <Link onClick={() => route.push(`/fileDetails/${file.id}`)}
         sx={{ cursor: 'pointer' }}>{file.filename}</Link>
      {file.hasExceptions &&
➥  <ErrorOutlineOutlined color="error" />}
</TableCell>
```

If a file has exceptions, displays an icon to indicate it has them

With that, when an ACH file with exceptions is uploaded, there should be a visual indication on our Recent ACH Uploads, as shown in figure 8.4.

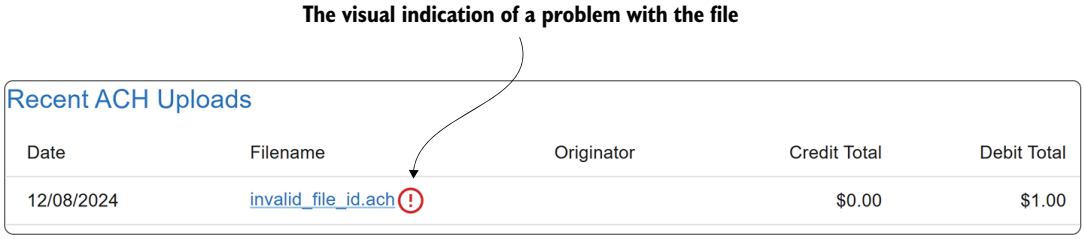


Figure 8.4 Visual indication of a file error

8.8 Additional UI changes

Now that we have the basic structure of our exception viewing, we can take care of a few other requests that came from users during the initial MVP evaluation.

1. The files screen should also show the specific exceptions for that file. This should follow after our initial exception handling. We can refactor our UI exception page to become a component and pass the needed exception information into it.
2. The batch screen should contain a component with statistics such as totals and a line-item breakdown of the transaction codes.
3. Our listing of the items in the batch should contain the applications they are associated with, such as Checking, Savings, and so forth. While the transaction codes relate to those applications, ensuring they are included in our DataGrid will provide a better UX.
4. Our screens should be more responsive to size changes. Currently, our components are stacked in a column, and we would like to have them use up the available width as well.

8.8.1 Creating an exceptions component

We can refactor our page to create an exception component, which will help prevent code duplication when we want to present errors elsewhere. The exception information that we need to display is the same, regardless of whether we want to view all exceptions or just filter them by file. On the one hand, that means we could easily copy and

paste that existing page to create a new page or component that filters on our file, and we would be done. On the other hand, we could create a common exception component that both pages can use, passing to it either the full list of exceptions or one that has been filtered for a particular file. As we have stated previously, whenever feasible, we want to avoid duplicating work as that may pay off in the short term by getting something out the door, but future maintenance becomes difficult and tedious as fixes must be applied multiple times. The following listing shows the results of pulling the DataGrid and the associated calls from the original page into a stand-alone component.

Listing 8.38 Stand-alone exception component

```
…
interface ExceptionsProps {
    exceptions: AchExceptionsResponse[];
}

export default function Exceptions({exceptions}:
➥ Readonly<ExceptionsProps>) {

    const [isOpen, setIsOpen] = useState(false);
    const [unparsedRecord, setUnparsedRecord] = useState('');
    const columns: GridColDef[] = [
…
            <DataGrid columns={columns} rows={exceptions}/>
…
            <AlertMessage open={isOpen} setOpen={setIsOpen}
➥ message={unparsedRecord} title="Unparsed Record" />
…
```

With our new exceptions component, we can include it in the existing `fileDetails` page by calling the `/file/{fileId}/exceptions` endpoint to get a list of exceptions for the file we are viewing and passing the results to our new component. The following listing shows the required changes.

Listing 8.39 Using our new component

```
…
import Exceptions from "@/app/components/Exceptions";
import {AchExceptionsResponse} from "@/app/interfaces/AchExceptionsResponse";
…  axios.get<AchExceptionsResponse[]>(`${apiUrl}/files/
➥${fileId}/exceptions`, {
            headers: {
                'Content-Type': 'application/json'
            }
        })
            .then(response => {
                console.log(`Response data ${JSON.stringify(response.
    data)}`);
                setExceptions(response.data);
            })
            .catch(error => {
```

Sets the returned exceptions

```
                console.log(error);
            });

    }, []);
…
      <Exceptions exceptions={exceptions} />
…
```

Includes our new component passing the exceptions to it

We love how by creating components, it becomes so easy to import and use them anywhere we may need them. The only thing we want to stress about this is to be careful about the data that a component may need or fetch. We may have to evaluate whether we want to pass data to it or have it request data itself. As we saw previously with the dashboard, sometimes we can save redundant API calls by passing the required data in. We did leave the fetching of our unparsed records within the component because those are occasional calls that are best encapsulated in the component.

8.8.2 Batch statistics component

We can now tackle the batch statistics component. This request is meant to give users a quick overview of some important aspects of the batch, as shown in figure 8.5.

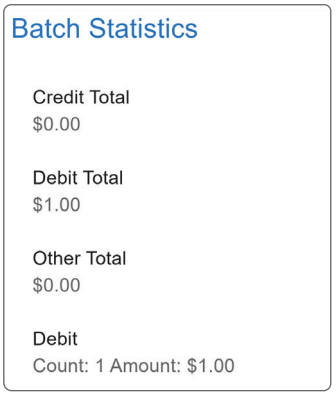


Figure 8.5 Batch statistics component

While charts and graphs provide nice visualizations and can help compare large amounts of data, we will use a simple list for this component as we only need to provide a handful of totals for our overview.

We will continue to use the existing `AchBatchEntriesResponse` for this component. Using a simple list and items from MUI, we can create our component. We use the `ListItemText` for displaying primary and secondary text. We also create some of the elements with `createListItems` and include them in the render. The following listing shows our basic component.

Listing 8.40 The `AchBatchStatistics` component

```
export default function AchBatchStatistics({entries}:
➥ Readonly<AchBatchStatisticsProps>) {

    const computedBatchStatistics =
➥ computeBatchStatistics(entries);
    const computedBatchTotals =
➥ computeBatchTotals(entries);
    const listItems =
➥ createListItems(computedBatchStatistics);

    return (
```

Helper functions to collect, aggregate, and format data as necessary

```
        <>
           <Title>Batch Statistics</Title>
           <List dense={true}>
              <ListItem key="creditTotal">
                 <ListItemText
                    primary="Credit Total"
                  secondary={`${formatCurrency(computedBatchTotals.credit)}`}
                 />
              </ListItem>
              <ListItem key="debitsTotal">
                 <ListItemText
                    primary="Debit Total"
                  secondary={`${formatCurrency(computedBatchTotals.debit)}`}
                 />
              </ListItem>
              <ListItem key="otherTotal">
                 <ListItemText
                    primary="Other Total"
                 secondary={`${formatCurrency(computedBatchTotals.other)}`}
                 />
              </ListItem>
                 {listItems}
              </List>
        </>
    );
}
```

Includes any additional ListItem elements

In the following listing, we created the `computeBatchTotals` function to provide a sum of our debits, credits, and other amounts. We will end up supporting additional transaction codes, but for now, we will only be considering the transaction codes 22 and 27. Also, because the `AchBatchEntriesResponse` had formatted the amounts, we needed to strip those characters away. Ideally, we would like to go back and not have the response format those amounts, but that may not always be practical.

Listing 8.41 The `computeBatchTotals` function

```
function computeBatchTotals
➥(entries: AchBatchEntriesResponse[]): BatchTotals {
    const totals: BatchTotals = {
        debit: new Decimal(0),
        credit: new Decimal(0),
        other: new Decimal(0)
    };
    entries.forEach(entry => {
        const cleanAmount = entry.amount.replace(/[$,]/g, '');
        const amount = new Decimal(cleanAmount);
        if (entry.transaction_code === 22) {
            totals.debit = totals.debit.plus(amount);
        } else if (entry.transaction_code === 27) {
            totals.credit = totals.credit.plus(amount);
        } else {
            totals.other = totals.other.plus(amount);
        }
```

Initializes the BatchTotals object

We need to remove the currency formatting.

```
    });
    return totals;
}
```

Returns the BatchTotals object

In listing 8.42, we also create the `computeBatchStatistics` function that contains counts and amounts for each transaction. For simplicity's sake of the code and testing, we create another function that spins through the entries in the batch. We group the items by their description because we will make use of that when we display them.

Listing 8.42 The `computeBatchStatistics` function

```
function computeBatchStatistics
➥(entries: AchBatchEntriesResponse[]):
➥ Map<string, BatchStatistics> {
    const countMap = new Map<string, BatchStatistics>();
    entries.forEach(entry => {
        const cleanAmount = entry.amount.replace(/[$,]/g, '');
        const stats = countMap.get(entry.transaction_description)
➥ || { count: 0, amount: new Decimal(0) };
        stats.count += 1;
        stats.amount = stats.amount.plus(cleanAmount);
        countMap.set(entry.transaction_description, stats);
    });
    return countMap;
}
```

Each entry in the batch will be processed.

Defines a new map of the BatchStatistics

Removes the formatting from the amount

Saves the entry

Uses the plus method from the Decimal package to maintain the amount

Increments the count

Tries to get an entry for the description; if not found, initializes one

We are ready to add the `BatchStatistics` we returned to our component. We dynamically create our `ListItem` elements in a function named `createListItems`, which takes our map of items and returns an array of React elements, as the following listing shows. With this approach, we can create as many elements as needed and return them in one swoop. Of course, we would need to be careful not to create too many elements, or otherwise create something that will not provide a meaningful user experience.

Listing 8.43 The `createListItems` function

```
…
interface BatchStatistics {
    count: number;
    amount: Decimal;
}
```

Defines an interface so that we can easily reference our data

```
…
function createListItems
➥(entries: Map<string, BatchStatistics>): React.ReactNode[] {
    const listItems: React.ReactNode[] = [];
    entries.forEach((entry: BatchStatistics, key: string) => {
        const listItem = (<ListItem key={key}>
            <ListItemText primary={key}
                  secondary={`Count: ${entry.count} Amount:
${formatCurrency(entry.amount)}`} />
        </ListItem>);
        listItems.push(listItem);
    });
    return listItems;
}
```

Creates an empty array of nodes

Creates a ListItem element

Adds it to the array

Returns the array

Finally, we can add some unit testing to this component. While we could have started with this approach, we also want to show that adding unit tests can be done at other points in our development as well. Typically, we add these unit tests at the point where we realize that we need something extra to process the data being returned. To make our testing easier, we will need to export the functions we used within the component. That may present opportunities to refactor the code to further remove the logic from the component, but for now, we will not take it that far.

Listing 8.44 Jest test for `computBatchTotals`

```
…
describe('Compute Batch Totals', () => {
   it('Has a debit for 1', () => {
      const achBatchEntriesResponse:
➥ AchBatchEntriesResponse[] = [{
        id: 'test1',
        transaction_code: 27,
        application: "Checking",
        transaction_description: "Debit",
        amount: "1.00",
        individual_name: "John Doe",
        account_number_last_4: "*********1234",
        addenda_count: 0
      }]
      const result = computeBatchTotals
➥(achBatchEntriesResponse)
      expect(result.debit.toNumber())
                        .toBe(new Decimal("1").toNumber())
})
```

Creates mock data

Passes it to our computeBatchTotals routine

Validates the result

From there, we can add additional entries for the same transaction code, different transaction codes, and anything else we may be interested in testing. This is a great way to test some of the edge cases that may require us to deliberately provide data, especially if we need to test exception conditions. With this basic test, we can create additional tests and start evaluating test coverage before moving on to our next enhancement request.

8.8.3 Adding the application

A relatively straightforward request is to add an application column, as shown in figure 8.6. While it is possible to determine the application from the transaction code for a better UX, we can provide the mapping as part of our returned data.

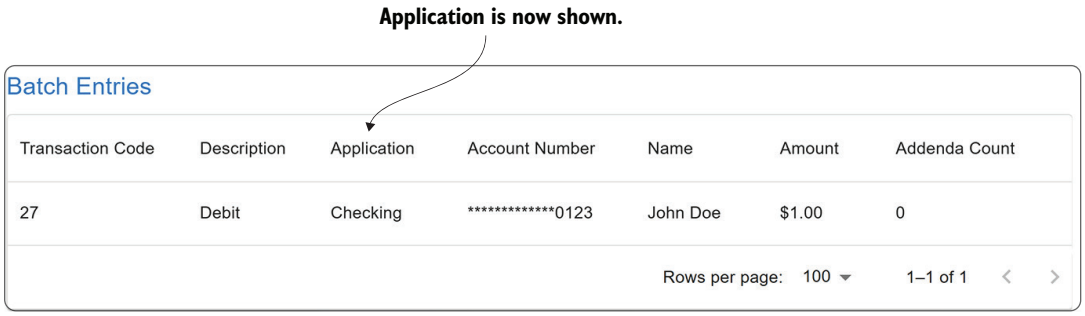


Figure 8.6 The batch entry screen with the added column

This type of request allows us to work through all the pieces of our application, letting us get another taste of the full stack experience using BDD. We can create a feature file named ach_entries_endpoint.feature containing the scenario, as shown in the following listing.

Listing 8.45 Our feature file and scenario

```
Feature: The /files/{fileId}/batches/{batchId}/entries endpoint
  Test returning entries for a batch

  Scenario: I want to check the entries for a Checking Application
    Given that I have a clean database
    And that I have posted the file "ppd-single-credit.ach"
    When I request entries for a file and batch
    Then I should have a response that includes all
➥ applications of "Checking"
```

The scenario shares some similarities with how we tested the files endpoint; however, we chose to create a separate step definition file to support this feature. This decision allows us to keep the scenario-specific steps isolated and gives us the ability to test only this particular feature. The following listing shows some of the step definitions that support running the scenario.

Listing 8.46 Step definitions for our feature

```
…
scenarios("../features/ach_entries_endpoint.feature")
```

```
…
@when('I request entries for a file and batch')
def request_unparsed_record(api_response):
    response = client.get("/api/v1/files/")
    assert response.status_code == 200, response.text
    file_id = response.json()[0]["id"]
    response = client.get(f"/api/v1/files/{file_id}/batches")
    assert response.status_code == 200, response.text
    batch_id = response.json()[0]["id"]
    response = client.get(f"/api/v1/files/{file_id}/
➥batches/{batch_id}/entries")
    assert response.status_code == 200, response.text
    api_response["response"] = response.json()


@then(parsers.parse('I should have a response
➥ that includes all applications of \"{application}\"'))
def response_that_includes_file(application, api_response):
    assert all(
        response["application"] == application for
➥ response in api_response["response"]
    ), f"Expected {application} in" f" {api_response['response']}"
…
```

With the ability to test the endpoint, we can now work on ensuring the application will be returned as part of the API. We can tackle these steps in several different orders, but we chose to start by updating the `AchBatchEntriesResponse` to include the application field, as shown in the following listing. This field is not coming from the database, and it is not technically part of the ACH file layout. Therefore, we limit the maximum length of the field to the longest word we plan on having there, which is "checking."

Listing 8.47 The updated `AchBatchEntriesResponse`

```
…
application: str = Field(
   default=...,
   description="The application that the entry is associated with.",
   title="Application",
   max_length=8,
)
…
```

Sets the max length for this field to the longest text we expect

Next, we update our query to pass back the application name based on the transaction code using a `CASE` statement, as shown in listing 8.48. As mentioned, there is some freedom in application naming. We could potentially choose to group both checking and savings accounts under the name Demand Deposit Accounts (DDA), which is another term used in the financial industry. The actual name for the application would be something that the business provides to us. We like having a bit more granularity in the application name, so we chose to use the names "Savings," "Checking," "GL" for

General Ledger, "Loan," and "Unknown" (to identify any transaction codes that we may have missed or in case of an incorrect transaction code in the database).

Listing 8.48 Updated query to support the `transaction_code`

```
CASE
   WHEN aepd.transaction_code IN (31, 32, 33,
➥ 34, 36, 37, 38, 39)
      THEN 'Savings'
   WHEN aepd.transaction_code IN (21, 22, 23,
➥ 24, 26, 27, 28, 29)
      THEN 'Checking'
   WHEN aepd.transaction_code IN (41, 42, 43, 46, 47, 48)
      THEN 'GL'
   WHEN aepd.transaction_code IN (51, 52, 53, 55, 56 )
      THEN 'Loan'
   ELSE 'Unknown'
END AS application,
```

Maps transaction codes to the application

Names the resulting column as application

Finally, we want to update our DataGrid to add the column. With all the heavy lifting already done by the server, we simply need to add the column and update the interface to include the application as well.

Listing 8.49 Added column and field

```
const columns: GridColDef[] = [
…
   { field: 'application',
     headerName: 'Application',
     width: 150},
…
export interface AchBatchEntriesResponse {
…
    application: string;
…
}
```

Includes the application column in our columns definition

We should now be able to navigate to our dashboard and drill down to see the application column being populated. This small example is a great way to illustrate why we love full stack development. We were able to work on unit testing, Python, APIs, and Typescript!

8.8.4 Making our UI more responsive

Currently, our dashboard components are stacked on top of each other. The MUI package that we have been using for our components provides options for controlling the layout that will help make our dashboard more responsive. We replace the `Box` elements that we wrapped our components in with a `Grid` container and `Grid` items. In addition, there are attributes `xs`, `sm`, `md`, `lg`, and `xl` that can be used to provide

breakpoints based on screen size. This means that we can customize our layouts for all different screen sizes. The following listing shows an example of using a grid with various breakpoints. The given items will be displayed in a column when the screen is small, and when expanded, it will contain more items per row, with all nine items in a single row when the screen is wide enough to accommodate them.

Listing 8.50 Example of a responsive grid

```
import React from 'react';
import { Paper, Typography } from '@mui/material';
import Grid from '@mui/material/Unstable_Grid2';


const ResponsiveCards = () => {
    return (
        <Grid container spacing={2} sx={{pl: 1, pr: 1}}>
            {[1,2,3,4,5,6,7,8,9].map((value) => (
                <Grid xs={12} sm={6} md={4}
➥lg={2} xl={1} key={value} >
                    <Paper elevation={3} sx={{ padding: 2 }}>
                        <Typography variant="h5" component="h3">
                            Card {value}
                        </Typography>
                        <Typography component="p">
                            This is some content
➥ inside card number {value}.
                        </Typography>
                    </Paper>
                </Grid>
            ))}
        </Grid>
    );
};
export default ResponsiveCards;
```

Uses the latest Grid element

Defines a container grid

Defines nine elements

Grid elements are different sizes based on breakpoints.

After exploring the responsiveness of the layout, we can apply the same approach to our existing components. Using the `Grid` elements to clean up some of our existing layout, we move the paper elements into our components, leaving us with just the layout elements in our page.tsx components (not counting the API calls and other needed functions). The following listing shows an example of the resulting page.tsx for the batch details screen.

Listing 8.51 The page.tsx for batch details

```
return (
   <Box sx={{ display: 'flex' }}>
      <CssBaseline />
      <StandardNavigation />
      <Grid container spacing={{ xs: 2, md: 3 }}
            columns={{ xs: 4, sm: 8, md: 12 }}
            sx={{mt: 8, pl: 1, pr: 1}}>
```

Defines a container

```
            <Grid xs={12} sm={6} key={1} >
               <AchTransactionCodesPieChart entries={entries} />
            </Grid> #B
            <Grid xs={12} sm={6} key={2} >
               <AchBatchStatistics entries={entries}/>
            </Grid>
            <Grid xs={12} key={3} >
               <Paper sx={{ minWidth: 1075 }}>
                  <Title>Batch Entries</Title>
                  <DataGrid columns={columns} rows={entries} />
               </Paper>
            </Grid>
         </Grid>
      </Box>
);
```

Places each of our components in a Grid element

Breakpoints can be used with other attributes as well. For instance, if we wanted to adjust the amount of `margin-top`, we could set it based on the screen size, as shown in the next listing. Notice how the margin is set to a smaller value of 4 for smaller screens and then to 12 for larger ones.

Listing 8.52 The `margin-top` based on screen size

```
mt: {
   xs: 4,
   sm: 4,
   md: 12,
   lg: 12,
   xl: 12
},
```

Breakpoints offer a powerful way to help with our layout and design. We can update the default values (pixels) for each breakpoint and even rename them, as shown in the following listing.

Listing 8.53 Customizing breakpoints

```
const theme = createTheme({
  breakpoints: {
    values: {
      mobile: 0,
      tablet: 640,
      laptop: 1024,
      desktop: 1200,
    },
  },
});
```

This chapter covered a lot, focusing on providing feedback when uploading a file that is not formatted properly. Exception handling provides a way to gracefully handle

unexpected situations within the software, giving us a chance to alert the user and recover as appropriate.

We started by looking at how we could expand our file-creation syntax to support adding these exceptions. Expanding the ability to produce files allows for easier and more robust file creation, which in turn gives us more files to load into our dashboard.

In terms of ACH handling and Pydantic, we focused on how defining our fields provided us with validation without the need for a lot of code. We were able to limit the field length, validate formatting with regular expressions, and even go back to a database to validate entries. We also added support for handling exceptions, which allowed us to get another taste of full stack development—by updating the backend, providing APIs to allow access to data, and finally providing UI elements to view the data.

Finally, we had the chance to expand our UI by adding new components, refactoring what was already being used, and looking at how to manipulate our layouts using breakpoints in MUI.

Summary

- After launching the MVP, user feedback highlighted both positive reception and areas for improvement, particularly the need for handling a wider variety of ACH-related exceptions.
- Planning enhancements involves prioritizing features based on user feedback and working on a timeline to implement ACH exceptions and search functionality.
- Exceptions in ACH processing can be categorized into errors, exceptions, and warnings, each with different severity and handling implications.
- Creating ACH files with exceptions involves extending BDD syntax to support file, batch, and entry-level exceptions for more robust testing and processing.
- Enhancing exception handling in ACH parsing includes using Pydantic for field validation and storing detailed exception information in the database.
- Exceptions API development focuses on enabling effective reporting and search capabilities, providing insights into file-specific errors.
- The UI is updated to display exceptions, incorporating components for viewing and filtering errors, with features such as DataGrid for interactive data presentation.
- Additional UI enhancements include creating reusable components, statistics for batches, and improving layout responsiveness using the MUI grid system.
- Utilizing unit testing and BDD helps ensure robustness and flexibility in handling ACH exceptions, while UI updates enhance UX and interaction with data.

9 Searching and auditing

This chapter covers

- Another iteration on our ACH dashboard
- Adding searching capabilities for ACH transactions
- Auditing of user actions

In previous chapters, we built the minimum viable product (MVP) and went through an iteration where we received feedback and used it to drive the next set of enhancements. This chapter continues using the same concepts. With the release of expanded support for exceptions and various UI/UX tweaks, we should expect feedback on those features and other functionality that users need.

Users would like to be able to search for transactions based on several criteria as they need to handle customer complaints about when a transaction was processed. In addition, other stakeholders have asked for the ability to audit interactions with the dashboard to gauge customer engagement and set up user tracking in the future.

9.1 Sprint planning

In the Agile world, the team needs to prepare for the sprint before starting any work. We already have an idea of what needs to be accomplished with this sprint but also have to define stories to know exactly what is expected in any given story and identify any acceptance criteria that may go along with it. We continue to follow the standard format for our stories: `As a [role], I [want to], [so that]`. For instance, to add search capabilities to our product, we may have something similar to `As a user, I want to be able to search for transactions, so that I can better research customer inquiries about their transactions`. Feel free to practice the phrasing for the additional user stories (even if it is just in your head) for the remaining tasks we cover in this chapter.

We take a moment to visualize the changes using another Gantt chart. Again, the chart presents a timeline of our changes, which can be shown to any stakeholders interested in the project. In our experience, this is one of the most effective means of communicating to stakeholders when the project is going to be complete. Often, stakeholders outside our team are not familiar with more Agile ways of tracking progress and just want to know when our code will be available. A simple Gantt chart gives them an idea of how the work should be progressing. The following listing shows the Gantt chart definition in PlantUML.

Listing 9.1 PlantUML Gantt chart definition

```
@startgantt
saturday are closed
sunday are closed

header ACH Dashboard Enhancements - Round 2

Project starts 2024-08-15
-- Searching Enhancement --
[Search API Design] starts 2024-08-15 and lasts 2 days
[UI Design/Changes] starts at \
            [Search API Design]'s end and lasts 4 days

-- Auditing Enhancement --
[Database Design] starts at 2024-08-15 and lasts 2 days
[Review APIs] starts at [Database Design]'s
➥ end and lasts 1 days
[Add Auditing logic] starts at [Review APIs]'s end
➥ and lasts 3 days
[UI Support for Auditing] starts at \
            [Add Auditing logic]'s end and lasts 3 days

[Sprint Complete] happens at 2024-08-28

@endgantt
```

Begins the Gantt chart definition

Defines days that should be skipped

Title for our chart

Defines timeline for searching enhancements; note that long lines can be split into multiple lines using the backslash character. Any trailing space after the backslash can cause parsing errors.

Defines timeline for auditing enhancements

Defines a milestone

Ends the Gantt chart definition

Figure 9.1 shows the generated Gantt chart. Not only does it provide a roadmap for where we are in the development process, but creating the chart helps us start thinking about the steps and requirements for any given enhancement.

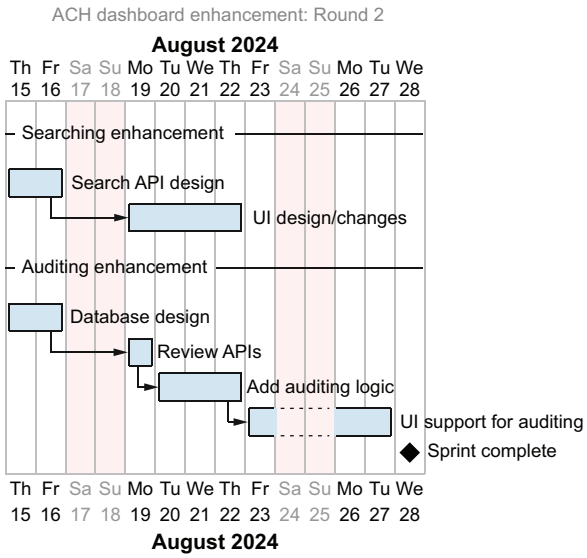


Figure 9.1 Gantt chart showing timeframes

We should now have an idea of the steps we need to take for the various enhancements and can get straight to work.

9.2 Searching ACH transactions

Which file contained that transaction? When was that transaction loaded? These are the types of questions we want to be able to answer with this enhancement. Throughout this book, we have focused on Prearranged Payment and Deposit (PPD) batches, which contain payments for payroll, utilities, and mortgage payments, among the rest. If a customer's utility payment is not processed correctly, and their water/electricity gets turned off, we can be sure we are going to hear about it. Of course, that is an extreme case. It is more likely that the customer has been alerted by a particular company that their payment was not received, and they have to pay late fees. Either way, the customer will not be too happy when they call.

We need to look at adding the ability to search our loaded files for both names and amounts. This feature will require changes to the UI as we will need to be able to enter search information and a new API to support the actual search and returning of results. Our immediate concern is the ability to find transactions within our limited dataset. In practice, ACH records should be kept for six years from the date of the receipt, and with billions of payments moving through the ACH network every year, chances are we will

have sizeable datasets to search. We need to evaluate other mechanisms for searching, in addition to our real-time method before this project is ready for production.

9.2.1 BDD testing for searching

Before putting together a feature to help validate the new (yet to be written) search API, we first must create a file we can use with our feature. We can use our existing file creation feature (ach_file_creation.feature) and add a new scenario to create a file for a specific user with a known amount. While we could search within one of the existing files, it is likely cleaner to keep the files associated with features separated so that our tests will remain more robust. The following listing shows creating a file for customer Sally Saver, with a single dollar transaction, which should be more than enough to get us started.

Listing 9.2 Creating a file for Sally Saver

```
Scenario: Create an ACH file with a single transaction
➥ for customer "Sally Saver"
  Given I want to create an ACH file named "sally_saver.ach"
  And I want to have an immediate destination of "990000013"
  And I want to have an immediate origin of "987654321"
  And I want to have 1 batch with ACH credits only and
➥ a standard entry class code of "PPD"
  And I want 1 entries per batch with random amounts between 100 and 100
  And I want to use individual names of "Sally Saver"
  And I want to have company name "My Company" and company id "1234567890"
  When my ACH is created
  Then I should have a file of the same name
  And there should be 1 batch in the file
  And there should be 1 entries in the file
```

With our file created, we can now work on setting up our BDD test to search for transactions. This should provide us with some insights into how we want the process to work in general. For instance, we know that we want to be able to search by at least the individual name and an amount. A couple of questions that should be up for discussion with various stakeholders in our project may be along the lines of

- Are we going to separate these searches out on the API side (e.g., allowing query parameters of `customer` and `amount`)? Or will we allow one search field and have the backend try to query the correct field?
- Are fields mutually exclusive? If we are searching for a customer, can we also search for an account?
- If we allow searching using multiple fields, will these be combined using AND or OR? Should that be determined by the user?

Asking those questions (and the answers to them) will help guide our design process. For now, we will take the approach where a user will be shown a single search box that will support any searches we want to perform. We will also put the responsibility

of deciding what the user wants to search for on the server side instead of on the UI. Another approach would be to have the UI split the fields and allow the API to accept different query parameters. With an approach to our search in mind, we can create a scenario to test our API search using a customer name.

Listing 9.3 Search by individual name

```
Scenario: Search by individual name
   Given that I have a clean database
   And that I have posted the file "sally_saver.ach"
   When I search for transactions with search criteria "Sally Saver"
   Then I should receive a list of "1"
➥ transaction with the amount of "1.00"
```

There was a need to add new syntax for the `When` and `Then` clauses to support our transaction search. The code is shown in the following listing.

Listing 9.4 Updates to test_file_api.py

```
…
scenarios("../features/search_ach_transactions.feature")
…
@when(parsers.parse('I search for transactions
➥ with search criteria \"{search_criteria}\"'))
def search_request_for_transactions(api_response,
➥ search_criteria):
    response =
 client.get(f"/api/v1/files/transactions/search?
➥ criteria={search_criteria}")
    assert response.status_code == 200, response.text
    api_response["response"] = response.json()
…
@then(parsers.parse('I should receive a list of
➥ \"{transaction_count}\" transaction with the amount of
➥ \"{transaction_amount}\"'))
def has_expected_transaction_and_amount(api_response, transaction_count,
transaction_amount):
    assert len(api_response["response"]) == 1, f"Expected
{transaction_count} transaction, but got
➥ {len(api_response['response'])}"
    response = api_response["response"][0]["amount"]
    assert transaction_amount == response,
➥ f"Expected {transaction_amount} in {response}"
```

Ensures the new scenario is loaded

Annotates the method with @when and parses the line

Defines a function to call our method and saves the response

Annotates the method with @then and parses the line

Defines a function to validate the results of the API response

Now that we have the supporting code behind the test, what should we expect to see if we were to run it? That's right—a `404 error` will be returned! That is to be expected because we have not yet defined the endpoint. In the next section, we will address this problem by adding the API.

9.2.2 Search API

At this point, we want to have an API that takes some input from the user and then searches the database for a transaction. As previously mentioned, we are only going to search for individual names and amounts. We expect the API to be called with a single query parameter named `criteria`. The API will determine whether the argument is applicable to an individual name or an amount.

This is not the only approach that we could have taken. We could also have split the search criteria into separate fields that would make the server side a bit easier but may have made searching more cumbersome from the user's perspective. Another method would be to blend specifying search criteria with keywords (e.g., allowing for searches such as `individual name = "user" and amount = $1.00`). Then we would have more parsing to do to get the fields we want to use. Another consideration is whether we want to use `GET` or `POST`. As mentioned, our example uses `GET`, and we pass a query parameter, which is what we will be searching for. This approach works, but the URL with the query parameters may appear on the web server, firewall, application, and third-party logs. If we enable searching for potentially sensitive information, such as account and tax identification numbers, we may want to reconsider this approach since we do not want such sensitive information showing in logs. There is no wrong answer—depending on our audience and what we expect to be searching, we may be told to go with a specific approach.

We first build a simple endpoint to work past the `404 HTTP` status code being currently returned. The following listing shows the initial implementation of the endpoint.

Listing 9.5 Initial endpoint definition for searches

```
@router.get(
    path="/transactions/search",
    response_model=list[str],
    summary="Retrieve ACH Transactions",
    description=
      "Retrieve the transactions matching
➥ the specified criteria.",
    response_description="ACH Transactions.",
    response_model_exclude_none=True,
    tags=["ACH Files"],
)
async def get_ach_transactions(criteria: str = Query(...,
  description="Search criteria for transactions"))
➥ -> list[str]:
    return [f"ACH Transactions for {criteria}"]
```

Defines a GET request with supporting documentation

Defines the method specifying the query parameter

Returns a hardcoded array entry containing the query parameter

While this code may get us past the `404 error`, it does not do much to help us create a useful endpoint. For that, we need to build both the SQL queries and response objects to return the data. We start by creating a `TransactionSearchResponse` class to hold our

data. The following listing shows the `TransactionSearchResponse` and the expected fields. If we find that we are missing something, we can always come back and add it.

Listing 9.6 The `TransactionSearchResponse` class

```
from decimal import Decimal
from typing import Annotated
from pydantic import BaseModel, Field, UUID4


class TransactionSearchResponse(BaseModel):
    file_id: UUID4 = Field(
        ..., description=
➥"Unique identifier for the ACH file.",
        title="File ID"
    )
    batch_header_id: UUID4 = Field(
        ...,
        description=
➥"Unique identifier for the ACH Batch Header.",
        title="Batch Header ID",
    )
    entry_id: UUID4 = Field(
        ...,
        description=
➥"Unique identifier for the ACH Entry Detail.",
        title="Entry Detail ID",
    )
    filename: str = Field(
        ..., description="The name of the ACH file.",
        title="Filename", max_length=255
    )
    individual_name: str = Field(
        ...,
        description=
➥"The name of the individual or company for the entry.",
        title="Individual Name",
        max_length=22,
    )
    amount: Annotated[
        Decimal,
        Field(
            ...,
            description="The amount of the entry.",
            title="Amount",
            max_digits=10,
            decimal_places=2,
        ),
    ]

    class Config:
        json_schema_extra = {
            "example": {
                "file_id":
```

Our standard import statements for Pydantic fields

Defines the class using the BaseModel from Pydantic as the base

Defines the various fields and constraints for our class

Provides an example of the response object for use with our documentation

```
➥"123e4567-e89b-12d3-a456-426614174000",
                "record_type_5_id":
 "123e4567-e89b-12d3-a456-426614174001",
                "record_type_6_id":
➥"123e4567-e89b-12d3-a456-426614174002",
                "filename": "test.ach",
                "individual_name": "John Doe",
                "amount": "100.00",
            }
        }
```

Provides an example of the response object for use with our documentation

Now that we have a place to store our response, we need to populate it. We do this by using a new class called `TransactionSearchSql`, which will store any code dealing with transaction search. We wanted to keep things simple with our first search attempt, so we handle the following three scenarios:

- Searching by a single amount
- Searching by an amount range
- Searching by an individual name

We created a method called `get_transactions`, as shown in the following listing. Here, we have two regular expressions to determine whether an amount was passed or a range of amounts was used. Otherwise, we default to using the name. For most people, the trickiest part of the code is likely to be the use of regular expressions.

Listing 9.7 The `get_transaction` method

A pattern to match a single amount

A pattern to match two amounts

Defines a function that returns a list of transactions

Checks the criteria against a single amount and calls the appropriate method

```
def get_transactions(self, criteria: str) ->
   list[TransactionSearchResponse]:
   amount_pattern = r"^\d+\.\d{2}$"
   multiple_amounts_pattern =
➥ r"(^\d+\.\d{2})\s+(\d+\.\d{2})$"
   if re.match(amount_pattern, criteria):
      return self._get_transactions_using_amount(criteria)
   elif match := re.match(multiple_amounts_pattern,
➥ criteria):
      begin_amount, end_amount = match.groups()
      return self._get_transactions_using_amount_range
➥(begin_amount,➥
 end_amount)
   else:
      return self._get_transactions_using_➥
➥individual_name(criteria)
```

Retrieves the amounts using the groups() method from match

Calls the method with the retrieved arguments

Uses the walrus operator to store the match results for the multiple_amounts_pattern

Otherwise, no amounts matched, and we should use the individual name.

Before moving on, let's break down the regular expressions to ensure you understand their usage. Here are the important points:

- Each pattern string is prefixed by `r` to denote a raw string, which prevents us from having to escape any strings. Otherwise, we would have to use `\\` instead of `\`.
- The characters `^` and `$` are input boundary assertions, indicating the start and end of the string, respectively. These help us eliminate matching strings that contain characters other than the pattern we are interested in.
- The `\d` matches any digit (0–9), and `+` is a quantifier that means to match one or more occurrences of the preceding atom (the most basic unit of a regular expression).
- The `\.` matches the literal period character because a single period will otherwise match any character.
- The `\d` matches any digit (0–9), and adding the `{2}` is another quantifier that specifies the count of atoms to match.
- The `\s` matches any whitespace. Again, we see the plus sign quantifier, which allows the two amounts to be separated by at least one character and up to any amount of whitespace.
- Wrapping a portion of the regular expression in parentheses creates a capture group. These groups can then be referenced later, as you'll see when we use `match.groups()` to extract the found values.

Regular expressions are not limited to the financial industry, and you will find them throughout your career. While they can be difficult to grasp, generative AI tools are a great way to help break down regular expressions and provide more insight into them. Practice makes perfect, and to get more hands-on experience with regular expressions, you may want to check out *Regular Expression Puzzles and AI Coding Assistants* (2023, Manning) by David Mertz.

With that behind us, let's take a look at the functions that are actually retrieving the transactions based on the search criteria provided. In our cases, the queries themselves remain much the same with only the `Where` clauses being unique for each query. The following listing shows the `_get_transactions_using_amount` function that looks for an exact amount and returns any results.

Listing 9.8 Retrieving transactions by amount

We declare that the row will be of type TransactionSearchResponse.

The standard columns that we are trying to select

```
def _get_transactions_using_amount(
    self, criteria: str
) -> list[TransactionSearchResponse]:
    with get_db_connection(
        row_factory=class_row(TransactionSearchResponse)
    ) as conn:
        sql = self.get_sql_selection_query()
```

```
sql += "WHERE aepd.amount = %s"
result = conn.execute(sql, [criteria])
return result.fetchall()
```

Appends the Where clause for the query

Uses the criteria that we want to pass to the query

Gets all rows and returns the response

We have factored out the common portion of the SQL query into the `get_sql_selection_query` method, which allows the `Where` clause to be added to the query. When searching for transactions by amount, we used an exact match. When using a range of amounts, we employ the Postgres syntax and use the keyword `BETWEEN`.

Listing 9.9 Using the `BETWEEN` keyword

```
sql += "WHERE aepd.amount BETWEEN %s AND %s"
```

Uses **BETWEEN** to search between two amounts; often more concise and readable than multiple conditions

Similarly, when searching for an individual name, we use the `ILIKE` keyword, as shown in listing 9.10. Using `ILIKE` allows us to do a case-insensitive search—otherwise, we are searching for the same string. Note that ILIKE is a Postgres-specific command. Other databases such as MySQL use the `LIKE` keyword, but it is case-insensitive by default. Still, other databases such as Oracle are case-sensitive and require the `UPPER` function in the comparison to achieve a case-insensitive search. So, always be aware of the database you are using.

We may consider appending a percentage sign (`%`) to the criteria, which is a SQL wildcard and would match the individual name if anything appeared after it. Or perhaps we could replace any spaces found in the string with the wildcard to expand our search results. Those are just some basic starting points if additional search capabilities are required.

Listing 9.10 Case-insensitive search

```
sql += "WHERE aepd.individual_name ILIKE %s"
```

Use **ILIKE** for a case-insensitive search of the individual name.

The following listing shows the common portion of the query that retrieves the transaction information. In addition to the need of joining records together to drill down to transaction details, we rename the fields to ensure they can be stored in our response object.

Listing 9.11 Common transaction query selection criteria

```
def get_sql_selection_query(self):
   return """
```

The selection criteria required for our object. The fields are prefixed to help determine the files they belong to. It is not always necessary when the names are unique but is helpful for maintaining the code.

```
        SELECT art1.ach_files_id AS file_id,
               art5.ach_records_type_5_id AS batch_header_id,
               art6.ach_records_type_6_id AS entry_id,
               af.file_name AS filename,
               aepd.individual_name AS individual_name,
               aepd.amount AS amount
          FROM ach_files af
    INNER JOIN ach_records_type_1 art1
➥ USING ( ach_files_id )
    INNER JOIN ach_records_type_5 art5
➥ USING ( ach_records_type_1_id )
    INNER JOIN ach_records_type_6 art6
➥ USING ( ach_records_type_5_id )
    INNER JOIN ach_entry_ppd_details aepd
➥ USING ( ach_records_type_6_id )
    """
```

Joining of the various records

With the queries working and returning our data, we should now see that all our tests have passed. Next, we look at how to add the search capability to our UI.

9.2.3 UI search page

To add search functionality to our page, we start by adding an icon to our sidebar navigation. This means we can update NavButtons.tsx to add a new button with an icon.

Listing 9.12 Updating NavButtons.tsx

Imports the search icon

```
…
import {CloudUpload, Logout, Error, Search} from "@mui/icons-material";
…
            <ListItemButton onClick={() => route.push("/search")}>
                <ListItemIcon>
                    <Search/>
                </ListItemIcon>
                <ListItemText primary="Search"/>
            </ListItemButton>
…
```

Includes the search icon

Of course, if we try clicking the button, we receive a `404 NOT FOUND` error message because we have yet to define our actual page. As with previous pages, we will create a simple page that will allow us to resolve the `404` error message and give us a starting point for developing the rest of the page. The following listing shows the src/app/search/page.tsx.

Listing 9.13 Search page stub

```
export default function SearchPage() {

    return (
        <ThemeProvider theme={defaultTheme}>
            <Box sx={{display: 'flex'}}>
                <CssBaseline/>
                <StandardNavigation/>
                <Box
                    component="main"
                    sx={{
                        backgroundColor: (theme) =>
                            theme.palette.mode === 'light'
                                ? theme.palette.grey[100]
                                : theme.palette.grey[900],
                        flexGrow: 1,
                        height: '100vh',
                        overflow: 'auto',
                    }}
                >
                    <Toolbar />
                    <Typography>Search Page</Typography>
                </Box>
            </Box>
        </ThemeProvider>
    );
}
```

Creates a simple search page

Something to identify the page so we know we navigated to it

With our page stubbed out, we can click the search button, and we should see our navigation and search page. Now, we can add the components that will give us the necessary functionality to create a search page capable of displaying our results. The hardest part is deciding how to layout the page. From previous API calls (such as returning exceptions in chapter 8), we already know how to fetch data from an API and display it within a DataGrid component. We will be following the same steps of defining an object to hold the data, a component to display the data, and then adding that to our search page.

We start by defining an interface `AchTransactionSearchResponse` to hold the data that the API response will pass back. The following listing shows the fields and their data types.

Listing 9.14 The `AchTransactionSearchResponse` interface

```
export interface AchTransactionSearchResponse {
    file_id: string;
    batch_header_id: string;
    entry_id: string;
    filename: string;
    individual_name: string;
    amount: string;
}
```

Exports the AchTransactionSearchResponse so that it is available everywhere

The fields returned from the API

Once we have a place to hold the retrieved data, we can create a component that will display the data. This `AchTransactionSearchResults` component will be similar to other components we have created, where we pass the results of our API call into the component via the results object of `AchTransactionSearchResultsProps`. We utilize the `renderCell` property on the column definitions to add links that allow users to jump directly to the file or batch of a particular search result. Furthermore, since our results did not have an ID column, we have to define one to the DataGrid component using the `getRowId` property. The `entry_id` can act as our ID column because for each result, the `entry_id` is the UUID for the transaction, and that is unique. The following listing shows the more important parts of the component.

Listing 9.15 The `AchTransactionSearchResults` component

```
interface AchTransactionSearchResultsProps {
    results: AchTransactionSearchResponse[];
}

export default function AchTransactionSearchResults
➥({results}:
Readonly<AchTransactionSearchResultsProps>) {

    const route = useRouter();

    const columns: GridColDef[] = [
        {field: 'filename', headerName: 'Filename',
➥ width: 150},
        …
        {field: 'viewFile', headerName: '',
                 sortable: false, width: 150,
                 renderCell: (params) => (
            <Link onClick={() =>
                route.push(`/fileDetails/${params.row.file_id}`)}
                sx={{ cursor: 'pointer' }}>Jump to file...</Link>
            )},
        {field: 'viewBatch', headerName: '',
                 sortable: false, width: 150,
                 renderCell: (params) => (
            <Link onClick={() =>
route.push(`/fileDetails/${params.row.file_id}/
➥batchDetails/${params.row.batch_header_id}`)}
              sx={{ cursor: 'pointer' }}>Jump to batch...</Link>
            )}
    ];

    return (
…
        <DataGrid columns={columns}
                  rows={results}
                  getRowId={(row) => row.entry_id}
        />
…
    );
}
```

Exports the search results function and declares the parameters read only

Uses the router for navigation purposes

Begins defining columns to show

Defines a column that will create a link, which will navigate to the file details page when clicked

Defines a column that will create a link, which will navigate to the batch details page when clicked

Since there is no specific parameter named id, we specify which field can be used.

Now that we have a place to store the data and a way to present them, all that is left is to make the API call and pass it to our component. For that, we updated our search page to have a text input field and a search button to make the actual API call. We use the `onChange` event of the `TextField` to store the text entered by the user into the `searchCriteria` field. When the user clicks the search button, the `onClick` event is fired, and the `handleSearch` is executed. The `handleSearch` makes a call to our API and stores the result. The important pieces of the updated search page are shown in the following listing.

Listing 9.16 Updated search page

```
…
export default function SearchPage() {
    const [searchCriteria, setSearchCriteria] =
➥ useState<string>('');
    const [results, setResults] =
useState<AchTransactionSearchResponse[]>([]);

    const handleChange = (event: { target: { value:
React.SetStateAction<string>; }; }) => {
        setSearchCriteria(event.target.value);
    };

    const handleSearch = async () => {
        …       axios.get<AchTransactionSearchResponse[]>
➥(`${apiUrl}/files/transactions/search?criteria=${searchCriteria}`, {
…
    return (
…
       <Box sx={{display: 'flex',
                 flexDirection: 'column',
                 alignItems: 'center', gap: 2}}>
          <TextField id="search"
                     label="Search"
                     variant="standard"
                     onChange={handleChange}
                     sx={{ width: '40%' }} />
          <Button variant="outlined"
                  color="primary" onClick={handleSearch}>
             Search
          </Button>
       </Box>
       <AchTransactionSearchResults results={results}/>
…
    );
}
```

Initializes the searchCriteria to an empty string

Initializes the results to an empty string

The handleChange function saves the entered text to be used later.

Uses axios to get the response

The handleSearch function calls the API and saves the results using setResults.

Defines a TextField element for a user to enter a search string

Defines a button element that will search for the entered string

A component that takes the results to display them

We now have a search page that can perform basic transaction searches and provides the ability to jump directly to the file or batch where the transactions were found. Next, we reinforce testing with Playwright to get some integration-level tests for our search capabilities.

9.2.4 Integration testing with Playwright

After we make searching for our transaction possible, it would be nice to add an integration test that we can use for integration and regression testing. We set up a test that requires us to

1. Upload a file to parse
2. Navigate to the search page
3. Populate the search page
4. Wait for an ACH response
5. Validate the result is populated
6. Take a screenshot

We created a Python script named test_search_page.py to perform all these necessary steps. In addition, we want to have the option to debug our test with the browser window displayed; to accomplish that, we need to create a few Pytest fixtures.

Listing 9.17 Pytest fixtures for Playwright

```
@pytest.fixture(scope="session")
def browser():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        yield browser
        browser.close()

@pytest.fixture(scope="function")
def page(browser):
    context = browser.new_context()
    pages = context.pages
    page = pages[0] if pages else context.new_page()
    yield page
    context.close()
```

Defines a fixture with the scope session, which will cause the fixture to persist for the length of the test session

Defines a function named browser that creates a browser instance. We use headless=False to step through the test session and see what the browser is doing. Using headless=True will hide the browser window and is the default way of running unit tests.

The yield browser command will return the browser object.

When all tests are finished, the browser is closed.

Creates a page that will be used with the test

Uses the default tab if it exists; otherwise, creates a new one

Returns the page using yield, so it is available to the test

Clears the context after the test is run

Defines a fixture with the scope of function, which is the default scope

That is enough to get the Playwright structure set up. Now, we want to ensure our database is empty and that we load our ACH file before starting our integration test.

Listing 9.18 Clearing the database and loading our file

```
@pytest.fixture(autouse=True)
def setup_teardown_method():
```

Define a fixture with autouse set to True. This causes it to be included for every test by default.

```
    SqlUtils.truncate_all()

    ach_file = "./data/sally_saver.ach"
    file_path = get_absolute_path("./data/sally_saver.ach")
    parser = AchFileProcessor()
    ach_files_id = SqlUtils.create_ach_file_record
➥(ach_file, str(randint(1, 99999999)))
    parser.parse(ach_files_id, file_path)
    yield

def get_absolute_path(relative_path):
    return current_file_dir / relative_path
```

Clears our database

Loads a file for the test into the database

Ensures the path is correct based on where we are running from

Now that the setup is done, we can start the actual testing with Playwright.

Listing 9.19 Playwright testing

```
def test_dashboard(page: Page):
    page.goto("http://localhost:3000/search")
    page.expect_navigation(wait_until="load")
    expect(page).to_have_url("http://localhost:3000/search")
    search_criteria = page.get_by_role("textbox")
    search_criteria.fill("sally saver")
    search_button = page.locator("#searchbtn")
    with page.expect_response("**/files/transactions/search*")
➥ as response_info:
        search_button.click()

    response = response_info.value
    assert response.status == 200
    search_result = page.locator('div[title="Sally Saver"]')
    expect(search_result).to_be_visible()
    page.screenshot(path="screenshots/search_results.png")
```

Finds the textbox and enters the text "sally saver"

Navigates to the search screen and ensures the page is loaded

Validates that we are on the correct page

Finds the search button

Checks that the response was 200

Takes a screenshot for posterity

We should be able to see an entry with the text "Sally Saver."

Expects an API response from the search button when it is clicked

Now you know how to perform some integration testing where you may need to make API calls and validate the results. In the next section, we look at another critical aspect of development in many industries: keeping an audit trail.

9.3 Auditing user interactions

One of the critical security controls put forth by The Center for Internet Security is that of audit log management (https://mng.bz/gaGE). While we won't be covering all the aspects, we will at least begin the journey by ensuring that we are collecting data about the usage of our application. Keep in mind that collecting data is the first step. If we do not have a strategy to monitor, review, and receive alerts as necessary, the logs are useless.

Many times, we have worked on a problem where we were put on the wrong path initially because someone had identified an error occurring in a log. Further research showed that the error had been occurring for many months (as far back as logs were readily available) and was not the source of the current problem. Sometimes, in the heat of a production incident, you can easily fall into that type of trap because of the need to resolve the problem quickly. With proper audit log management, you can gain a better understanding of the baseline application behavior and monitor for application misuse, among many other benefits.

While there are many commercial off-the-shelf tools (COTS) such as Splunk, Dynatrace, DataDog, and Sentry that help with logging and observability, we focus primarily on logging our data into the database and growing from there if necessary.

9.3.1 Database design

If we want to get started with auditing, we need to create a new database table to store our data. It will be based on the needs of your application and on what you are trying to accomplish. In our case, we are going to keep track of the API requests coming into the system. We introduce a new data type for dealing with hosts and networks—the `INET` data type for the `ip_address` field. While using a string data type for `ip_address` would certainly work as well, there are additional comparisons and functions in Postgres for dealing with `INET` that can make life easier. For instance, we could search for all addresses in a given range by using `WHERE ip_address << '192.168.1.0/24'`. This search may be helpful if we wanted to see activity coming from a list of addresses that we suspect of abusing our application. The following listing shows the creation of the table.

Listing 9.20 The `audit_log` table

Creates a table named audit_log

Saves when the record has been created using a TIMESTAMP datatype. We also use NOT NULL so that the field must always be populated. Finally, we default to using the NOW() function in Postgres, which will populate the field with the current date/time when records are inserted without a value.

```
CREATE TABLE audit_log (
    audit_log_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    user_id VARCHAR(25) DEFAULT NULL,
```

```
    ip_address INET DEFAULT NULL,
    user_agent VARCHAR(255) DEFAULT NULL,
    http_request VARCHAR(10) DEFAULT NULL,
    url VARCHAR(255) DEFAULT NULL,
    message TEXT NOT NULL
);
```

Creates a field named ip_address with the INET data type, which is a handy type for network addresses

With the table in place, we can create the code necessary to write these audit records out.

9.3.2 TDD testing for auditing

We create the unit tests for our API first and then add the required code to make our tests pass. Should we also consider the BDD-style testing? The answer to that depends on how the business will choose to use the audit log. If the log will serve primarily as something that developers use, then the business will likely not need to work with us to develop tests. In contrast, if the business plans to use the audit log to pull the activity to help determine metrics, we may need to work with them to develop tests. We could easily see the business making that type of request and the need to update our tests to read something along the lines of `WHEN I REQUEST A LIST OF FILES THEN I SHOULD SEE AN AUDIT LOG RECORD INDICATING FILES WERE SHOWN`. For now, we will stick with our standard unit testing. The unit test defines a record and then retrieves it so that we can compare the fields. We currently have a field for `user_id`, but since we are not tracking users at the moment, we will leave it empty.

Listing 9.21 Unit test for the audit log

```
class TestAuditLog:
    @pytest.fixture(autouse=True)
    def setup_teardown_method(self):
        SqlUtils.truncate_all()
        yield

    def test_insert_record(self):
        log_record = AuditLogRecord(
            ip_address="192.168.1.0/24",
            user_agent="Mozilla/5.0",
            http_request="GET",
            http_response=200,
            url="/api/v1/files",
            message="Test audit log insert",
        )
        logger = AuditLog()
        audit_record_id = logger.log_record(log_record)
        retrieved_record =
➥logger.get_log_record(audit_record_id)
        excluded_fields = {"audit_log_id", "created_at"}
        assert SqlUtils.get_row_count_of_1("audit_log") is True
➥, "Expected 1 record"
        assert retrieved_record.dict(exclude=excluded_fields)
➥ == #D
```

Clears the database each time

Creates a dummy record

Uses the AuditLog class to create and retrieve a record

Excludes the specified fields from the dictionary as these are populated by the database and do not affect the comparison; then asserts the record match

```
    log_record.dict(
           exclude=excluded_fields
       ), f"Expected {log_record},
➥ but got {retrieved_record}"
```

The unit test requires an object for `AuditLogRecord`, which is a standard Pydantic model to hold our data. We use two new data types in this model—`IPvAnyNetwork` and `AnyUrl` (both available as imports from Pydantic). The `IPvAnyNetwork` works well for representing the `INET` field we defined in the table and provides validation as expected for IP addresses, regardless of whether they are defined as a single address (e.g., `127.0.0.1`) or a range of them (e.g., `192.168.1.0/24`). Of course, we would not expect to see a range of addresses in this field for an audit record. The `AnyUrl` field similarly provides validation of URLs, saving us from having to write regular expressions to validate the field. Listing 9.22 shows our `AuditLogRecord` class, with some less interesting fields removed for brevity.

Listing 9.22 The `AuditLogRecord` class

```
class AuditLogRecord(BaseModel):
…
    ip_address: Optional[IPvAnyNetwork] = Field(
        None,
        title="IP Address",
        description="IP address from which the request originated.",
    )
…
    url: Optional[AnyUrl] = Field(
        None,
        title="URL",
        description="URL accessed by the request.", max_length=255
    )
…
    @field_validator("url", mode="before")
    @classmethod
    def convert_url_to_str(cls, v: AnyUrl) -> str:
        return str(v)
```

Defines the ip_address as a type of IPvAnyNetwork, which allows Pydantic to validate IP addresses and networks

Defines the URL as a type of AnyUrl, which allows Pydantic to validate whether the URL is properly formatted

Defines a field validator so that the URL is converted to a string to be stored in the database

As shown in listing 9.22, we also use a field validator to return the URL field as a string. This is because we will receive an error when attempting to insert it into the database because the URL datatype is not compatible with a string. We could have explicitly coded for it in the audit log class responsible for inserting the record, but this approach wraps up everything nicely, and other developers do not have to worry about it, which helps reduce cognitive load.

The final step of being able to write records and get this unit test to pass is generating an actual class to be inserted into the database. In the following listing, we create an `AuditLog` class that contains a `log_record` and `get_log_record` methods. For other database records, we specifically used `Sql` as part of the name. In this case, we did not necessarily want to tie it to a database.

Listing 9.23 The `AuditLog` class

```
class AuditLog:
    @staticmethod
    def log_record(log_record: AuditLogRecord):
        with get_db_connection(row_factory=dict_row) as conn:
            log_record_dict = log_record.dict()
            log_record_dict["url"] = str(log_record_dict["url"])
            result = conn.execute(
                """
            INSERT INTO audit_log (user_id, ip_address,
                        user_agent, http_request, http_response,
                        url, message)
            VALUES (%(user_id)s,
                 %(ip_address)s,
                 %(user_agent)s,
                 %(http_request)s,
                 %(http_response)s,
                 %(url)s,
                 %(message)s
                 )
            RETURNING audit_log_id
             """,
                 log_record_dict,
             )

        return result.fetchone()["audit_log_id"]

    @staticmethod
    def get_log_record(audit_log_id: str):
        with get_db_connection(row_factory=
➥class_row(AuditLogRecord)) as conn:
            result = conn.execute(
                """
                SELECT *
                FROM audit_log
                WHERE audit_log_id = %(audit_log_id)s
                """,
                {"audit_log_id": audit_log_id},
            )

        return result.fetchone()
```

Defines a dictonary to use

Defines the fields to insert; we will return the audit_log_id we obtained from the database.

Because we used a dict_row for row_factory, we can reference the audit_log_id directly.

With that in place, our unit test is passing, and we can add audit records wherever we want. Now, let's see how to integrate what we know with the FastAPI framework.

9.3.3 Auditing logic

The most obvious way to add auditing to our application is by making a call to the `log_record` method and passing an `AuditLogRecord`, as seen from our unit test. We could certainly take one of our existing endpoints and add the call into it, ending up with something akin to the following listing, which allows us to write a message but does not capture some of the other potentially useful information we defined fields for.

Listing 9.24 One way to add logging of our API

```
async def read_files() -> list[AchFilesResponse]:
    AuditLog().log_record(
        AuditLogRecord(
            message="User retrieved ACH files -test"
        )
    )
    return AchFileSql().get_files_response()
```

Manually calls the log_record method whenever the API is executed, but we are missing many of the interesting fields

We could enhance the previous listing by including the request field. Then we can pull the additional information from the request object.

Listing 9.25 Including request information in our API

```
async def read_files(request: Request) ->
➥ list[AchFilesResponse]:
    AuditLog().log_record(
        AuditLogRecord(
            ip_address=request.client.host,
            message="User retrieved ACH files -test"
        )
    )
    return AchFileSql().get_files_response()
```

Updates the method to include the Request object

We now have access to some of the interesting fields

While that works, it requires the developers to add more boilerplate logic than we would prefer. Since we are primarily interested in auditing the usage of our APIs, we can take another approach.

9.4 Using middleware for our logging

With the goal of logging the API requests, we can look at adding a middleware function that has the sole purpose of logging our requests. We can employ both the request and response, which allows us to capture the response code as well. While we could also look at configuring our Nginx logs to capture the information in the production environment, seeing how to accomplish this in FastAPI is also helpful since it can be widely applied for other requirements. We use the `@app.middleware` notation to write our log record.

Listing 9.26 The middleware layer for logging

```
@app.middleware("http")
async def log_requests(request: Request, call_next):
    response = await call_next(request)
    log_message = getattr(request.state, 'log_message',
                  "Default audit log message")
```

Defines a middleware component that executes for every HTTP request

Passes the request on to the next layer

Defines a method to log requests

Gets the log_message that was defined in the request.state and defaults if not present

```
    log_record = AuditLogRecord(
        ip_address=request.client.host,
        user_agent=request.headers.get
➥('user-agent', 'unknown'),
        http_request=request.method,
        http_response=response.status_code,
        url=str(request.url),
        message=log_message,
    )
    logger = AuditLog()
    logger.log_record(log_record)
    return response
```

Builds an AuditLogRecord with our desired request fields

Writes the log messages

Returns the response

It is relatively easy to insert that middleware layer and have it perform the logging. Then we can log all the calls that are coming in, which removes the need for the boilerplate code. Moreover, it keeps our code cleaner and lets the developers focus on their code. The only drawback is that adding a message would take a little more effort. Of course, the message is going to reflect what the `REST API` call is doing. So, in some respect, it is redundant (and a waste of space) to log a message. For instance, do we need to know someone viewed ACH files when the log also shows that they were performing a `GET` on `/api/v1/files`? Of course, we may need to do it for business reasons because it is easier for people unfamiliar with the data to interpret the message when being reviewed. Rather than waste the storage, we may decide to have a utility method that could perform that interpretation and insert the appropriate message during an extract. The use cases could vary, of course, and we only want to illustrate some of the alternatives. In this case, we do want to store the text in the database table, and we are going to accomplish that using a decorator. The following listing shows how to include a message in the request state using a `log_message` annotation we defined.

Listing 9.27 The `log_message` annotation

```
def log_message(message: str):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            request: Request = kwargs.get("request", None)
            if request:
                request.state.log_message = message
            return await func(*args, **kwargs)

        return wrapper

    return decorator
```

If we have a request, it saves the message that was passed.

Finally, we only need to make some minor changes to the API. By using our new annotation, we can customize the message written to the database. The next listing shows the final changes to our API call.

Listing 9.28 Final changes for logging

```
@log_message("Retrieving ACH files")
async def read_files(request: Request) ->
➥ list[AchFilesResponse]:
    return AchFileSql().get_files_response()
```

Annotates each of our API calls with a unique message

Ensures the request object is part of the method

Adding that little bit of logic completes the logging of our API messages. Now, log messages are written out for any APIs we annotate with `@log_message`. This is just the beginning of tracing and auditing your application. You can vastly expand collecting data in your application by exploring projects such as OpenTelemetry, OpenTracing, or Prometheus, to name just a few.

9.5 Viewing the audit log

Now that we have the application logging API requests, let's add the ability to view the log in our application. This page will function in much the same way the exceptions page from chapter 8 did. We need to retrieve the audit records and place them in a DataGrid to be viewed. Of course, we can expand the functionality as needed, but for now, simply returning the data is all that we need to accomplish.

9.5.1 Creating the page

We create the page.tsx under `src/app/audit`. This page will be responsible for making a request to the API and then passing the returned data down to a component that can display the records. The following listing shows the basic page.

Listing 9.29 Audit page

```
export default function AuditPage() {

    const [entries, setEntries] = useState<AuditResponse[]>([]);

    useEffect(() => {
        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        axios.get<AuditResponse[]>(`${apiUrl}/audits`, {
            headers: {
                'Content-Type': 'application/json'
            }
        })
            .then(response => {
                console.log(`Response data ${JSON.stringify(response.
 data)}`);
                setEntries(response.data);
            })
            .catch(error => {
                console.log(error);
            });
    }, []);
```

Gets the audit records from the database

```
    return (
…
       <AuditRecords records={entries}/>      ← Displays the audit records
…
    );
}
```

To finish this page, we also want to create the `AuditResponse` interface, as shown in listing 9.30. We keep the names aligned with those from the API because there is less work for us. However, in previous chapters, we have shown the process of mapping fields to internal representations of the data, and that is still valid here. Keep in mind the tradeoffs we incur by exactly matching the API—the biggest problem being when the API changes or when we move to another vendor where the field names may change. These changes must be propagated throughout the application instead of at a single point. We recommend always trying to use a layer of abstraction to prevent these types of problems. Nobody wants to spend their time changing `created_at` to `created_on` and having to retest everywhere it may touch.

Listing 9.30 The `AuditResponse` interface

```
export interface AuditResponse {
    audit_log_id: string;
    created_at: Date;
    user_id: string;
    ip_address: string;
    user_agent: string;
    http_request: string;
    http_response: number;
    url: string;
    message: string;
}
```

9.5.2 *Creating the page components*

We pass the data down to our `AuditLogRecords` component so that the presentation logic is taken care of. At this point, we should hopefully see how components help break our code apart into smaller, more manageable pieces. We try to keep the pages responsible for retrieving the data, while components on the page are responsible for presenting that data. We tend to have more duplicated code when applications are first being developed. This is often due to developers trying to rush to meet deadlines. As the code matures (and the developers get a better sense of the application), we can often start to create more generalized code. Breaking down our code into components helps us get to that point quicker. The next listing shows the code for the presentation of an audit record.

Listing 9.31 The `AuditLogRecords` component

```
interface AuditRecordsProps {
    records: AuditResponse[];
```

```
}

export default function AuditRecords({records}: Readonly<AuditRecordsProps>)
     {
    const [isOpen, setIsOpen] = useState<boolean>(false);
    const [auditDetails, setAuditDetails] =
                    useState<AuditResponse | null>(null);
    const columns: GridColDef[] = [
        {field: 'view', headerName: 'View',
➥ sortable: false, width: 10, renderCell: (params) => (
                <IconButton
                    onClick={(e) => {
                        e.preventDefault();
                        setAuditDetails(params.row);
                        setIsOpen(true);
                    }}
                    color="primary"
                >
                    <InfoIcon />
                </IconButton>
            )},
        {field: 'created_at', headerName: 'Date',
➥ width: 150, valueGetter: (params) =>
convertDateFormat(params.value)},
➥         {field: 'ip_address',
➥headerName: 'IP Address', width: 150, valueGetter: (params) =>
➥ stripSubnet(params.value)},
        {field: 'message', headerName: 'Audit Message', width: 300},
    ];

    return (
        <ThemeProvider theme={defaultTheme}>
            <Box sx={{display: 'flex'}}>
                <CssBaseline/>
                    <Container maxWidth="lg" sx={{mt: 4, mb: 4, ml: 4}}>
                        <Paper
                            sx={{
                                p: 2,
                                display: 'flex',
                                flexDirection: 'column',
                                height: 480,
                                width: '100%'
                            }}
                        >
                            <DataGrid columns={columns}
➥ rows={records} getRowId={(row) => row.audit_log_id} />
                        </Paper>
                    </Container>
            </Box>
            <AuditModal open={isOpen}
➥] onClose={setIsOpen.bind({}, false)} auditData={auditDetails} />
        </ThemeProvider>
    );
}
```

Defines a field to keep track of the modal

Defines the contents of the audit details for the modal

When the button is clicked, passes information to the modal and sets that it is now open

Uses the DataGrid element to display the text

Includes the AuditModal component to display the detailed contents

There is an extra utility method we have also built as part of the component—the `stripSubnet` method. The code for this method is shown in the following listing. Note it only removes a trailing `/32`, so why do we have a function to remove it, and why exactly is it there?

Listing 9.32 The `stripSubnet` utility method

```
export function stripSubnet(ipWithSubnet: string): string {
  const [ip, subnet] = ipWithSubnet.split('/');

  if (subnet === '32' ) {
    return ip;
  } else {
    return ipWithSubnet;
  }
}
```

Splits the IP address on the slash

If the subnet is 32, that indicates a single IP address.

Returns the IP only when it is a single IP address

Otherwise, returns the string as it came through

The `/32` is part of the Classless Inter-Domain Routing (CIDR) notation, which is a way to define IP address ranges. The `/32` means that the given address is a single address, and it is just an explicit way to indicate that. To illustrate, we could represent the addresses between `192.168.1.0` and `192.168.1.255` with `192.168.1.0/24`. That explains what the `/32` is but not why it is there. We are seeing the `/32` because we chose to represent the IP address in our Pydantic model with `IPvAnyNetwork`.

Pydantic supports many network types. The ones that we are primarily concerned with at this point are `IPvAnyAddress` and `IPvAnyNetwork`. The decision to use `INET` for our `ip_address` column was the catalyst for the problem. The `INET` datatype supports storing IP addresses using the CIDR notation. For that reason, we chose to use a datatype in Pydantic that closely modeled that functionality, even though we should only have a single IP address in our table. We used this example to illustrate how design decisions can have a downstream effect. The `INET` requirement for the column may be set because there is a need to be able to query for ranges of IP addresses, and having an `INET` data type makes sense even though there is no business need for the audit record to contain more than a single IP address. Perhaps, the decisions were already made (for better or worse) by the time the UI piece lands on our plate. The business does not want users to see the `/32` as part of the IP address because most users will not understand the notation, so we need to correct the display.

The following listing shows some additional unit tests that help illustrate the difference between the Pydantic types of `IPvAnyAddress` and `IPvAnyNetwork`.

Listing 9.33 Unit testing of the network types

```
class IpModel(BaseModel):
    ip_any_address: IPvAnyAddress
    ip_any_network: IPvAnyNetwork
```

```
class TestPydanticNetworkTypes:

    def test_valid_any_address(self):
        my_address = IpModel(ip_any_address="127.0.0.1",
➥ ip_any_network="127.0.0.1")
        assert str(my_address.ip_any_address) == "127.0.0.1"

    def test_invalid_any_address(self):
        with pytest.raises(ValueError):
            IpModel(ip_any_address="127.0.0.256",
➥ ip_any_network="127.0.0.1")

    def test_valid_any_network(self):
        my_address = IpModel(ip_any_address="127.0.0.1",
➥ ip_any_network="127.0.0.1")
        assert str(my_address.ip_any_network) == "127.0.0.1/32"

    def test_invalid_any_network(self):
        with pytest.raises(ValueError):
            IpModel(ip_any_address="127.0.0.1",
➥ ip_any_network="127.0.0.256")
```

Another custom component we make use of is `AuditModal`. When the details of an audit record are clicked, we want to display a formatted window with the audit information (listing 9.34). This is meant to be a more complete view of the audit record, so we do not need to clutter up the initial view. `AuditModal` is a relatively straightforward component. As we already have the entire record returned, this is just a presentation of the data, as opposed to the previous chapter where we specifically retrieved the ACH record data when the user clicked the view icon. Remember, the difference was that we did not want to return possible NPI data unless the user requested it. Now we have the ability to view audit information, monitor for users requesting NPI data, and address any potential misuse (whether intentional or not) of the data.

Listing 9.34 The `AuditRecords` component

```
interface AuditRecordsProps {
    records: AuditResponse[];
}

export default function AuditRecords({records}:
➥ Readonly<AuditRecordsProps>) {

    const [isOpen, setIsOpen] = useState<boolean>(false);
    const [auditDetails, setAuditDetails] =
➥ useState<AuditResponse | null>(null);
    const columns: GridColDef[] = [
        {field: 'view', headerName: 'View', sortable: false,
➥ width: 10, renderCell: (params) => (
                <IconButton
                    onClick={(e) => {
                        e.preventDefault();
                        setAuditDetails(params.row);
```

Clickable button that displays detailed information on the audit record

```
                        setIsOpen(true);
                    }}
                    color="primary"
                >
                    <InfoIcon />
                </IconButton>
            )},
        {field: 'created_at', headerName: 'Date', width: 150,
➥ valueGetter: (params) => convertDateFormat(params.value)},
➥       {field: 'ip_address', headerName: 'IP Address',
➥ width: 150, valueGetter: (params) => stripSubnet(params.value)},
        {field: 'message', headerName: 'Audit Message', width: 300},
    ];

    return (
        <ThemeProvider theme={defaultTheme}>
            <Box sx={{display: 'flex'}}>
                <CssBaseline/>
                    <Container maxWidth="lg" sx={{mt: 4, mb: 4, ml: 4}}>
                        <Paper
                            sx={{
                                p: 2,
                                display: 'flex',
                                flexDirection: 'column',
                                height: 480,
                                width: '100%'
                            }}
                        >
                            <DataGrid columns={columns}
➥ rows={records} getRowId={(row) => row.audit_log_id} />
                        </Paper>
                    </Container>
            </Box>
            <AuditModal open={isOpen} onClose=
➥{setIsOpen.bind({}, false)} auditData={auditDetails} />
        </ThemeProvider>
    );
}
```

Clickable button that displays detailed information on the audit record

DataGrid element that displays all the audit record results

AuditModal component that displays when the icon is clicked

With the ability to pull and view audit data, we are missing one critical piece to our puzzle—let's add the API endpoint to retrieve our data.

9.5.3 Adding the APIs for the audit log

While we built the initial SQL calls that retrieved records and the UI layers to display the data, we have not built the API itself yet. We now need to ensure we can call the API and return the data. The following listing shows that all it takes is a few lines to accomplish this task, and most of those are helping document the API for others to use. We created the audit API in its own file named routers/audit.py. Because this endpoint has nothing to do with ACH files, we wanted to separate it.

Listing 9.35 Audit router

```
@router.get(
    path="",
    response_model=list[AuditLogRecord],
    summary="Retrieve Audit Information",
    description="Retrieve audit records for the ACH API.",
    response_description="The overview of the requested audit records.",
    tags=["Audit"],
)
@log_message("Audit Information")
async def read_audit_information(request: Request) -> list[AuditLogRecord]:
    return AuditLog().get_all_log_records()
```

Simple call to get and return all audit records

Since we separated our code for easier maintenance, the API will not be reachable until we add it using an `include_router` method. We had initially done this for the router/files.py file, and all our APIs up until this point have been put into that file, meaning there has been no need to revisit the main.py file until now. The next listing shows the additional line needed to access our API.

Listing 9.36 Adding the router

```
app = FastAPI()

app.include_router(files.router)
app.include_router(audit.router)

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,  # List of origins that should be allowed
    allow_credentials=False,  # Allows cookies to be included
    allow_methods=["*"],  # List of HTTP methods allowed for CORS
    allow_headers=["*"],  # List of HTTP headers allowed for CORS
)
```

The additional router to handle the audit API requests

That was the final piece to put into place. Now we should have fully functioning audit capabilities. We can expand on this with additional logging and more specific logging of business scenarios, such as invalid input and other scenarios where we may want to log more than just confirmation that an API was accessed.

Summary

- Each successive iteration helps improve the functionality of our application. We looked to address requests/concerns brought up by the end-users, new functionality required by the business, and technical debt.
- Being able to search is a necessary feature of our application. Transactions are loaded at various times through different files throughout the day. If a customer

has a question about when a transaction was loaded, we need to be able to provide an answer quickly.

- Whether we are working toward protecting our application from intruders or disgruntled employees to gain deeper insight into the application, it all starts with proper auditing for security and compliance, accompanied by general analysis.
- We touched on the importance of not just logging but also of how having a successful strategy for managing logs will play a part in our success. We should not trust that the application is working or not being attacked—instead, we must have the data and analyze it.
- You were shown the importance of expanding exception handling and enhancing UI/UX based on user feedback to improve overall user satisfaction and system functionality.
- The significance of enabling transaction search by multiple criteria was highlighted in addressing customer complaints effectively and ensuring better UX.
- Stakeholder requests for audit capabilities emphasized the value of tracking dashboard interactions for understanding user engagement and planning future system improvements.
- We focused on the importance of Agile sprint planning, including defining clear user stories and acceptance criteria to ensure alignment and successful delivery of sprint goals.
- Gantt charts are effective in visualizing project timelines, aiding in stakeholder communication and expectation management.
- The requirement for robust search capabilities in large datasets underscores the need for an efficient search API, which prompted us to explore UI changes and real-time search methods.
- Developing BDD tests illustrated the critical role of testing in validating new search functionalities, ensuring they meet specified scenarios and user needs.
- Implementing a search API involved understanding various approaches for criteria handling, highlighting the importance of flexible and user-friendly search functionalities.
- The exploration of logging and auditing emphasized the significance of security practices in monitoring API usage and user interactions to ensure safe and accountable system behavior.
- Designing a database for audit logs reinforced the need to capture detailed request information such as user agent and IP address for comprehensive security and analysis.
- Using middleware in FastAPI demonstrated how to streamline logging processes, which reduces redundant code and enhances code maintainability.
- Integrating an audit log view into the application showed the value of transparency and monitoring of system activities to support ongoing improvement and accountability.

10 Company information

This chapter covers

- Adding functionality to support a company dashboard
- Adding a Company Information component
- Adding a Recent ACH Batches component
- Adding an Expected Files component
- Adding a Company Limits component

This chapter looks at another iteration during our program increment. The business had previously mapped out the additional functionality regarding the storing and displaying of company information. Basically, the business is now requesting a dashboard at the company level. In addition, users have provided some input on additional functionality they would like to see implemented, which was not in the original scope of the project, and the business has asked us to include it as well.

10.1 Sprint planning

In this sprint, we create a dashboard that focuses on data at the company level. In addition to tables and APIs, there are four additional components that our team must create during this sprint:

- Company information
- Recent batches
- Expected files
- Company limits

The company information tables are identified as blockers in our sprint because they must be designed and built before working on the components. It may be possible to start with some other components with little to no rework, as many of the tables for enhancements—such as the Expected Files component—only have a foreign key requirement back to the company ID. In such situations, it may be okay to start working without the foreign key, but other changes such as displaying customer information will require tables built to a certain extent. Figure 10.1 shows the Gantt chart created using PlantUML.

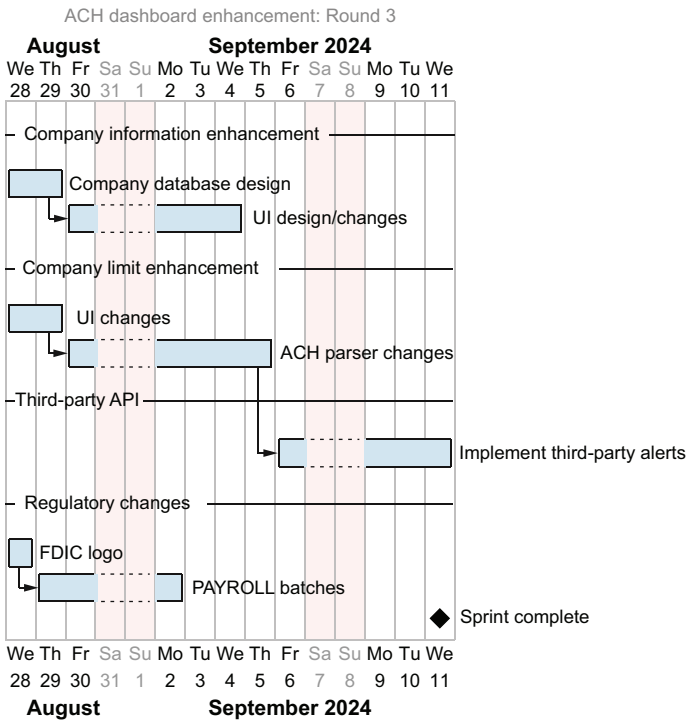


Figure 10.1 ACH dashboard enhancement timeline

10.2 Company Information

As stated previously, this sprint is all about working with companies. Each ACH file consists of batches with various entry and addenda records. These batches are associated with companies. If the companies are customers of our financial institution, we have an opportunity to provide various reports and controls around them. It is important to understand that the batches are not necessarily from companies associated with our institution—they may be from another company or partners we work with to deliver transactions to the institution. The company dashboard and components we build for it will provide additional ACH-processing services to our customers. While these components can be indispensable to the company as they provide the potential for additional auditing, alerts, and controls, they are not part of the ACH standard.

We have to create database tables and APIs to support storing and accessing the company information in our database. The companies are business customers of our financial institution, and as such, we want to provide them with additional information and controls of ACH files that are beyond what may be typically needed for a personal customer. For instance, processing ACH batches for our personal customers usually results in posting paychecks and other prearranged payments or debits. From an individual's perspective, there is no interest in the batch information and statistics except that the resulting transaction within the batch posts to their account correctly.

If we contrast that to business customers (companies), they may very well be interested in the batch information because they will be able to tell the totals for all transactions within the batch. Similar to personal customers managing their account balances for individual transactions, companies need to be aware of all the transactions within the batch because they affect the company.

While we may receive and process ACH batches from companies that are not customers of our financial institution, in cases where they are, we want to provide additional details about the batch. By creating database tables for these companies, we also open the possibility to originate files in addition to receiving files to process, as we have been doing throughout the project. Originating ACH files is beyond the scope of the current development, but it is something that we could look at to enhance our project as a logical next step in building a fully functional ACH system. We will be populating our company tables with dummy data to give ourselves something to work with throughout the chapter.

10.2.1 Database design: Company information

In addition to the `audit_log` in chapter 9 (listing 9.20) and the `bank_routing_numbers` table in chapter 8, we had to add many tables that were not specifically ACH related. With supporting company information, we now get to add a few more tables and supporting types. First, it makes sense to start with the company table and see what we need to keep track of.

Listing 10.1 introduces a couple of custom types into our database. For both the `tax_id_type` and `industry` fields, we created new custom types to provide some validation

and assist with data entry. These custom types help enforce the values of data we expect to be present in the field, which prevents entering erroneous data.

The `tax_id_type` and `tax_id_number` are correalted because in the United States, there is more than just the Social Security Number (SSN) that individuals are assigned, such as Employer Identification Number (EIN) and Individual Tax Identification Number (ITIN) just to name a few. So, we give the field a more general name instead of simply calling it `ssn` or `ein`. We default to `EIN` because companies are more likely to use an EIN number.

The `industry` is used for reporting purposes so that various reporting and marketing campaigns can filter out industries when users wish to drill down or create reports based on industries. The `logo` field is used to store an image for the company so that the image can be loaded onto the company page as necessary and only serves to make the pages and reporting a bit nicer. We use a new type `BYTEA`, which can store binary data. While we would benefit from having the logo stored outside the database in terms of UI performance, being able to keep things within the database prevents us from having to introduce other technologies such as AWS S3 or a CDN.

Listing 10.1 shows the `CREATE TABLE` statement. The `ach_company_id` is used to identify the company within ACH. This field corresponds to the company identification field on the batch header record (type 5) in an ACH system. While it is common to use the EIN number or some variation of it, using it is not obligatory. We only need to keep in mind that the field is 10 positions long and should be padded with leading zeros when the file is created to maintain file formatting.

Listing 10.1 Table to store company details

```
CREATE TABLE company (
    company_id UUID PRIMARY KEY
                    DEFAULT uuid_generate_v4(),
    name VARCHAR(255) NOT NULL,
    tax_id_type TIN_TYPE NOT NULL DEFAULT 'EIN',
    tax_id_number VARCHAR(9) NOT NULL,
    ach_company_id NUMERIC(10) NOT NULL,
    duns NUMERIC(9) DEFAULT NULL,
    logo BYTEA DEFAULT NULL,
    website VARCHAR(255) DEFAULT NULL,
    industry INDUSTRY_TYPE DEFAULT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);
```

Defines the tax_id_type with a default value of EIN since, most likely, this will be the type of value used

The company identification number used on the ACH type 5 record (batch header)

Many companies have a DUNS number, required for many services when registering as a company.

We store the logo as BYTEA so that we can upload an image to the database. Typically, this is a URL for the location where the image is stored. Storing it in our database was only for convenience.

The custom types in the previous listing need to be defined before creating the table. Creating the types is a straightforward process, and having these custom types helps us with an additional level of validation for the database and fields. The approach of course can vary, depending on our application's needs. For instance, it would also be acceptable to store these as strings in the database, or perhaps as a numeric value, while having the labels stored in another location.

Depending on several factors (company policy, team members, application, database, etc.) it may be preferable to store the values in another table rather than defining a type in the database to avoid database change requests. Since we have limited values, and there is no need for them to change very often, we are comfortable with defining them as types. The following listing shows creating of our `industry_type` and `tin_type`.

Listing 10.2 The `industry_type` and `tin_type`

```
CREATE TYPE industry_type AS ENUM ('basic materials', …, 'utilities');

CREATE TYPE tin_type AS ENUM ('SSN', 'EIN', 'ITIN', 'ATIN', 'PTIN');
```

As shown, creating a type to store values is simple. Another concern we may have—which is also a reason why team members may argue for using separate tables—is the ability to extract the list of valid values (e.g., if we wanted to build a dynamic drop-down of the valid values for the `tin_type` on the UI side). The following listing shows how to extract the labels for an `ENUM` type.

Listing 10.3 Returning the labels for an `ENUM` in Postgres

```
SELECT e.enumlabel AS label
  FROM pg_type AS t
  JOIN pg_enum AS e
    ON t.oid = e.enumtypid
 WHERE t.typname = 'tin_type';
```

Returns the label for our ENUM

Postgres tables needed to return the ENUM label

The specific ENUM type we are interested in

Once we have our company table in place, we can create an address table. It will show two addresses—the mailing and a street address—which we define with `address_type`. The rest of the table is standard in terms of what we need to store. We allow four lines for an address to cover apartment/floor numbers or additional information, plus other standard fields such as city, state, and zip.

Listing 10.4 The `address_type` and table

```
CREATE TYPE address_type AS ENUM ('mailing', 'street');

CREATE TABLE company_address (
```

Defines possible mailing address types

```
    company_address_id UUID DEFAULT uuid_generate_v4(),
    company_id UUID NOT NULL REFERENCES company(company_id)
                            ON DELETE CASCADE ON UPDATE CASCADE,
    address_type ADDRESS_TYPE NOT NULL DEFAULT 'mailing',
    address_line_1 VARCHAR(255) NOT NULL,
    address_line_2 VARCHAR(255) DEFAULT NULL,
    address_line_3 VARCHAR(255) DEFAULT NULL,
    address_line_4 VARCHAR(255) DEFAULT NULL,
    city VARCHAR(255) NOT NULL,
    state VARCHAR(2) NOT NULL,
    zip_code NUMERIC(5) NOT NULL,
    zip_code_4 NUMERIC(4) DEFAULT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    PRIMARY KEY (company_id, address_type)
);
```

This record is tied to the company, so a foreign key is defined.

Standard address lines with room for department or floor numbers

Defaults the address_type to a mailing address

Standard address lines

Fields that can help with auditing

Prevents more than one type of address for a company by defining a primary key that also uses the address_type

We also created a table named `company_phone` to store phone numbers for the company. Again, we define a `phone_type` to identify different categories of phones. We also define a primary key of the company and phone type so that each company can have only one type of phone defined. Another restriction is to have unique `phone_number` and `extension` so that a phone number cannot be reused.

Listing 10.5 The `company_phone` type and table

```
CREATE TYPE phone_type AS ENUM ( 'main', 'direct', 'department',
'fax', 'toll-free', 'mobile', 'home', 'other');

CREATE TABLE company_phone (
    company_phone_id UUID DEFAULT uuid_generate_v4(),
    company_id UUID NOT NULL REFERENCES company(company_id)
                            ON DELETE CASCADE ON UPDATE CASCADE,
    phone_type VARCHAR(25) NOT NULL,
    phone_number NUMERIC(10) NOT NULL,
    extension NUMERIC(5) DEFAULT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    PRIMARY KEY (company_id, phone_type),
    UNIQUE (phone_number, extension)
);
```

Defines possible values for types of phones

Phone number fields

Audit fields

The table is associated with the company table with a foreign key.

Prevents more than one phone number from being used

Prevents having more than one type of phone per company

We should now have more than enough database structure to support companies in our ACH dashboard. Much of this information is not strictly necessary when working with ACH files, but the additional information will be useful for expanding our project with different functionality.

10.2.2 Using generative AI to generate mock data

We can use generative AI to help boost our productivity by creating some test data for our companies. Using the industries we defined for our SQL type `industry_type`, we can instruct ChatGPT to do the following:

> CK Given this list of industries "'basic materials,' . . . 'utilities'," create fake company names, address information, and logos for each industry.

We were provided with company names such as Elemental Resource Inc. with the address of 123 Mineral Lane, Boulder, CO 80301. We also asked for randomly generated EIN and DUNS numbers. It is also possible to see whether ChatGPT can share with us any interesting phone numbers that we can include in our data. For instance, here are two amusing ones if you have anyone on your team who likes Phil Collins or Hall and Oates:

- 424-888-PHIL (7445), known as the Phil Call-Ins Hotline, where you can listen to songs from Phil Collins.
- 719-266-OATES (2837), known as the Callin Oates Hotline, where you can listen to a variety of songs from Hall and Oates.

While having some fun test data can brighten our day, we need to get all that data incorporated into SQL as well. Up until this point, we have been keeping all our SQL in the init.sql in the Dockerfile to create tables and data. With the need to input the generated company data, we want to do a bit of restructuring. We create a subfolder named `sql` and move all the `SQL` commands that currently exist in init.sql, while breaking them up into different files. It should be easier now to manage and update our SQL as necessary. The following listing shows the new structure of our Docker folders.

Listing 10.6 Docker folder structure for the database

```
docker/
├─ db/
|  ├─ sql_scripts/
|  |  ├─ sql/
|  |  |  ├─ create_tables.sql
|  |  |  ├─ general_ach_data.sql
|  |  |  ├─ elemental_resources_inc.sql
|  |  ├─ init.sql
```

Next, we update init.sql so that it references the new files.

Listing 10.7 Updating the init.sql file

```
…
\i /docker-entrypoint-initdb.d/sql/create_tables.sql
\i /docker-entrypoint-initdb.d/sql/general_ach_data.sql
\i /docker-entrypoint-initdb.d/sql/
➥elemental_resources_inc.sql
…
```

Includes the creation of tables

Includes general ACH data such as exception information

Each company can be stored in its own file.

Each company is added with the following SQL.

Listing 10.8 Adding a company

```
DO
$$
    DECLARE
        saved_uuid UUID;
    BEGIN
        WITH company_insert AS (
            INSERT INTO company (name, tax_id_number,
                                 duns, industry, logo)
                VALUES ('Elemental Resources Inc.', 459876543,
        123456789, 'basic materials', '...')
                RETURNING company_id
        )
        SELECT company_id INTO saved_uuid FROM company_insert;

        INSERT INTO company_address (saved_uuid, address_line_1,
                                     city, state, zip_code)
        VALUES (saved_uuid, '123 Mineral Lane',
                'Boulder', 'CO', 80301);

        INSERT INTO company_phone (saved_uuid, phone_number)
        VALUES (saved_uuid, '7192662837');
    END
$$;
```

Creates a variable saved_uuid as we need it for the other inserts to satisfy the foreign key constraint

Uses a CTE to insert the company

Inserts a record with our test data

Inserts the phone using the stored company ID

Inserts the address using the stored company ID

Uses our CTE to save the company_id into our saved_uuid

Returns the company_id assigned by the database

We can tell ChatGPT to use the previous listing as a template to insert the company data it generated previously. Having ChatGPT generate some of this setup code makes it easier to get some default companies added to the database. We certainly could have

done it ourselves, but it would have been somewhat tedious and error prone as we copied/pasted the data.

10.2.3 API design

We start with a simple unit test with a new twist to ensure that we can get a `200 HTTP` response code, which will indicate we are reaching the endpoint successfully. As we have seen, we have default companies that are loaded into the database when our container is started. Typically, we have been clearing our database for each test using `SqlUtils().truncate_all()`, and if we wanted to avoid truncating a table, we would need to update that method to exclude that particular table. Running our tests in this fashion also required the containers to be up and running.

We will now create a test that uses Testcontainers to start and stop our containers so that we receive a fresh database every time. This makes the test self-contained, which is perfect for when we must move our project into a CI/CD environment.

Listing 10.9 Initial unit test with Testcontainers

```
...
@pytest.fixture(scope="class")
def client():
    return TestClient(app)
```

Defines a fixture to make test calls

```
@pytest.fixture(scope="module")
def docker_compose():
    absolute_path = Path("../../../../docker/db").resolve()
    with DockerCompose(absolute_path) as compose:
        compose.start()
        yield compose
        compose.stop()
```

Defines the container using Docker Compose because we have been using it throughout the project. Everything is set up as expected.

```
class TestCompanyApis:

    @pytest.fixture(autouse=True)
    def mock_client_host(self):
        with patch(
            "fastapi.Request.client",
            new_callable=lambda: type("Client", (),
                                      {"host": "127.0.0.1"}),
        ):
            yield
```

Defines an IP address for our record

```
    def test_get_companies(self, client, docker_compose):
        response = client.get("/api/v1/companies")
        assert response.status_code == 200, response.text
```

Defines a test that validates we return successfully; further validation can be done for the response as well.

We now have a way to run tests in an isolated environment. The more we isolate our tests and keep them self-contained, the easier it will be for the rest of the team to run them in their own environment. That way, we can avoid the WOMM (Works on My Machine) problems—a common theme in software development.

We now follow the same formula as for previous APIs. We first create a record that we can return and that contains some company information we will display as an overview. At this point, we just want to be able to display a list of companies that users can then drill down into. In the following listing, we define an enumeration for the `IndustryType`.

Listing 10.10 The `IndustryType` enumeration

```
class IndustryType(Enum):
    BASIC_MATERIALS = "basic materials"
    CONSUMER_GOODS = "consumer goods"
    CONSUMER_SERVICES = "consumer services"
    FINANCIALS = "financials"
    HEALTHCARE = "healthcare"
    INDUSTRIALS = "industrials"
    OIL_AND_GAS = "oil & gas"
    TECHNOLOGY = "technology"
    TELECOMMUNICATIONS = "telecommunications"
    UTILITIES = "utilities"
```

We define a `CompanyOverviewRecord` as shown in the following listing. While you have seen most of this code before, we do include a regular expression to verify the image is a valid format for displaying on the page.

Listing 10.11 The `CompanyOverview` Record

```
class CompanyOverviewRecord(BaseModel):
    company_id: Optional[UUID4] = Field(
        None,
        title="Company ID",
        description="Unique identifier for the company.",
    )
    name: Optional[str] = Field(
        None,
        title="Name",
        description="Name of the company.",
        max_length=255,
    )
    industry: Optional[IndustryType] = Field(
        None,
        title="Industry",
        description="Type of industry the company operates in.",
    )
    logo: Optional[str] = Field(
        None,
        title="Logo",
        description="Base64 Image of the Logo.",
```

We use the IndustryType to help with validation.

States that the logo should be Base64

```
        pattern=r"^data:image\/[a-zA-Z]+;
➥base64,[A-Za-z0-9+/=]+$"
    )
```

Enforces the correct format with a regular expression

Next, we create a class to handle any SQL calls dealing with companies. The following listing shows the simple `SELECT` statement using a `row_factory` of `class_row`. In such cases where we are simply returning the data and not performing data manipulation, we could also use the `dict_row` just as easily—the choice really depends on our application needs.

Listing 10.12 `CompaniesSql` class

Retrieves the database connection and sets the row_factory to a class_row for the CompanyOverviewRecord

```
class CompaniesSql:

    def get_all_companies(self) -> list[CompanyOverviewRecord]:
        with get_db_connection(row_factory=
➥class_row(CompanyOverviewRecord)) as conn:
            result = conn.execute(
                """
                SELECT company_id, name, industry, logo FROM companies
                """,
                [],
            )

        records = result.fetchall()

        return records
```

Limits the fields we are returning; we prefer specific fields rather than "*".

Returns all records

We are almost there! Now we need to add the endpoint. We have built the pieces starting from the bottom as we feel that makes things more reasonable as we develop the API. However, we could have started with defining the endpoint, drilling down to details of retrieving the data.

Listing 10.13 Getting the endpoint for all companies

We assign a prefix for all the endpoints.

```
router = APIRouter(prefix="/api/v1/companies")
…
@router.get(
    path="",
    response_model=list[CompanyOverviewRecord],
    summary="Retrieve an Overview of Companies",
    description="Return a list of defined companies.",
    response_description="An overview of the company.",
    tags=["Company"],
)
@log_message("Company Overview")
```

Standard documentation for the API

Our annotation to provide some audit information for the API

```
async def get_company_overview(request: Request) ->
list[CompanyOverviewRecord]:
    return CompaniesSql().get_all_companies()
```

The SQL call for getting all companies

Defines the function and that it will return a list of CompanyOverviewRecords

Finally, we only need to add the router to our main.py. We do a little refactoring to add the routers into an array and loop through them instead of having multiple `include_router` statements. The following listing shows the refactoring results.

Listing 10.14 Refactored `app.include_router`

```
routers = [files.router, audit.router,
➥ companies.router]
for router in routers:
    app.include_router(router)
```

Defines an array of routers

Loops through them and includes each of them, which prevents the somewhat redundant include_router statements

With that, we have a passing test, but we should not stop here. It is important to validate the data being returned as well, not only that we received a valid response code of `200`. We should take a moment to think about what else needs to be validated to feel confident in publishing the code with a passing test.

In this case, we could receive a `200`-response code but not receive any companies. If the database has been populated with company data, should we check whether those company names exist? Or maybe there is a subset of those companies? We only want to drive home the point that the purpose of testing is to have confidence in the code we have written. Even if we do not intend to publish our code to production after successful automated testing, having a test suite that covers and tests our code so thoroughly that we feel confident in publishing it is a valuable goal.

On the side note, in addition to the ability to read companies from the database, for a fully functioning dashboard, we also need the ability to add, update, and delete. This would be a great time to take a break from reading, pull down one of the project versions, and get busy with some full stack development!

10.2.4 User interface

The first step in creating the user interface is to add a button to NavButtons.tsx so that we can navigate to the new page. Finding an appropriate icon should be the hardest part of this process. Luckily, Material UI has more than enough icons to choose from.

Listing 10.15 NavButtons.tsx

When the navigation button is clicked, we navigate to the companies page.

```
<ListItemButton onClick={() => route.push("/companies")}>
```

```
   <ListItemIcon>
      <Store/>
      </ListItemIcon>
   <ListItemText primary="Companies"/>
</ListItemButton>
```

Defines the icon to display

Sets the text to Companies

When navigating to the company page, there will be a list of companies provided. We use the same pattern as for other pages. The page.tsx makes the API call retrieve the information about the companies, and we pass it to another component that does the actual work to display it.

Listing 10.16 The `CompanyListing` page

```
export default function CompaniesOverviewPage() {

    const [companies, setCompanies] =
useState<CompaniesListingResponse[]>([]);

    useEffect(() => {
        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        axios.get<CompaniesListingResponse[]>
➥(`${apiUrl}/companies`, {
…
            .then(response => {
                setCompanies(response.data);
            })
…
                    <CompaniesListing companies={companies}/>
…
```

Defines a function for the page

Variable to store the companies; defaulted to an empty array

API call

Passes the companies down to the CompaniesListing

When the response is successful, sets the companies from the response data

The URL for the API; defaults to an empty string when the environment variable is not defined

The `CompaniesListing` component, shown in the following listing, adds several new items that we have not used. The `onRowClick` for the DataGrid element allows us to click anywhere on the row and navigate to details about that company and the row height property so that we can see the company images a bit better. We also created an `ImageBase64` to encapsulate some of the needed pieces to display an image.

Listing 10.17 The `CompanyListing` component

```
interface CompaniesListingProps {
    companies: CompaniesListingResponse[];
}
```

Defines the properties for this component

```
export default function CompaniesListing({companies}:
Readonly<CompaniesListingProps>) {

    const route = useRouter();
    const columns: GridColDef[] = [
        {field: 'view', headerName: '', sortable: false,
            width: 100, renderCell: (params) => (
            <ImageBase64 base64={params.row.logo}
                         alt={params.row.name}
                         width="100%" maxWidth="200px" />
        )},
        {field: 'name', headerName: 'Name', width: 300 },
        {field: 'industry', headerName: 'Industry', width: 150 },
    ];

    const handleRowClick = (params: { id: any; }) => {
        const companyId = params.id;
        route.push(`/companies/${companyId}/dashboard`);
    };
…
       <DataGrid
          rowHeight={100}
          columns={columns}
          rows={companies}
          getRowId={(row) => row.company_id}
          onRowClick={handleRowClick}
       />
…
```

The function defines the properties as Readonly.

Obtains a router object so that we can navigate as necessary

Columns definitions for the DataGrid

An image element that can display the company logo

Defines the row click method that executes when a row is clicked and takes us to the company dashboard

Uses onRowClick to handle the user clicking a row on the DataGrid

The `ImageBase64` is straightforward—we have a routine to ensure that the image is in the correct format, as seen in the following listing. This allows us to perform validation, and we return an empty string if we do not have a correctly formatted string.

Listing 10.18 `Base64` image

```
const isBase64Image = (base64: string) => {
    const regex = /^data:image\/[a-zA-Z]+; base64,/;
    if (!regex.test(base64)) {
        return '';
    } else {
        return base64;
    }
}

export default function ImageBase64({base64,
      alt = 'Image',  width = '100%',  maxWidth = '100px'}:
➥ Readonly<ImageProps>) {

    const imageSource = isBase64Image(base64);

    return (
```

A function to ensure we have a src string formatted correctly to display the image in the source tag. If not, we return an empty string, which will display the alt tags for the image.

Assigns the result of the function call to isBase64Image to imageSource

Defines the component to allow for multiple properties and default values if they are not passed

```
            <Box
                component="img"
                src={imageSource}
                alt={alt}
                sx={{
                    width: width,
                    maxWidth: maxWidth,
                }}
            />
        );
}
```

There is no img component; instead, the appropriate parameters are passed to a Box element.

We could certainly have just used a `Box` element in our component or page, but wrapping the element and treating it as its own component has some benefits. We can use this tactic in multiple areas and enforce consistency throughout the code. Having a dedicated `ImageBase64` may be a too specific. We could create a more general element and pass a type to it that determines the type of `src` we are using and performs the necessary validation.

At this point, we have the database structure to store company information and the APIs to access the information about the company. We have also created a component capable of listing our companies, as shown in figure 10.2

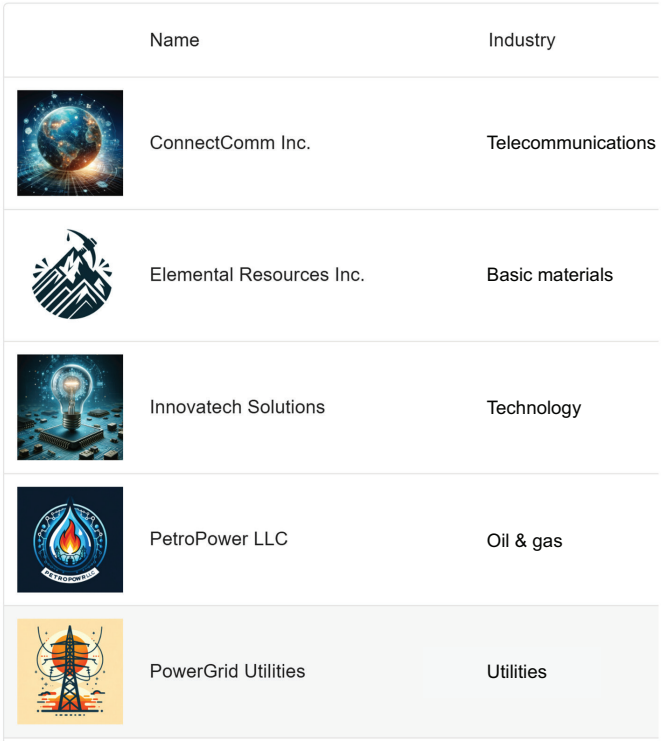


Figure 10.2 Component for listing the companies

Now that we can view the companies in our database, we begin putting our company dashboard together.

10.3 Company dashboard

With a way to navigate to a listing of companies in place, we want to be able to display a company-level dashboard. This feature will allow us to create a few other components that—although not necessarily unique to companies (we could incorporate versions of these components at the file level)—can provide information specific to companies and present other directions for our application to grow.

10.3.1 Navigation for the company dashboard

With a way to view all the companies, we want to be able to drill down and get to the company dashboard. The structure required to support the company listing and dashboard is shown in listing 10.19.

Listing 10.19 UI structure for companies

```
src/
├─ app/
|  ├─ companies/
|  |  ├─ [companyId]/
|  |  |  ├─ dashboard/
|  |  |  |  ├─ page.tsx
|  |  ├─ page.tsx
```

This page will contain the company specific dashboard; the [companyID] will be the UUID of the company.

This is the page that displays the company listing shown in figure 10.2.

With the page structure set up, we only need to create an empty dashboard page that can be populated with the company-specific components. We can use our file-level dashboard as a template.

Listing 10.20 Empty company dashboard

Gets the Company ID from the URL; this is possible because of the file-based routing being used and the directory we defined as [companyId].

```
…
export default function CompanyDashboard({params}: any) {

    const companyId: string = params.companyId;
    const [company, setCompany]
                    = useState<Company | undefined>(undefined);
…
    useEffect(() => {
        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
axios.get<CompanyDetailResponse>(`${apiUrl}
➥/companies/${companyId}`, {
…
            .then(response => {
        console.log(`Response data ${JSON.stringify
➥(response.data)}`);
```

Stores the company

Retrieves the company details

```
…
    }, []);

    return (
…
                    <Typography>Empty Company Dashboard</Typography>
…
    );
}
```

Placeholder for the dashboard to verify navigation

While we can now navigate to the dashboard and retrieve company details, the dashboard is a little barren. However, before we can start creating dashboard components, we need ACH batches for the companies. The next section shows how to produce ACH files with company-specific data.

10.3.2 Creating ACH files for our companies

We have already created BDD-style tests to generate ACH files for our dashboard. Thankfully, we can use the same code to create files specifically for our companies. The following listing shows a sample scenario we wrote to create a test file for building the company dashboard. We used ChatGPT only for getting some original names that might be appropriate for the company's customers.

Listing 10.21 Creating a file for Elemental Resources Inc.

```
  Scenario: Create an ACH file for company "Elemental Resources Inc"
    Given I want to create an ACH file named "elemental_resources.ach"
    And I want to have an immediate destination of "990000013"
    And I want to have an immediate origin of "987654321"
    And I want to have 2 batch with ACH credits and debits
➥ and a standard entry class code of "PPD"
    And I want 10 entries per batch with random amounts
➥ between 100 and 5000
    And I want to use individual names of "Rocky Gravel, Sandy Stone,
 Clay Earthman, Mason Bricks, Gemmy Quartz, Rusty Ironwood,
Flint Stoney, Dusty Boulderson, Shelly Shale"
    And I want to have company name "ElementalResourcesInc"
➥ and company id "459876543"
    When my ACH is created
    Then I should have a file of the same name
    And there should be 2 batch in the file
    And there should be 20 entries in the file
```

Here we only have to ensure that our company name is not too long as this could offset the record. In our original code, we made sure that the field itself was padded out to 16 characters when too short. Now, we have the opposite problem where we entered a field with 22 characters. The following listing shows a simple piece of code that can be used to truncate the company name.

Listing 10.22 Truncating and padding the company name

```
setup_info["company_name"] = company_name[:16].ljust(16, " ")
```

It may be tempting to go through the code and find all those areas that look like they should be truncating strings as well. If you go down that route, make sure to leave enough time for testing.

> **To fix or not to fix?**
>
> We encountered similar production problems before and had to make the call on whether we needed to expand the scope of the story (e.g., where an invalid amount was reported on a particular field). The user story was originally written to fix the single field reported as incorrect; however, after researching the problem and finding the root cause, it was discovered that the problematic pattern was used in several other places in the program. Due to the problem causing invalid data being sent to a third party, it was decided to update the story and expand the scope (i.e., increase the story points) so that the problem could be addressed in full.

With company data and specific company ACH files loaded into the database, we can start building the dashboard.

10.4 Company Information component

The first component we want to build is a screen with more details about the company. These will only be some additional details that can be edited as needed.

We create the required `Pydantic` class, which is used to return the data. We are returning more information in the `CompanyDetailRecord` than we need to display because the data will also be passed to a form where all the data will be updateable. The following listing shows the `CompanyDetailRecord` where we pass back the arrays of `Address` and `Phone` objects. The company has a one-to-many relationship for the `Address` and `Phone` types, so it is possible to have more than one row.

Listing 10.23 The `CompanyDetailRecord` field

```
class CompanyDetailRecord(BaseModel):
    company_id: Optional[UUID4] = Field(
        None,
        title="Company ID",
        description="Unique identifier for the company.",
    )
    name: Optional[str] = Field(
        None,
        title="Name",
        description="Name of the company.",
        max_length=255,
    )
```

```
...
    addresses: Optional[list[Address]] = Field(
        None,
        title="Addresses",
        description="List of Addresses.",
    )
    phones: Optional[list[Phone]] = Field(
        None,
        title="Phones",
        description="List of Phone Numbers.",
    )
```

An array of Address objects used to store addresses for the company

An array of Phone objects used to store phone numbers for the company

Because the address and phone records are arrays, we find that building the `CompanyDetailRecord` is somewhat more complicated than our previous data retrievals. This is because, before, we used to simply pull a record from the database and map it to the object. Now, we have multiple rows from multiple tables that we need to handle.

We could follow several approaches, but we want to stay with a single query like we did in the past. However, depending on our query design, this will potentially duplicate fields in the company table. Returning duplicated data is not the end of the world; however, it is an indicator that there is some room for improvement. After all, we have taken great pains not to carelessly return all fields by simply using an asterisk (which can result in returning fields we have no intention of using). The duplication may be of concern for performance reasons, as well as for causing confusion among developers.

It may not be obvious, but for each address and phone number, the query in listing 10.24 will return a row. So, if we have two addresses and three phone numbers, we will receive five rows from the query. Given the expected number of addresses and phone numbers, this is not extremely problematic at this point. We do have a relatively large amount of data being returned for the image, but as mentioned before, storing data as a `Base64`-encoded string was a matter of convenience.

Furthermore, we use `LEFT JOIN` as we have no guarantee that companies will have addresses or phones populated. Had we used `INNER JOIN`, companies would have required addresses and phone numbers to be returned by the query. While having company records without an address/phone is not likely, it is not impossible. Perhaps the initial record was set up but—for business reasons—the address was not populated (maybe there was no official documentation for their address), or perhaps, fields did not come over properly during a conversion or migration.

Listing 10.24 Query to return company details

We organized selected fields by their respective tables, and these are from the companies table.

```
   SELECT c.company_id, c.ach_company_id, c.name, c.tax_id_type,
c.tax_id_number, c.duns, c.logo, c.website, c.industry,
```

```
ca.company_address_id, ca.address_type, ca.address_type,
ca.address_line_1, ca.address_line_2, ca.address_line_3,
ca.address_line_4, ca.city, ca.state,
ca.zip_code, ca.zip_code_4,
cp.company_phone_id, cp.phone_type,
cp.phone_number, cp.extension
FROM companies AS c
LEFT JOIN company_addresses AS ca USING (company_id)
LEFT JOIN company_phones AS cp USING (company_id)
WHERE company_id = %s
```

These fields are from the company address table.

These fields are from the company phone table.

We use a LEFT JOIN because records are not required to be in these tables.

Limiting our details by the company, with %s being a placeholder

We have multiple rows and will need the application logic to ensure the duplicated data is not passed back. The following listing shows the full method to accomplish this. We must loop through each record and use a hash to ensure that `id` has not been encountered before.

Listing 10.25 Returning company details

```
def get_company_by_id(self, company_id: UUID) -> CompanyDetailRecord:
   with get_db_connection(row_factory=dict_row) as conn:
      result = conn.execute(
…
      )

      rows = result.fetchall()

      if not rows:
         raise KeyError(f"Record with id {company_id} not found")

      addresses = {}
      phones = {}
        for row in rows:
           if row["company_address_id"] not in addresses:
              addresses[row["company_address_id"]] = Address(
              …
              )
           if row["company_phone_id"] not in phones:
               …
              )

        company_data = {
            "company_id": rows[0]["company_id"],
            "name": rows[0]["name"],
            "tax_id_type": rows[0]["tax_id_type"],
            "ach_company_id": rows[0]["ach_company_id"],
            "tax_id_number": rows[0]["tax_id_number"],
            "duns": rows[0]["duns"],
            "logo": rows[0]["logo"],
            "website": rows[0]["website"],
            "industry": rows[0]["industry"],
```

Fetches all rows for our SQL query

If no details are found, this is an error and is reported as such.

Sets up empty hashes to store our data

Loops through each row that was returned

Creates an Address object as necessary

Creates a Phone object when we have a phone number not encountered before

We can just reference the company fields from our first row.

```
            "addresses": list(addresses.values()),
            "phones": list(phones.values()),
        }

        return CompanyDetailRecord(**company_data)
```

Creates an array from our hash

Creates an object using the company_data

While this code works as expected, we can also condense the resulting dataset by using more complicated SQL. We have explored CTE in Postgres in previous examples and can use some of the JSON processing available in Postgres as well. The tradeoff is that our SQL may become less portable, but JSON processing is available for other major databases, such as MySQL, with some slight modifications. The following listing shows the updated SQL.

Listing 10.26 Getting company data using Postgres CTE

```
WITH company_info AS (
   SELECT c.company_id, c.ach_company_id, c.name, c.tax_id_type,
          c.tax_id_number, c.duns, c.logo, c.website, c.industry
     FROM companies AS c
    WHERE c.company_id = %s
),
addresses AS (
   SELECT ca.company_id,
          JSON_AGG(
             JSON_BUILD_OBJECT(
                'company_address_id', ca.company_address_id,
                'address_type', ca.address_type,
                'address_line_1', ca.address_line_1,
                'address_line_2', ca.address_line_2,
                'address_line_3', ca.address_line_3,
                'address_line_4', ca.address_line_4,
                'city', ca.city,
                'state', ca.state,
                'zip_code', ca.zip_code,
                'zip_code_4', ca.zip_code_4
             )
          ) AS addresses
     FROM company_addresses AS ca
     WHERE ca.company_id = %s
     GROUP BY ca.company_id
),
phones AS (
   SELECT cp.company_id,
          JSON_AGG(
             JSON_BUILD_OBJECT(
                'company_phone_id', cp.company_phone_id,
                'phone_type', cp.phone_type,
                'phone_number', cp.phone_number,
                'extension', cp.extension
             )
          ) AS phones
```

Uses the JSON_AGG and JSON_BUILD_OBJECT to create an array of addresses for the company

Uses the company_addresses table

Selects only the records for our company

Groups the records by company_id

Uses the JSON_AGG and JSON_BUILD_OBJECT to create an array of phones for the company

```
    FROM company_phones AS cp
    WHERE cp.company_id = %s
    GROUP BY cp.company_id
)
SELECT ci.company_id, ci.ach_company_id, ci.name,
ci.tax_id_type,  ci.tax_id_number, ci.duns,
ci.logo, ci.website, ci.industry,
COALESCE(a.addresses, '[]'::json) AS addresses,
COALESCE(p.phones, '[]'::json) AS phones
FROM company_info AS ci
LEFT JOIN addresses AS a ON ci.company_id = a.company_id
LEFT JOIN phones AS p ON ci.company_id = p.company_id;
```

Uses the JSON_AGG and JSON_BUILD_OBJECT to create an array of phones for the company

Uses COALESCE to return an empty JSON array when there are no records in the table

The application becomes more streamlined, as shown in the following listing. We can use ** to unpack the returned JSON into keyword arguments passed to their respective objects, thus creating a list of elements.

Listing 10.27 Python code to return the data

```
        row = result.fetchone()

        if not row:
            raise KeyError(f"Record with id {company_id} not found")

        addresses = [Address(**address) for
➥ address in row["addresses"]]
        phones = [Phone(**phone) for phone in row["phones"]]

        company_data = {
            "company_id": row["company_id"],
             …
            "addresses": addresses,
            "phones": phones,
        }

        return CompanyDetailRecord(**company_data)
```

Loops through the data returned to create appropriate array

Assigns them to the company_data hash

Creates a CompanyDetailRecord object using the hash we created

Several factors should always be considered when choosing one path over another. This selection may depend on the application and data involved. Performing profiling on the application and doing `EXPLAIN ANALYZE` on the SQL queries are some of the ways to determine which method performs better. A human factor is also involved; it takes into consideration which version is clearer and easier to maintain. Either way, we can now retrieve and display customer data!

Company Information form challenge

Because we are prepopulating the company information, and completing a form covers much of the same material laid out in previous chapters, we believe this is another good opportunity to take a break and put together a form to display the company information. You may create something from scratch, find a template, or use

generative AI to see if things can be done faster that way. Using generative AI is a good way to get started. Try feeding the SQL tables or interfaces defined for the fields to see whether it can generate the boilerplate code.

With a basic form in place, we move to a component showing recent company batches.

10.5 Recent Batches component

When we navigate to the company dashboard, we want to be able to get an idea of activity for the company. A great way to see the activity is listing the batches found for the company. Being able to pull batch information will require us to have some knowledge of the ACH layout for batches because we need to gather some information from both the batch header (type 5) and the batch control (type 8) records. Figure 10.3 shows an example of the component we will be building.

Figure 10.3 Recent company batches

To build the Recent Batches component, we follow three easy steps:

1. Create an API endpoint in FastAPI.
2. Write the SQL query to gather the required data.
3. Display the returned data in a MUI DataGrid.

10.5.1 Implementing the recent batches API

Whereas we could build an endpoint that is related to other company functions, we will instead implement this function as a search. While doing so is certainly a preference for our individual approach, we feel that rather than having something such as `/companies/{company_id}/batches`, we will achieve more flexibility if we just create an endpoint similar to our transaction search from the previous chapter. Therefore,

we create an endpoint of `/api/v1/files/batches/search` and use it. The following listing from the files.py generates our API endpoint.

Listing 10.28 Batches search endpoint

```
@router.get(
…
)
@log_message("Retrieving ACH batches")
async def search_ach_batches(
    request: Request,
    criteria: str = Query(..., description="Search criteria for batches"),
) -> list[BatchSearchResponse]:
    return BatchSearchSql().get_batches(criteria)
```

The endpoint should be straightforward by now. The `get_batches` routine follows a flow very similar to that in our transaction search. We support a few different search methods right from the get-go because we ignore the YAGNI principle from chapter 3. While this is usually something we stay away from, rules are meant to be broken. Since this routine is based on the transaction search, it won't take too much effort to code this up to make it available (of course, testing is another matter). The following listing shows the initial `get_batches` method. While these patterns will likely be useful in the future, for now, we are interested in when we do not match any of the previous patterns and will instead search using company's UUID. Recall that when we are on the company dashboard, we have the company's UUID in the URL. So, this will be easy to pass as a query parameter for the search.

Listing 10.29 The `get_batches` method

```
    def get_batches(self, criteria: str) -> list[BatchSearchResponse]:
        amount_pattern = r"^\d+\.\d{2}$"
        multiple_amounts_pattern = r"(^\d+\.\d{2})\s+(\d+\.\d{2})$"
        company_id_pattern = r"^\d+$"
        company_name_pattern = r"^[a-zA-Z1-9\s]+$"
        if re.match(amount_pattern, criteria):
            return self._get_batches_using_amount(criteria)
        elif match := re.match(multiple_amounts_pattern, criteria):
            begin_amount, end_amount = match.groups()
            return self._get_batches_using_amount_range(begin_amount,
end_amount)
        elif re.match(company_id_pattern, criteria):
            return self._get_batches_using_company_id(criteria)
        elif re.match(company_name_pattern, criteria):
            return self._get_batches_using_company_name(criteria)
        else:
            return self._get_batches_using_company_uuid(criteria)
```

We created the ability to search for batches using many different methods, even though we are only interested in searching by the UUID at this point.

The call for `_get_batches_using_company_uuid` is simple. As with our previous search queries, we have pulled the selection logic out of the query into a shared method since each of these search methods shares the same selection logic and only differs in how we filter the results.

Of course, the selection criteria are going to be the biggest and most important part of this search effort. The query needs to pull together a few different tables to build the search response. Listing 10.30 shows the SQL we used. Note that most of the tables are `INNER JOIN`s, but the companies table is using a `LEFT JOIN` because there is no guarantee that the company name exists in our database.

We know that ACH files can contain batches from companies that may or may not be our customers. Because it is entirely possible that our search results may return a batch for a specified company not defined in our database, the query may fail to return any results. We use the Postgres `COALESCE` command to pull the name from the company table. If that is `NULL`, we will use the record from the batch header and, finally, an empty string. The following listing shows the SQL query.

Listing 10.30 SQL query

We use the name as we have it in our database, as displayed on the batch header, or an empty string.

```
SELECT art1.ach_files_id AS file_id,
       art5.ach_records_type_5_id AS batch_header_id,
       af.file_name AS filename,
       COALESCE(c.name, abh.company_name, '') AS company_name,
       abh.company_identification AS company_identification,
       abcr.total_credit_entry_dollar_amount
                          AS total_credit_entry_dollar_amount,
       abcr.total_debit_entry_dollar_amount
                          AS total_debit_entry_dollar_amount,
       abcr.entry_addenda_count AS entry_addenda_count
  FROM ach_files af
INNER JOIN ach_records_type_1 art1 USING ( ach_files_id )
INNER JOIN ach_records_type_5 art5 USING ( ach_records_type_1_id )
INNER JOIN ach_records_type_8 art8 USING ( ach_records_type_5_id )
INNER JOIN ach_batch_headers abh USING ( ach_records_type_5_id )
INNER JOIN ach_batch_control_records abcr USING ( ach_records_type_8_id )
 LEFT JOIN companies c ON
 abh.company_identification = c.ach_company_id
```

We use a LEFT JOIN but must use ON instead of USING since the fields we want to join are not named the same.

The results of our query are stored in the `BatchSearchResponse` object in api/batch_search_response.py and is a standard Pydantic data model that we will not elaborate on since we have defined a fair amount of these already.

10.5.2 Displaying recent batches

This UI component is simple and is responsible for making an API call to get data and display the results in a DataGrid. The main thing to consider is whether we truly want the component to be responsible for making the API call. We have changed our approach when handling different components. When multiple components are on a page and use similar data, we pass that data to the components to minimize server calls, hoping to reduce latency and provide a better UX. In this case, the data is specific to this component, so we keep everything contained in the component. We can envision that this may become problematic if we were to allow users customize components on the dashboard, but some redesign would be required, and we will cross that bridge once we get there.

Listing 10.31 shows the basics of the UI component. We use the `renderCell` attribute of the MUI DataGrid to provide clickable View links that take a user directly to the file or batch. We also use the `formatCurrency` method whenever we need to format a currency. Also, note how we can use our search API with the parameter `companyId` that was passed in. So, in this case, the search is programmatically driven and not necessarily done via the user input like when searching for transactions.

Listing 10.31 The `RecentCompanyBatches` Component

```
interface RecentCompanyBatchesProps {
    companyId?: string;
}

export default function
➥RecentCompanyBatches({companyId}: Readonly<RecentCompanyBatchesProps>) {

    const router = useRouter();
    const [results, setResults] =
➥ useState<AchBatchSearchResponse[]>([]);

    useEffect(() => {

        if (!companyId) {
            return;
        }

        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        axios.get<AchBatchSearchResponse[]>(`${apiUrl}/files/
➥batches/search?criteria=${companyId}`, {
…
    }, [companyId]);

    const columns: GridColDef[] = [
       {field: 'file_id', headerName: 'File',
        sortable: false, width: 50, renderCell: (params) => (
            <Link onClick={() =>
router.push(`/fileDetails/${params.row.file_id}`)}
    sx={{ cursor: 'pointer' }}>View</Link>
```

The useRouter enables the component to navigate to other pages.

Stores the search response

Uses Axios to retrieve data from our endpoint

Creates a hyperlink that allows us to navigate to the file containing the batch

```
            )},
          {field: 'batch_header_id', headerName: 'Batch',
           sortable: false, width: 50, renderCell: (params) => (
              <Link onClick={() =>
router.push(`/fileDetails/${params.row.file_id}/batchDetails/
   ${params.row.batch_header_id}`)}
           sx={{ cursor: 'pointer' }}>View</Link>
…
          {field: 'total_credit_entry_dollar_amount',
              headerName: 'Total Credit', width: 125,
              renderCell: (params) => (
                 formatCurrency(Number(
                    params.row.total_credit_entry_dollar_amount))
              )},
          {field: 'total_debit_entry_dollar_amount',
           headerName: 'Total Debit', width: 125,
           renderCell: (params) => (
              formatCurrency(Number(params.row.total_debit_entry_dollar_
     amount))
              )},
    ];

    return (
…
       <Title>Recent Company Batches</Title>
       <DataGrid rows={results}
                 columns={columns}
                 getRowId={(row: any) => row.batch_header_id}/>
…
```

Creates a hyperlink that allows us to navigate directly to the batch

Reuses our formatCurrency routine

Passes the required fields to create our grid; we also need to specify an ID for the row because we do not have a field named ID.

Now, we can show users current activity for the company, which is very useful and provides an at-a-glance UX. If a customer were to call for ACH support, they are mostly likely calling about something that happened recently. Being able to quickly jump to a particular file is going to be important. The next component will provide another way to help customers stay on top of their ACH files.

10.6 Expected Files component

Another useful component for our company dashboard is a component that allows users to keep track of any expected ACH files for a company. A company may want a financial institution to process files on a set schedule, and if the particular file has not arrived, then someone needs to be alerted and follow up. For instance, if the financial institution's customer expects to see their paycheck on the 15th, then the bank better make sure it is processed before that date.

This component displays a simple list and an indicator of whether the file has been loaded. We will be using the filename as the basis for whether a file has been loaded. It is common for financial institutions, data centers, and similar to rely on naming conventions to help facilitate various checks and balances, so using a filename for our component is not a unique approach to our dashboard. Of course, we could also expand the

conditions to match any other constraints that might be useful from an ACH perspective—whether it is the company ID or totals from the batch.

Although we are approaching this problem from the company perspective, it would also be incorporated from a more general perspective (e.g., a financial institution will also expect files on a set schedule). Figure 10.4 shows an example of the component.

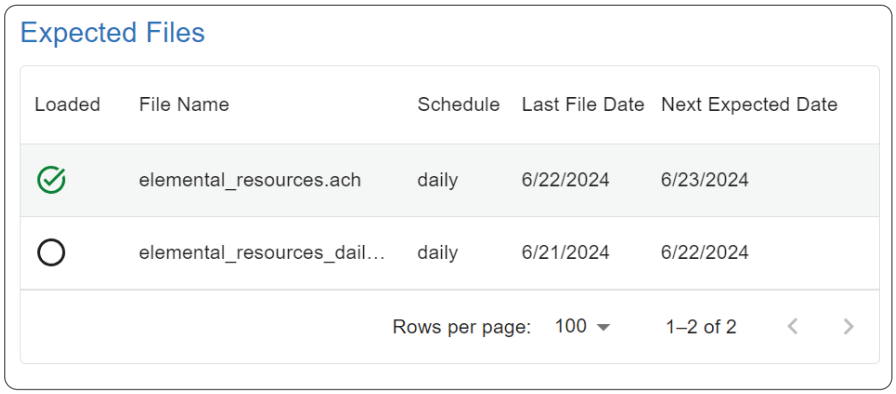


Figure 10.4 Expected Files component

We build the Expected Files component using the following steps:

1. Define the database table and type.
2. Define the API endpoint.
3. Display the data using a MUI DataGrid.

10.6.1 Storing expected files

For our database, we create another database table named `company_expected_files` to store a list of filenames. We also create a `SCHEDULE_TYPE` like before to limit the possible values in the field. The table will only be responsible for holding the data necessary to determine whether the file should be found. There may be cases where an external job would make the determination and update a flag in the table. That is not required in our case, and we can rely on the API call to return the additional flag.

The following listing shows the creation of the table and type. We also use the `last_file_date`, which has to be maintained to be adjusted. We need to revise our file-upload logic to update this field.

Listing 10.32 Creating the type and table for expected files

```
CREATE TYPE schedule_type AS ENUM ('daily', 'weekly', 'bi-weekly',
       'monthly', 'quarterly', 'semi-annually', 'annually');
```

```
CREATE TABLE company_expected_files (
    company_expected_files_id UUID DEFAULT uuid_generate_v4(),
    company_id UUID NOT NULL REFERENCES companies(company_id)
    ON DELETE CASCADE ON UPDATE CASCADE,
    file_name VARCHAR(255) NOT NULL,
    schedule SCHEDULE_TYPE NOT NULL DEFAULT 'daily',
    last_file_date TIMESTAMP DEFAULT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW(),
    PRIMARY KEY (company_id, file_name)
);
```

Although it is not required, we define a UUID that the record could be referenced by.

The key is actually the company and filename as we only want to have one filename per company.

We take similar steps to populate the database with some data to see the component in action. Listing 10.33 shows the `INSERT` for the `company_expected_files` table. We can insert these records for each company by using a `SELECT` statement to select the `company_id` field based on the `tax_id_number` for the company. Remember that the `company_id` is set dynamically by the database when the record is inserted. We could also choose to insert the record with a known UUID, but using the preceding approach allows us to work with the data as we expect to see it in production and prepare ourselves for the times when we do not have the luxury of inserting a known field in our test.

Listing 10.33 Inserting records into the expected files table

```
INSERT INTO company_expected_files ( company_id, file_name,
            schedule, last_file_date )
VALUES (
   (SELECT c.company_id FROM companies AS c
             WHERE tax_id_number = '345678901'),
   'stellar_services.ach',
   'daily',
   NOW() - INTERVAL '1 day'
);
```

Gets the company_id by searching companies by the tax_id_number

With our database set up, we can now work on the logic that will power the `Expected-Files` component.

10.6.2 Implementing the expected files API

With the data for expected files loaded into the database, we need to be able to retrieve it, and we also need to be able to update the `last_file_date` when a file is uploaded.

The first thing we did was start with a test to work on building the required SQL logic. Again, we utilized Testcontainers to run our `docker_compose` container for the duration of the test.

Listing 10.34 shows the basic structure of our test file. As part of the test, we created a basic `CompanySearchSql` class to search for companies by their `tax_id_number` or `ach_company_id` number. The need for such a class is another result of the database dynamically assigning the UUID. To avoid scope creep, we only built enough functionality to get a single company, but it could easily be incorporated into a search page or company

page so that when we have hundreds or thousands of customers, we do not have to page through them all.

Listing 10.34 Test for the `ExpectedFilesSql` class

```
@pytest.fixture(scope="module")
def docker_compose():
    absolute_path = Path("../../../../../docker/db").resolve()
    with DockerCompose(absolute_path, build=True,
                       services=["postgres"])
 as compose:
        compose.start()
        yield compose
        compose.stop()
…
    def test_get_expected_files_for_company(self, docker_compose, sql):
        company_id =
CompanySearchSql().get_companies("459876543").company_id
        assert company_id is not None
        results = sql.get_expected_files_for_company(company_id)
        assert len(results) == 2
        assert any(
            record.file_name == "elemental_resources.ach"
                                            for record in results
        )
        assert any(
            record.file_name == "elemental_resources_daily_billpay.ach"
                                            for record in results
        )
```

Uses Testcontainers and DockerCompose to build the container and start only the Postgres service. There is no need to start CloudBeaver for a temporary test.

Ensures the container is started and stopped for our test

Uses CompanySearchSql to get the company UUID

Runs the SQL that we want to test

Makes sure we actually find something

Makes the necessary assertions to ensure the method worked

With our unit test in place, we are free to build the logic behind the `get_expected_files_for_company`. The `SQL` query creates two additional fields that are not stored directly in the database. These fields are a Boolean value for whether the file has been loaded and the `next_file_date`, which is the next time we expect to see the file according to the file's schedule. For now, we do not consider filtering our results by the date simply because that makes testing a bit more difficult. So, we are returning all the data. This does not necessarily make sense for the component when the current objective is to show files that were expected today. However, it works well enough to demonstrate the intent of the functionality. We could expand this component or create a new one that shows files for the coming week, gets the number of files expected, and similar. The `SQL` query utilizes the `INTERVAL` function, which makes it easy to compute the given time interval. The following listing shows the `SQL` statement.

Listing 10.35 SQL for the Expected Files component

```
WITH future_file_dates AS (
   SELECT ef.company_expected_file_id,
       ef.company_id,
       ef.file_name,
       CASE ef.schedule
          WHEN 'daily' THEN ef.last_file_date + INTERVAL '1 day'
          WHEN 'weekly' THEN ef.last_file_date + INTERVAL '1 week'
          WHEN 'bi-weekly' THEN ef.last_file_date + INTERVAL '2 weeks'
          WHEN 'monthly' THEN ef.last_file_date + INTERVAL '1 month'
          WHEN 'quarterly' THEN ef.last_file_date
                                              + INTERVAL '3 months'
          WHEN 'semi-annually' THEN ef.last_file_date
                                              + INTERVAL '6 months'
          WHEN 'annually' THEN ef.last_file_date + INTERVAL '1 year'
          ELSE ef.last_file_date
       END AS next_file_date
  FROM company_expected_files AS ef
 WHERE ef.company_id = %s
)
SELECT ef.company_expected_file_id, ef.file_name, ef.schedule,
    ef.last_file_date, ffd.next_file_date,
    CASE
       WHEN af.file_name IS NOT NULL
            AND af.created_at::date = ef.last_file_date::date
               THEN TRUE
       ELSE FALSE
    END AS file_loaded
   FROM company_expected_files AS ef
INNER JOIN future_file_dates AS ffd USING (company_expected_file_id)
 LEFT JOIN ach_files AS af ON ef.file_name = af.file_name
     WHERE ef.company_id = %s;
```

Uses a CTE to determine when the file is next expected

Determines whether the file was loaded

Finally, we revise the posting of a file to update the `last_file_date` and `updated_at` fields for the table. This approach works well for now—when expected files are loaded, the date will increment accordingly and everything will function normally. However, problems arise when files are not uploaded. Perhaps the expected file does not need to be loaded that day–maybe it was a company holiday. For that reason, many financial institutions do not process files on weekends. When a file is loaded on a Friday, it may be necessary to adjust the next expected day to Monday instead of a Saturday. Handling these edge cases or addressing our users' additional needs helps make our software more robust. The following listing shows the addition of updating the necessary dates for our expected files.

Listing 10.36 Updating the dates for our expected files

```
    parser = AchFileProdates
    ach_file = AchFileSchema(file_name=file.filename,
file_hash=file.content_type)
    ach_files_id = AchFileSql().insert_record(ach_file)
```

```
    try:
        print(f"Processing file {temp_file.name}")
        seq = parser.parse(ach_files_id, temp_file.name)
        ExpectedFilesSql().update_last_file_date(file.filename)
    except Exception as e:
        print(e)
        return {"error": "Invalid file"}
    return {
        "filename": file.filename,
        "content_type": file.content_type,
        "last_seq": seq,
```

Once parsing of the file is completed, we can update the date.

The logic of updating the dates can be kept simple for now. The `updated_at` is just for audit purposes, and determining when the file was uploaded. The last time the file was uploaded was also at the time of the update, so we use the function `NOW` for both updates. The following listing shows the simple update logic used by the `update_last_file_date` mehod.

Listing 10.37 Simple update logic

```
UPDATE company_expected_files
                    SET last_file_date = NOW(),
                        updated_at = NOW()
                  WHERE file_name = %s
```

Updates the table to indicate the file has been uploaded

Things would get more complicated if we wanted to always keep the last file date set to a Monday and the processing fell on a weekend. This is an important problem because financial institutions may not be open on the weekend (or federal holidays), and it may not be appropriate to process files on those days. We certainly do not want to be enjoying our Sunday afternoon only to have numerous alerts come in, showing the files are missing! The following listing assumes a daily interval, but we could easily change this dynamic as needed.

Listing 10.38 Default to Monday as needed

If the date falls on a Sunday or Saturday, adds the appropriate number of days to force it to Monday. Be aware that this does not work if Monday is a holiday and the financial institution is closed. That would take some additional effort.

```
WITH adjusted_date AS (
    SELECT
        NOW() + INTERVAL '1 day' AS initial_date
)
UPDATE company_expected_files
        SET last_file_date = (SELECT CASE
                            WHEN EXTRACT(DOW FROM initial_date) = 0 THEN
initial_date + INTERVAL '1 day'
                            WHEN EXTRACT(DOW FROM initial_date) = 6 THEN
initial_date + INTERVAL '2 day'
                            ELSE initial_date
                        END
```

```
                                FROM adjusted_date),
            updated_at = NOW()
        WHERE file_name = %s
```

While we are not overly concerned about addressing the field `last_file_date`, we should begin to see some of the problems that may occur with scheduled tasks, especially when we must consider what happens when files are not loaded. It is unlikely that our application should have to directly address this, and—ironically enough—there may be another scheduled job that needs to run and clean the database. For now, we can retrieve the needed data from the database, so let's proceed with creating the UI side of our component.

10.6.3 Displaying the expected files

The heavy lifting should have been done on the backend. The UI component simply needs to display the information and an icon dependent on the `file_loaded` Boolean that is returned as part of the response. The most interesting part of the UI component (considering we have done this many times by now) is the ternary statement we use to decide whether to display a `TaskAlt` or a `RadioButtonUnchecked` icon. Otherwise, it is the typical flow of adding the component to the page.tsx and having the component retrieve the necessary data and display it in the DataGrid. The following listing shows the UI component.

Listing 10.39 The `ExpectedFiles` component

```
…
interface ExpectedFilesProps {
    companyId?: string;
}
export default function ExpectedFiles({companyId}:
➥ Readonly<ExpectedFilesProps>) {

    const [results, setResults] =
     useState<CompanyExpectedFilesResponse[]>([]);
…
  axios.get<CompanyExpectedFilesResponse[]>
➥(`${apiUrl}/companies/${companyId}/expect…

    const columns: GridColDef[] = [
        {field: 'file_loaded', headerName: 'Loaded', width: 75,
         renderCell: (params) => (
                params.value ? <TaskAlt sx={{color: "green"}} /> :
<RadioButtonUnchecked />
…
        {field: 'file_name', headerName: 'File Name', width: 200},
        {field: 'schedule', headerName: 'Schedule', width: 75},
…
            field: 'last_file_date', headerName: 'Last File Date',
            width: 100,
            renderCell: (params) => (
```

We use a ternary statement to determine which icon to display.

```
                params.value ? new Date(params.value as
➥ string).toLocaleDateString() : ''
…
            field: 'next_file_date',
            headerName: 'Next Expected Date', width: 150,
            renderCell: (params) => (
                params.value ? new Date(params.value as
➥ string).toLocaleDateString() : ''
…

    return (
…
                    <Title>Expected Files</Title>
                    <DataGrid rows={results} columns={columns}
                     getRowId={(row: any) =>
                     row.company_expected_file_id}/>
```

That completes the expected files component. We started by defining the required database table and a custom type to support the scheduling of expected files. Then we created a unit test with Testcontainers so that we could get a fresh database for each run. This approach allowed us to work out the needed SQL to return the data required for our component. Finally, we called the API from our UI and formatted the data to display it to the user.

Next, we look to add another component that can help companies monitor and control suspicious activity with company limits.

10.7 Company Limits component

This component is centered around proactive monitoring for the company. We would like the company to be able to set limits that would trigger alerts and possibly prevent transactions from being posted should those limits be hit. This can help companies identify and deal with transactions or activity that may be outside of normal behavior.

As a business, if you expect less than, say, $10,000 of debit activity in a day, and suddenly, there is a $15,000 batch being posted for your company, you likely want to be alerted about that activity. In principle, this is a straightforward component that lists totals for the day and compares them to a limit set by the company or the bank—functionality that needs to be built out in a future sprint. In practice, we need to handle alerting the customer, allowing for processing when the company is okay with exceeding the limit, and rejecting the file load otherwise. Figure 10.5 shows the Company Limits component we will be implementing.

We work on creating this component using the following steps:

1. Create the database table.
2. Define the API endpoint.
3. Display the returned information to the user.

Figure 10.5 Company Limits component

10.7.1 Storing the company limits

We need a new table to store limits for the company. As with previous tables, this table will use a foreign key that references the company ID from the companies table. Remember that the use of the foreign key helps us maintain referential integrity for the database. The `ON DELETE CASCADE` and `ON UPDATE CASCADE` will ensure the record is removed when the company is deleted or changed accordingly if updated. Not every company needs to have limits, so we will keep that in mind as we create queries which utilize this table.

Listing 10.40 Company Limits table

```
CREATE TABLE company_limits
(
    company_limit_id   UUID                DEFAULT uuid_generate_v4(),
    company_id         UUID      NOT NULL REFERENCES
                                 companies (company_id)
                   ON DELETE CASCADE ON UPDATE CASCADE,
    daily_debit_limit  NUMERIC(12, 2)      DEFAULT NULL,
    daily_credit_limit NUMERIC(12, 2)      DEFAULT NULL,
    created_at         TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at         TIMESTAMP NOT NULL DEFAULT NOW()
);
```

Keeping track of when records are created and updated can help with auditing.

There are a few approaches to implementing this functionality, and we will use a basic one that can be expanded as needed. The basic approach will not require us to store the totals. Instead we rely on the API to gather the totals and determine whether they are exceeded. This means that our logic will be executed when the company dashboard is displayed and when the file has finished loading so that we can add an associated exception. Remember from chapter 8 that if we want to add an exception, we will have to add it to the database along with any recovery options if desired. The following listing shows how the general_ach_data.sql was updated to insert these records.

Listing 10.41 Setting up the exception data

```
INSERT INTO ach_exception_codes (exception_code,
    exception_severity, exception_description)
VALUES ('006', 'warning',
 'Company Limits  Exceeded');
…
INSERT INTO ach_recovery_options
       (exception_code, recovery_option)
VALUES ('006', 'Alert company their limit has been exceeded');
```

We can add a new exception code for our company limits condition.

It is helpful to provide the users with some guidance on what to do when an exception is encountered.

We also set up the data for the company limits so that there is existing data in the database. The following listing shows the simple insertion into the database table.

Listing 10.42 Inserting records for the company limit

```
INSERT INTO company_limits
   ( company_id, daily_debit_limit, daily_credit_limit)
VALUES (
        (SELECT c.company_id FROM companies AS c
              WHERE tax_id_number = '459876543'),
        400.00,
        750.00
       );
```

That should be all we need to get up and running with company limits. The next section will show you how to update APIs so that you can work with the table.

10.7.2 Implementing the company limits API

Take a minute to think about what we would like to accomplish with the company limits API. Of course, we need to be able to retrieve what the limits are set to, but we will also want to determine what the current totals are for the company and a Boolean flag to indicate whether the limits have been exceeded. As mentioned, when designing the database tables, we decided not to store the current totals for the debits and credits, although we certainly could have created those fields and updated them as we loaded them.

A potential drawback in this case is that depending on when/where we decide to do this check, we may have limited ability to narrow down the problem for the user. We should take a moment to think about where the exception can be written. For example, we just added a check for the expected files right after the file was loaded. Should we put this check in the same place? That may be our first inclination since we just added similar code. However, by putting it there, we miss out on being able to identify which batch is responsible for causing the company limit to be exceeded as we do not access the record numbers at that point, and it makes it difficult to put the actual batch header on the exception. The following listing shows that we included the exception during the processing of the batch control record.

Listing 10.43 Adding the exception

```
case "8":
    ach_record = AchRecordType8Schema(
        ach_records_type_5_id=current_batch_header_id,
            unparsed_record=line,
            sequence_number=sequence_number,
         )
    ach_record_id = AchRecordsSqlType8().insert_record(ach_record)
    self.last_trace_number = None
    self._parse_batch_control(ach_record_id, line)
```

```
    if CompanyLimitsSql().
➥file_exceeds_company_limits(ach_file_id):
        AchExceptionsSql().insert_record(
            AchExceptionSchema(
                ach_files_id=ach_file_id,
                ach_batch_id=current_batch_header_id,
                record_number=sequence_number,
                exception_code=
                   AchExceptions.COMPANY_LIMITS_EXCEEDED.value,
            )
        )
```

Checks whether we exceeded the company limits

Calls the insert_record method from AchExceptionsSql to insert the record

This is our standard exception schema. It is the first time we actually use the ach_batch_id, although there are plenty of exceptions we need it for.

We write out our new `COMPANY_LIMITS_EXCEEDED` exception when a file exceeds company limits. This means that we need to define the `CompanyLimitsSql` class and a method to determine whether the file exceeds the company limits. We can tell from the use of the method that it just needs to return a Boolean value to determine whether this causes an exception. The method itself is straightforward, with the usual getting of the connection running a SQL query and fetching the results. The following listing shows the method details.

Listing 10.44 Determining whether a file exceeded company limits

```
with get_db_connection(row_factory=class_row(CompanyLimitsRecord)) as conn:
   result = conn.execute(
   """
      WITH ach_batch_totals AS (
         SELECT
                c.company_id,
                SUM(abcr.total_credit_entry_dollar_amount)
                                    AS current_credit_total,
                SUM(abcr.total_debit_entry_dollar_amount)
                                    AS current_debit_total
           FROM ach_files AS af
     INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
     INNER JOIN ach_records_type_5 AS art5 USING (ach_records_type_1_id)
     INNER JOIN ach_records_type_8 AS art8 USING (ach_records_type_5_id)
     INNER JOIN ach_batch_headers AS abh USING (ach_records_type_5_id)
     INNER JOIN ach_batch_control_records AS abcr
        USING (ach_records_type_8_id)
     INNER JOIN companies AS c
        ON c.ach_company_id = abh.company_identification
          WHERE af.ach_files_id = %s
       GROUP BY c.company_id
   )
   SELECT 1
   FROM ach_files AS af
```

Creates a CTE to select the batch totals

Sums the credits and debits

We only need to return something.

```
    INNER JOIN ach_records_type_1 AS art1 USING (ach_files_id)
    INNER JOIN ach_records_type_5 AS art5 USING (ach_records_type_1_id)
    INNER JOIN ach_records_type_8 AS art8 USING (ach_records_type_5_id)
    INNER JOIN ach_batch_headers AS abh USING (ach_records_type_5_id)
    INNER JOIN companies AS c ON c.ach_company_id = abh.company_identification
    INNER JOIN company_limits AS cl USING (company_id)
     LEFT JOIN ach_batch_totals AS abt USING (company_id)
         WHERE af.ach_files_id = %s
           AND ( abt.current_credit_total > cl.daily_credit_limit
                 OR abt.current_debit_total > cl.daily_debit_limit )
         LIMIT 1
    """,
    [file_id, file_id],
    )
return result.fetchone() is not None
```

Selects the file and sums

We want to return a Boolean, so the "is not None" helps us return True/False.

The previous listing helps us determine whether we need to write an exception, but we also need to define a method that will be used with the UI. This is similar to the previous SQL with the addition of the actual fields for the limits, the current totals, and flags indicating whether the limit was exceeded. We prefer to return explicit flags for the UI to use rather than have the UI rely on making the determination itself. The following listing shows the SQL used to return the required information for our `get_company_limits` method.

Listing 10.45 Query for the totals, limits, and flags

```
WITH ach_batch_totals AS (
    SELECT
        c.company_id,
        SUM(abcr.total_credit_entry_dollar_amount)
                                AS current_credit_total,
        SUM(abcr.total_debit_entry_dollar_amount)
                                AS current_debit_total
    FROM ach_records_type_5 AS art5
    INNER JOIN ach_records_type_8
                          AS art8 USING (ach_records_type_5_id)
    INNER JOIN ach_batch_headers
                          AS abh USING (ach_records_type_5_id)
    INNER JOIN ach_batch_control_records
                          AS abcr USING (ach_records_type_8_id)
    INNER JOIN companies AS c
              ON c.ach_company_id = abh.company_identification
             AND c.company_id = %s
    GROUP BY c.company_id
)
SELECT cl.company_limit_id,
       cl.daily_debit_limit,
       cl.daily_credit_limit,
       COALESCE(abt.current_credit_total, 0) AS current_credit_total,
       COALESCE(abt.current_debit_total, 0) AS current_debit_total,
```

Returns the current debit/credit totals; defaulting to zero when no batches are found

```
        CASE
            WHEN abt.current_credit_total > cl.daily_credit_limit
            THEN TRUE
            ELSE FALSE
        END AS daily_credit_exceeded,
        CASE
            WHEN abt.current_debit_total > cl.daily_debit_limit
            THEN TRUE
            ELSE FALSE
        END AS daily_debit_exceeded
FROM company_limits AS cl
LEFT JOIN ach_batch_totals AS abt USING (company_id)
WHERE cl.company_id = %s
```

Returns flags to indicate whether limits have been exceeded

There is just some minor housekeeping to create the API endpoint and the class that holds the response for the UI, but there is plenty of existing code to reference for these tasks. In the next section, we finish adding the company limits to the dashboard.

10.7.3 Displaying the company limits

As we saw in figure 10.5, the component itself is straightforward as we are just trying to convey to the user in the simplest terms whether the limits have been exceeded. We do want to add a little more visual appeal to the component by displaying the amount as red when it exceeds the limit and green otherwise. The following listing shows the basics of setting this feature up.

Listing 10.46 Company credits and debit limits

```
…
        axios.get<CompanyLimitsResponse>(`${apiUrl}/companies
➥/${companyId}/limits`, …
    return (
…
      <Title>Company Limits</Title>
      <Typography component="p" variant="h4">
         Company Credits:
      </Typography>
      <Typography component="p" variant="h6">
      <Typography component="span"
                  variant="h6"
                  style={{color: response.daily_credit_exceeded ?
                                                  'red' : 'green'}}>
         {formatCurrency(response.current_credit_total)}
      </Typography>
         {" of "}
         {formatCurrency(response.daily_credit_limit)}
      </Typography>
      <Typography component="p" variant="h4">
         Company Debits:
      </Typography>
      <Typography component="p" variant="h6">
      <Typography component="span"
                  variant="h6"
```

Colors the field red when the field has been exceeded

```
                    style={{color: response.daily_debit_exceeded ?
                                                    'red' : 'green'}}>
        {formatCurrency(response.current_debit_total)}
    </Typography>
        {" of "}
        {formatCurrency(response.daily_debit_limit)}
...
```

Colors the field green when the field is within range

The other aspects of the UI for the company limits already fall into place because we are using the existing logic for exceptions. So, that means that the file is automatically flagged as having an exception when viewing the dashboard, and the exception is returned to the DataGrid when viewing the exceptions. This does bring up a problem with viewing the exception for this company limit. We see the ACH batch control record (type 8), but we cannot easily tell which company it was dealing with. Even though we did save the batch header UUID as part of creating the exception, this was the first time we ever used it. It would be nice to update our code to include some batch header information when viewing an exception. Let's use this opportunity to make a few changes to the exceptions and exception detail screens.

First, the simplest change is to drop the column for the error code from the exceptions screen. From a developer's perspective, this move makes sense, but it can be confusing and does not add any value to the UX, especially since end-users have the description the code translates to.

Next, the details for the exception consisted of only the unparsed record, which used a generic alert component we created for other purposes. Now that we have some experience in creating a modal to display details as we did with the audit details in chapter 9, we can use those skills to build a more detailed screen. We can work on displaying the details we already have (we can move the error code to this screen if we want), as well as information we have not used yet, such as the recovery action. Figure 10.6 shows the exception details modal we ended up with.

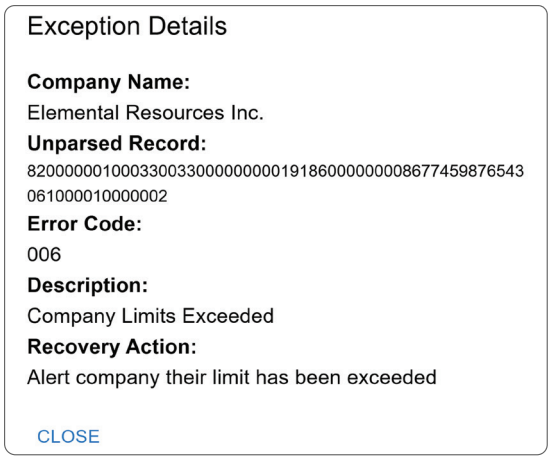


Figure 10.6 Exception details modal

The `AchExceptionDetailsResponse` will have to be updated so that it returns the required information—the company name if available and a recovery action. If we have access to the records, there is certainly an opportunity to return a lot more information, but in the interest of bringing back the YAGNI principle (chapter 3), and as a reminder of KISS (Keep It Simple, Stupid), we only return what we need at the moment.

The main work that we need to do is update the SQL statement for the `get_exception_detail_response`. We have to pull in extra tables to get the company name and recovery options. Keep in mind that we are not guaranteed to have recovery options nor are we guaranteed to have a company name. This means that we will want to avoid using an `INNER JOIN`. Instead, we will use a `LEFT JOIN` as in previous cases where we needed to keep the rows on one side of the query even when there were no matching records in the subsequent parts. The following listing shows the additional SQL logic that needs to be included. Note that since we have saved the parsed header record ID, we do not have to do a lot of joins to get to the record like in previous queries, where we had to start at the file level and drill down.

Listing 10.47 Updates to get more exception details

```
SELECT
       ae.ach_exceptions_id AS id,
       af.created_at AS created_at,
       ae.exception_code AS exception_code,
       aec.exception_description AS description,
       acr.unparsed_record AS unparsed_record,
       COALESCE(c.name, abh.company_name, '') AS company_name,
       aro.recovery_option AS recovery_option
FROM ach_exceptions AS ae
INNER JOIN ach_exception_codes AS aec USING (exception_code)
INNER JOIN ach_files AS af USING (ach_files_id)
INNER JOIN ach_combined_records AS acr
             ON ae.ach_files_id = acr.ach_files_id
             AND ae.record_number = acr.sequence_number
LEFT JOIN ach_recovery_options AS aro
                 USING (exception_code)
LEFT JOIN ach_batch_headers AS abh
                 USING(ach_records_type_5_id)
LEFT JOIN companies AS c
             ON abh.company_identification = c.ach_company_id
WHERE ae.ach_files_id = %s
AND ae.ach_exceptions_id = %s
```

These fields are new to our query and select the company name if available and any recovery steps.

Joins the new tables. Again, we need to use LEFT JOIN because we are not guaranteed data will exist in tables.

With the fields now included in our response, we can make the necessary updates to the UI to include them as well.

Listing 10.48 Using the `ExceptionsModal`

```
const [exceptionData, setExceptionData] =
        useState<AchExceptionDetailsResponse | null>(null);
```

```
…
            setExceptionData(response.data);

            <ExceptionsModal open={isOpen}
                             onClose={() => setIsOpen(false)}
                             exceptionData={exceptionData} />
…
```

Includes the ExceptionsModal and the required parameters

The `ExceptionsModal` takes care of formatting the `exceptionData` and is based on what we did before for our `AuditModal` in chapter 9. We will not display the company name with this component if it is an empty string, and we would like to provide a default recovery option of None Provided if no recovery options were available. Depending on the situation, it can often be beneficial to display a confirmation or message that validates this is an expected situation—sort of a 21st-century "This page intentionally left blank" message. It may make sense not to include the company name if the problem is not directly dealing with a company (although a case could be made for referencing the batch for any entry exceptions as well). However, hiding the recovery options may make users think something has gone wrong, especially when they may depend on referring to recovery options. The following listing shows the basics of the modal. Since the query to return the exception detail uses the `COALESCE` function, we are guaranteed to return at the very least an empty string for the company name, so we use that knowledge to check whether the label and field should be displayed. In addition, we use `??` to default the recovery text when nothing has been provided.

Listing 10.49 The `ExceptionsModal` Component

```
…
interface ExceptionModalProps {
  open: boolean;
  onClose: () => void;
  exceptionData: AchExceptionDetailsResponse | null;
}
…
const ExceptionsModal: FC<ExceptionModalProps> =
      ({ open, onClose, exceptionData }:
➥ Readonly<ExceptionModalProps>) => {
  return (
 …
        { exceptionData?.company_name !== '' &&
            <Box>
        <Typography id="audit-modal-description"
           sx={{ mt: 2 }}  fontWeight={'fontWeightBold'}>
          Company Name:
        </Typography>
        <Typography>{exceptionData?.company_name}</Typography>
        </Box>
        }
  …
```

Conditionally includes the company name if something was returned

```
        <Typography>
          {exceptionData?.recovery_option ?? 'None Provided'}
        </Typography>
      <Button onClick={onClose} sx={{ mt: 2 }}>
          Close
        </Button>
…
```

Always prints something for the recovery steps, even if it is just something to let users know they are on their own

Only slight adjustments to the Exceptions component are necessary to ensure that data is passed properly to the new modal. The following listing shows the updated Exceptions component. The biggest change here is simply passing a function that will set the open state to `False` when the modal button is clicked.

Listing 10.50 The Exceptions component

```
…
const [isOpen, setIsOpen] = useState(false);
    const [exceptionData, setExceptionData] =
              useState<AchExceptionDetailsResponse | null>(null);
        {field: 'file_name', headerName: 'Filename', width: 150},
        {field: 'created_at', headerName: 'Date', width: 150,
           valueGetter: (params) => convertDateFormat(params.value)},
        {field: 'record_number', headerName: 'Record Number', width: 150},
        {field: 'exception_code', headerName: 'Code', width: 10},
        {field: 'description', headerName: 'Description', width: 300},
…
            <ExceptionsModal open={isOpen}
onClose={() => setIsOpen(false)}
 exceptionData={exceptionData} />
…
```

No longer needed because it does not add anything useful for the user

Includes the ExceptionsModal

So, we have now built a company dashboard that provides information that can ensure correct/desired files are loaded, show recent batches for the company, and send alerts when activity does not match the company's expectations.

Summary

- Sprint planning in this chapter focused on building key components—company information, recent batches, expected files, and company limits—emphasizing the importance of comprehensive data-structure design.
- A company-level dashboard was created to address the need for storing and displaying detailed company information beyond the initial project scope.
- The importance of building robust company data management systems that provide a framework for enhancing ACH-processing services for business customers who need detailed batch information was highlighted.
- Designing foundational company information tables as blockers stressed their dependency on further development and the consideration of early work on some components to reduce modifications.

- Generative AI can help create meaningful test data quickly, which enhances productivity by generating more than typical name and address information, including logos.
- Implementing a company dashboard demonstrated the necessity of organizing UI structures for efficient navigation and detailed data presentation for specific company metrics.
- Creating detailed UI components for company overviews highlighted the value of user-friendly interfaces that facilitate seamless navigation and interaction at the company level.
- Designing an API for company data retrieval reinforced the importance of well-structured responses and validation mechanisms for dynamic user interaction with dashboards.
- Developing recent batch components highlighted the importance of flexible search functions for batch data retrieval to meet diverse business needs.
- Creating expected file components stressed the need for alerting mechanisms and schedule compliance to ensure timely processing of crucial documents and mitigate risks.
- Proactive monitoring of ACH files was explored to shift from reactive to proactive measures for improved customer service.
- Company limit components emphasized configurable alerts and thresholds to safeguard against unauthorized or unexpected financial transactions.
- Expanding exception functionality, such as alert generation, can help us notify customers through various channels if limits are exceeded.
- Utilizing Testcontainers enhances testing reliability by ensuring isolated environments, eliminating dependency on developer-specific configurations, and allowing robust and portable tests.

11 International ACH transactions and OFAC scanning

This chapter covers

- IAT batches
- Enhancing the project to process IAT batches
- The OFAC list
- Scanning ACH files to stay in compliance

This chapter covers the final sprint of our program iteration. Of course, the business unit will get together for another PI planning session, and the process will start all over again. In this sprint, we are tasked with enhancing the project to expand beyond domestic ACH transactions and deal with batches containing international ACH transactions (IAT), which enable sending money electronically between accounts in different countries. Along with IAT processing, there is the need to ensure the financial institution is not sending transactions to individuals or countries that are currently under restrictions or sanctioned by the US government. Specifically, the Office of Foreign Asset Control (OFAC) provides a list of "Specially Designated Nationals," also known as the SDN list, which is a register of individuals and companies whose assets are blocked, and dealing with them is prohibited.

11.1 Sprint planning

In this final sprint, we have a big request by the line of business, which is to enable IAT ACH transactions. In the original scope of our project, we were told that the financial institution was not going to take on the additional risk of processing international transactions. Therefore, the database was not designed to support these types of transactions. However, recently, the financial institution has been attempting to pursue larger business customers and has had trouble attracting them because such customers require the ability to receive and transfer funds internationally.

As it often happens when meeting customer demands, we now have to update the dashboard to be able to support these types of transactions, which means including additional tables to the database, parsing of the file, and scanning of the customers involved to stay in compliance. Figure 11.1 provides a timeline for the proposed tasks associated with this sprint. Having a timeline that we can provide to other stakeholders can be helpful when addressing questions about the tasks we will be working on during the sprint and our schedule.

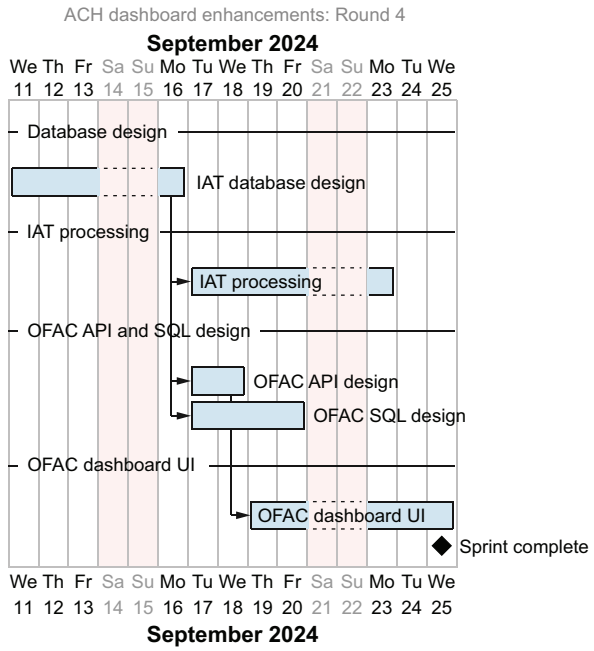


Figure 11.1 Timeline for IAT ACH transaction processing

With the sprint planning in place, we can move forward with work to support international ACH transactions.

11.2 International ACH transactions

Until this point, we have dealt with Prearranged Payment and Deposit Entry (PPD) batches. As you know, PPD batches are commonly used for direct deposits of payroll and pension payments. We may also remember that ACH started as a way of processing payments for domestic transactions in the early 1970s. With the expansion of the ACH system, support for IAT began in 2009, and the capabilities for ACH were expanded beyond domestic transactions. So, what is so different about IAT batches that we need to dedicate an entire sprint to add support for processing them? First and foremost, the IAT batches must conform to the 94-character limit imposed on all ACH records. As we will see in this section, this affects the way data must be transmitted.

11.2.1 IAT batches: An overview

Before we dive into supporting IAT batches, we first need to understand how a batch may be laid out. Figure 11.2 shows a sample batch we will work with to better understand the data involved. This batch represents an individual named Elena Santiago from Bilbao, Spain, sending a gift of $1.00 to a friend named David Wiliams from her account at Iberia Global Bank to his checking account at Futuristic FinTech.

```
101 990000013 691000134240712062 7A094101Futuristic Fintech     ASF APPLICATION SUPERVIXXXXXXXX
5220                FF3                US1234567890IATGIFT      USDUSD240712194112345678000001
6229900000130007             0000000100123                                1123456780000001
710WEB000000000000000100                        David Williams                        0000001
711ELENA SANTIAGO                  CALLE FICTICIA 123                                 0000001
712BILBAO*BIZKAIA\                 ES*48001\                                          0000001
713IBERIA GLOBAL BANK              0112345678                          US             0000001
714Futuristic Fintech              01247611456                         US             0000001
715               9999 NORTH BROADWAY                                                 0000001
716NEW YORK*NY\                     US*10001\                                          0000001
822000000100330033000000000000000000001001234567890                    123456780000001
900000100001000000001002670841300000000000000000000100
```

Figure 11.2 Sample IAT batch

That is a fair amount of information packed into the batch, so let's take a moment to unpack it (figure 11.3), and then we can jump into the code.

As shown in figure 11.4, the batch is marked as an IAT, with a Company Entry Description of GIFT. The FF3 is the Foreign Exchange Indicator, while the FF means fixed-to-fixed. The originated amount is the same as the one being received, and the 3 indicates that the Foreign Exchange Reference is spaces. The USDUSD represents the originating currency code (USD) and the destination currency code (USD), respectively.

The entry record contains the number of addenda records that were passed (figure 11.5). We expect to see seven (0007) addenda records passed, which also happen to be a type 7. Another important field is obviously the amount—the $1.00 that was sent in this case.

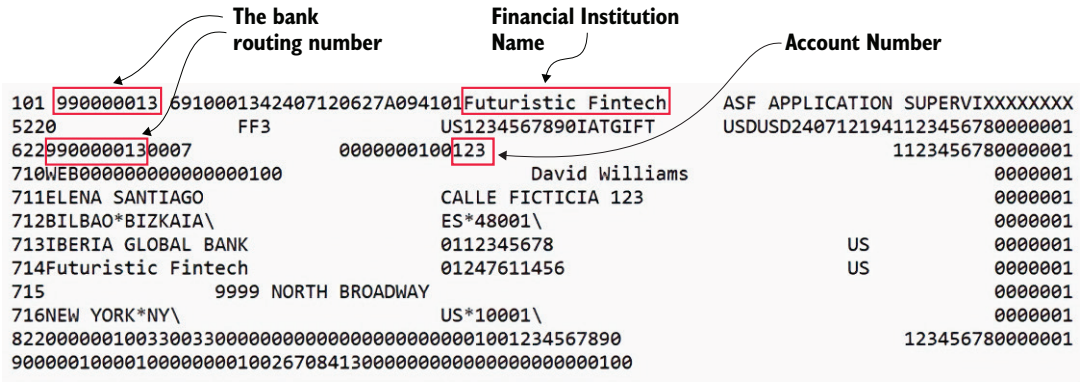


Figure 11.3 Routing number and account

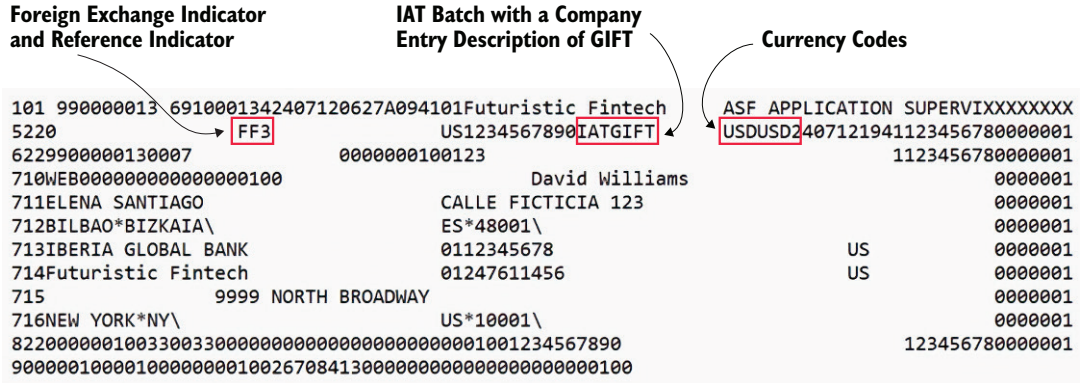


Figure 11.4 IAT batch header

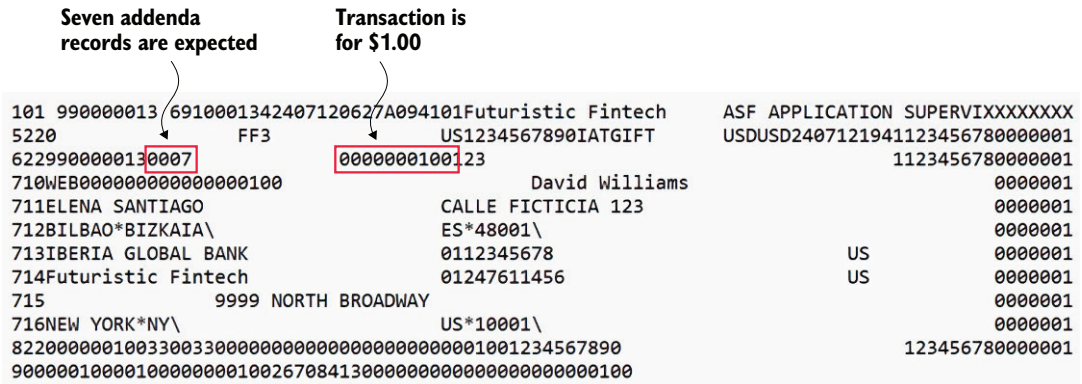


Figure 11.5 IAT entry record

There are between 7 and 12 addenda records per record when dealing with IAT entries. This is different from what we have dealt with when parsing PPD batches, as we had an indicator for an optional addenda record. While that addenda indicator is still present on an IAT entry, it will always be set to 1.

To keep things relatively simple, we will only consider the seven mandatory addenda records. They all start with a 7 to indicate an addenda record, followed by an addenda type code in the next two positions, producing the mandator records range from 710 to 716:

- *710*—Foreign payment amount and receiver's name
- *711*—Originator's name and street
- *712*—Originator's city, state, country, and postal code
- *713*—ODFI name, ID, and branch
- *714*—RDFI name, ID, and branch
- *715*—Receiver's ID number and street
- *716*—Receiver's city, state, country and postal code

Equipped with new knowledge, let's create some IAT batches.

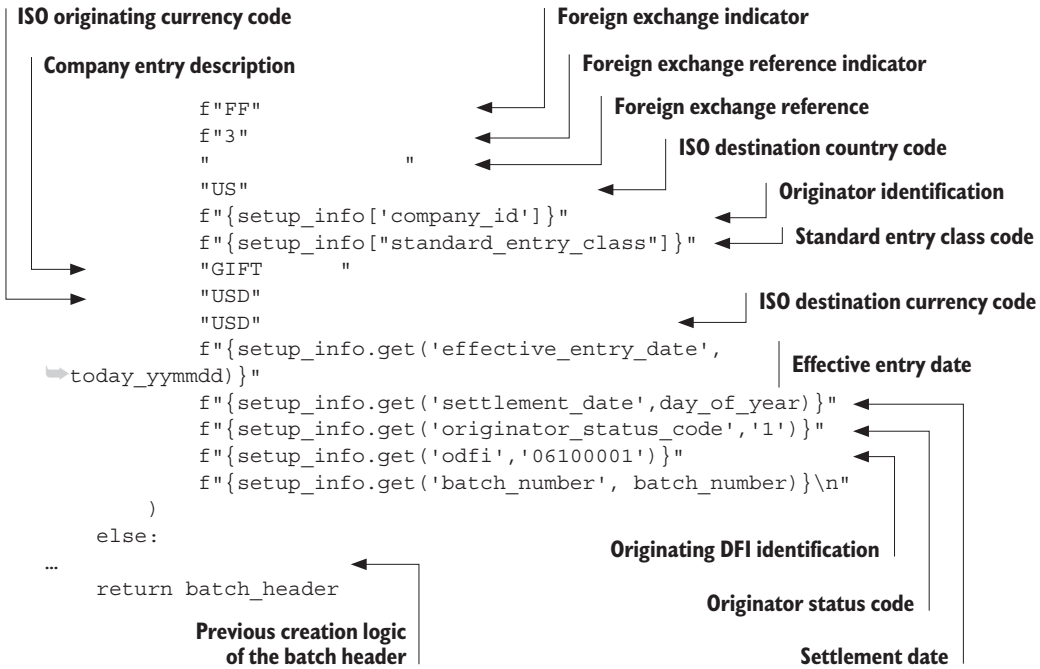
11.3 Creating IAT batches

Before we can start making changes and parsing a file, we must have a file. We have a sample provided in figure 11.2, and that, combined with the existing code we have to produce ACH files, should be more than enough to build from. If you reference the https://achdevguide.nacha.org/ach-file-details and https://mng.bz/eyOv as guides, you can see that we may be dealing with a few different layouts.

When we dig into the details, we'll find we are working with new records and formats. This means that when creating files specifically, we need to consider whether we have an IAT batch and create a file accordingly. We already have the code within our ach_file_creation.feature and test_create_ach_files.py to deal with creating batches with a specified Standard Entry Class (SEC) code such as the line `And I want to have 1 batch with ACH credits and debits and a standard entry class code of "PPD"`, which drives the creating of the batch. We must update our method `create_batch_header` to take the SEC code into consideration. So, as shown in the following listing, we start by defaulting (hardcoding) on some of the values. As usual, we go back when we need them to be dynamic and handle them at that point.

Listing 11.1 Updated `create_batch_header`

```
def create_batch_header(setup_info, batch_number):
…
    if setup_info["standard_entry_class"] == "IAT":
        batch_header = (
            f"5{setup_info['service_class_code']}"
            f"                "                    <-- IAT indicator
```



Since the SEC code is stored as part of our unit test information, we can reference it wherever necessary, and we will need it again when writing out the entries for an IAT batch. As described in the previous section, every entry record for an IAT record contains at least seven addenda records. We follow the same pattern, using the SEC code to create a new method that will take care of creating the needed entry and addenda records keeping things hardcoded for the most part and allowing for dynamic values. The code to create files is now over 500 lines long, and we should start to consider where refactoring is possible to clean it up. The logic to create specific types of files is a good candidate for refactoring since those details do not necessarily need to be part of this process.

For now, however, the code should suffice for our initial file creation. The feature in the following listing should look familiar to our other file-creation steps. This is a good thing as it means our grammar is generic enough to handle some variations. Of course, we have some work to do to really make things dynamic, but we have more than enough information to create the files.

Listing 11.2 Creating an ACH IAT file

```
Scenario: Create an ACH for for IAT
  Given I want to create an ACH file named "iat.ach"
  And I want to have an immediate destination of "990000013"
  And I want to have an immediate origin of "987654321"
  And I want to have 1 batch with ACH credits only
➥ and a standard entry class code of "IAT"
```

```
    And I want 1 entries per batch with random amounts between 100 and 100
    And I want to use individual names of "James Smith,
➥ Sarah Johnson, David Williams, Emma Martinez, Olivia Thomas"
    And I want to have company name "My Company"
➥ and company id "1234567890"
    When my ACH is created
    Then I should have a file of the same name
    And there should be 1 batch in the file
    And there should be 1 entries in the file
```

With an ACH file containing an IAT batch available, we can start working on the necessary tables and structure to support storing the file.

11.3.1 Database changes

We need to add a minimum of nine tables to the database to support IAT processing. We are only considering the new batch header format, entry format, and the required addenda records. The layout of our tables in the database mimics the layout of the ACH records. Let's take a look at some of the tables. Remember that our database structure relies on having both unparsed and parsed records, the idea being that our system will eventually expand to process the files asynchronously once they have been uploaded. Since no new record numbers have been introduced, we do not need to expand anything regarding the unparsed records as IAT will still fit nicely into that structure. If we want to have our parsed records stored (and we do), we need to add the nine tables.

Let's take a look at the `ach_iat_batch_headers` table in listing 11.3. Notice how it still has a foreign key reference to the `ach_records_type_5` table. Also, the majority of the fields are stored as `VARCHAR`, because as an initial iteration, we are looking to take a simple approach to the structure. In future iterations, fields such as `service_class_code` and `effective_entry_date` could be updated to `NUMERIC` or `DATE`, respectively. Those constraints will help ensure the integrity of the record and are worth dealing with.

Listing 11.3 IAT batch headers table

The rest of the fields required to support the parsing of the ACH record. Consider using more specific data types to ensure the record is formatted correctly by enforcing correct typing at the database level.

The foreign key for the batch header with CASCADEs for the deletes and updates

```
CREATE TABLE ach_iat_batch_headers
(
    ach_records_type_5_id          UUID UNIQUE NOT NULL
         REFERENCES ach_records_type_5 (ach_records_type_5_id)
                                          ON DELETE CASCADE
                                          ON UPDATE CASCADE,
    record_type_code               VARCHAR(1)  NOT NULL,
    service_class_code             VARCHAR(3)  NOT NULL,
    iat_indicator                  VARCHAR(16) NOT NULL,
```

```
    foreign_exchange_indicator      VARCHAR(2)  NOT NULL,
    foreign_exchange_ref_indicator  VARCHAR(1)  NOT NULL,
    foreign_exchange_reference      VARCHAR(15) NOT NULL,
    iso_destination_country_code    VARCHAR(2)  NOT NULL,
    originator_id                   VARCHAR(10) NOT NULL,
    standard_entry_class_code       VARCHAR(3)  NOT NULL,
    company_entry_description       VARCHAR(10) NOT NULL,
    iso_originating_currency_code   VARCHAR(3)  NOT NULL,
    iso_destination_currency_code   VARCHAR(3)  NOT NULL,
    effective_entry_date            VARCHAR(6)  NOT NULL,
    settlement_date                 VARCHAR(3)  NOT NULL,
    originator_status_code          VARCHAR(1)  NOT NULL,
    originating_dfi_identification  VARCHAR(8)  NOT NULL,
    batch_number                    NUMERIC(7)  NOT NULL
);
```

The rest of the fields required to support the parsing of the ACH record. Consider using more specific data types to ensure the record is formatted correctly by enforcing correct typing at the database level.

The rest of the tables are handled in much the same way, but we should be aware of a couple of design choices and standards we may need to consider when dealing with the addenda records. As an example, let's use the addenda record that contains the originator's city, state, country, and postal code. This addenda record has a record type code of 7 (because it is an addenda record) and an addenda type code of 12 (because that is what Nacha decided).

First, we need to name our tables. So, should we use the `ach_iat_originator_address_info` or `ach_iat_addenda_712_records` or some variation on that? In general, it does not matter unless we go with a name that is extreme (i.e., too long or too short and cryptic). We originally went with `ach_iat_addenda_712_records` because the 712 will be at the beginning of every line for those types of addenda records in the file, and we can key off that when trying to remember the name of our table. It also saves us from having to know what that type of addenda we are dealing with (i.e., whether this is the originator address or receiver address). Of course, there may be some complaints about the table name, such as

- It contains the word addenda and a 7 which is redundant, as a type 7 record is always an addenda
- It does not separate the 7 and 12 with an underscore even though they are two separate fields
- It uses the word records, and previously, tables using "records" contained unparsed records

We point these out to highlight the importance of consistency and standards. The closer we stay to established standards, the more consistent we are likely to be. Given that we wanted to try to adhere to a standard, we implemented the following:

- Use `_details` for tables that contained the parsed records
- Use `_records` for tables that contained the unparsed records
- Use the prefix `ach_ppd`, `ach_iat`, and so on for tables that dealt with specific ACH formats, as we were not always consistent with where the name `ppd` was used in the table

Thus, we ended up with names such as `ach_iat_entry_details` and `ach_iat_addenda_10_details`.

Next, let's tackle the table itself. Listing 11.4 shows the `ach_iat_addenda_712_records` table. Some of these IAT addenda records are unique to the ACH standards because they contain fields that are delimited within the fixed records. Strange, huh?

Previously, other formats were strictly a fixed-length record. So, we could say those 15 characters are the name and will be stored in the name field. With some of these addenda records, we have a fixed length field, such as the 35 character "Originator City & State/Province," which contains both the city and state. The data elements are delimited by an asterisk `*`, and the backslash `\` is the terminator for the last element. This results in a table that has seven fields (excluding the UUID), whereas the record has six fields. Note that one of the fields is reserved and not in use, and therefore, it is not represented in the table. Consequently, these parsed fields are represented in the table by individual fields. We left each individual field at the maximum field size, so although the originators city/state are both contained in one field of 35 characters, we kept both the city and state fields in the table at 35 bytes to avoid confusion.

Listing 11.4 Table for IAT addenda type 12

```
CREATE TABLE ach_iat_addenda_712_records
(
    ach_records_type_7_id        UUID UNIQUE NOT NULL
            REFERENCES ach_records_type_7 (ach_records_type_7_id)
            ON DELETE CASCADE ON UPDATE CASCADE,
    record_type_code             VARCHAR(1)  NOT NULL,
    addenda_type_code            NUMERIC(2)  NOT NULL DEFAULT 12,
    originator_city              VARCHAR(35),
    originator_state             VARCHAR(35),
    originator_country           VARCHAR(35),
    originator_postal_code       VARCHAR(35),
    entry_detail_sequence_number NUMERIC(7)  NOT NULL
);
```

Defaults the addenda_type_code to be a 12 as it must always be a 12 for this record

Leaves these records as VARCHAR(35), although we could consider enforcing a stricter data type

This code provides the pattern for how all the IAT tables were dealt with. Next, we look at how the records are parsed.

11.4 IAT record parsing

To parse the records, we use the file creation steps we built earlier to create a sample file and then use unit test to build the needed code to parse the various records and store them in the database (listing 11.5). We have our sample record that we are looking to parse—the expected result. We set up the necessary records for testing by calling `setup_iat_addenda_test`, which simply adds the needed headers and entry records to the database so that all the foreign keys work as expected. Then, we call the `_parse_iat_addenda_712` and the class `AchIat712AddendaSql`. It can sometimes

be tempting to simply define the `expected_result` based on the return value when we are sure our logic is correct. We would advise against that and make sure the record is parsed in another way, whether that is by hand or with another tool so that the results are independently verified.

Listing 11.5 Unit testing the parse routine for Type 12 addenda

```
class TestParsingIat712AddendaRecords:

    TABLE_NAME: str = "ach_iat_addenda_712_records"

    @pytest.fixture(autouse=True)
    def setup_teardown_method(self):
        SqlUtils.truncate_all()
        yield

    def test_parse_iat_addenda_712_records(self):
        sample_addenda_record = "712BILBAO*BIZKAIA\
➥                   ES*48001\
➥                  0000001"

record
        expected_result = {
            "record_type_code": Literal["7"],
            "addenda_type_code": 12,
            "originator_city": "BILBAO",
            "originator_state": "BIZKAIA",
            "originator_country": "ES",
            "originator_postal_code": "48001",
            "entry_detail_sequence_number": 1,
        }

        _, ach_records_type_7_id =
            SqlUtils.setup_iat_addenda_test(
               sample_addenda_record
            )

        parser = AchFileProcessor()
        parser._parse_iat_addenda_712(ach_records_type_7_id,
                                      sample_addenda_record)

        sql = AchIat712AddendaSql()
        retrieved_record =
           sql.get_record(ach_records_type_7_id).model_dump(
               exclude={"ach_records_type_7_id"}
           )

        assert SqlUtils.get_row_count_of_1(
            self.TABLE_NAME
        ), f"Expected 1 row in {self.TABLE_NAME}"
        assert (
            retrieved_record == expected_result
        ), f"Expected {expected_result},
➥ but got {retrieved_record}"
```

The fixture to ensure the database is empty; we have autouse set to True so there is no need to include it in our test methods as it will automatically be executed.

A sample addenda record taken from our test ACH file

The dictionary of the expected result to validate the retrieved record

Retrieves the record, excluding the UUID field. Since the UUID is assigned by the database, we cannot hardcode it in our expected results.

Sets up the needed database records so the constraints are satisfied

Parses the record, which will also add it to the database

Performs asserts to ensure there is only one row and that the record matches our expected value that was previously defined

The previous unit test will fail until we have built the required functionality. We will work through the required pieces of code to establish a pattern. All the unit tests for this will follow a similar approach. The first missing method that we encounter is the `_parse_iat_addenda_712` method, as in the following listing. Although it is straightforward, we do have to populate the `expected_record_types`, which helps the parser determine when records are out of sequence.

We want to keep verification in mind when we get to updating the processing logic for our ACH file because we will need to determine whether we have received all the required records and ensure that there are no duplicate addenda records.

Listing 11.6 The `_parse_iat_addenda_712` record

```
def _parse_iat_addenda_712(self, ach_records_type_7_id: UUID, line: str):
     self.expected_record_types = ["6", "7", "8"]

     ach_iat_addenda_record =
        AchRecordProcessor().parse_iat_addenda_712(
           ach_records_type_7_id, line
        )
        AchIat712AddendaSql()
           .insert_record(ach_iat_addenda_record)
```

The expected record types that can be passed to us

With the parsed record, we need to insert it into the database.

The parsing is actually done with the AchRecordProcessor. At this point, we may also consider moving some of our parsing routines to new classes.

Next, we create the method `parse_iat_addenda_712`. This task could also involve its own separate unit tests since the purpose of that class is twofold. First, it consolidates the actual parsing logic into a central location, which reduces the code in our ACH file processor and allows clearer understanding of the ACH processing flow. Second, it allows us to test the parsing logic in isolation without the need for a bunch of setup in the database.

However, the parsing is not overly complicated and will be tested by this overall process, so for the time being, we will not worry about it having its own unit test. The following listing shows the code for parsing the IAT addenda.

Listing 11.7 Parsing the IAT addenda record

```
def parse_iat_addenda_712(
        self, ach_records_type_7_id, line
    ) -> AchIat712AddendaSchema:
   regex = r"([^*]+)\*([^\\]+)\\"
```

Creates a regular expression that will parse the delimited fields within the record

```
    match = re.match(regex, line[3:38])
    if not match:
        raise ValueError("Error parsing originator
➥ city and state")
    originator_city, originator_state = match.groups()

    match = re.match(regex, line[38:73])
    if not match:
        raise ValueError("Error parsing originator country
                                and postal code")
    originator_country, originator_postal_code = match.groups()

    return AchIat712AddendaSchema(
        ach_records_type_7_id=ach_records_type_7_id,
        record_type_code=line[0],
        addenda_type_code=line[1:3],
        originator_city=originator_city.strip(),
        originator_state=originator_state.strip(),
        originator_country=originator_country.strip(),
        originator_postal_code=originator_postal_code.strip(),
        entry_detail_sequence_number=line[87:94],
    )
```

Ensures we have matches and extracts them for the city and state

Ensures we have matches and extracts them for the country and postal code

Parses the record and returns it as part of our schema

Moving on, we need to define the schema used for these records. As shown in listing 11.8, we provide a minimum layout that matches what we expect to insert into the database.

Listing 11.8 IAT addenda 712 schema

```
class AchIat712AddendaSchema(BaseModel):
    ach_records_type_7_id: UUID
    record_type_code: str = Literal["7"]
    addenda_type_code: int = Literal[12]
    originator_city: str = Field(..., max_length=35)
    originator_state: str = Field(..., max_length=35)
    originator_country: str = Field(..., max_length=35)
    originator_postal_code: str = Field(..., max_length=35)
    entry_detail_sequence_number: int = Field(..., ge=0)
```

The final step is to create a class to handle the SQL logic for inserting and retrieving the record.

Listing 11.9 IAT addenda type 12 SQL

Obtains a database connection

Defines a class and method to insert our record schema

```
class AchIat712AddendaSql:

    def insert_record(self,
                      ach_iat_addenda: AchIat712AddendaSchema):
        with get_db_connection() as conn:
```

```
            conn.execute(
                """
                   INSERT INTO ach_iat_addenda_712_records (
                            ach_records_type_7_id, addenda_type_code,
                            originator_city, originator_state,
                            originator_country, originator_postal_code,
                            entry_detail_sequence_number )
                         VALUES ( %(ach_records_type_7_id)s,
        %(addenda_type_code)s, %(originator_city)s,
        %(originator_state)s,
        %(originator_country)s, %(originator_postal_code)s,
        %(entry_detail_sequence_number)s)
                """,
                ach_iat_addenda.model_dump(),
            )
...
```

SQL to insert the fields and values for our model

Creates a dictionary of the schema to use with the INSERT statement

The rest of the methods, specifically getting a record by the UUID

With that, we should be able to go back and ensure all the import statements are in place and then run our unit test successfully. Assuming we are parsing the record correctly (for both the expected and the actual record), we should have a passing unit test.

> **Unit testing IAT challenge**
>
> This pattern needs to be repeated for all the new database records, and now is a great time to take a break and write some code. We have sample files. It is now just a matter of working through parsing the remaining records. If the previous code examples still seem a bit daunting, we could also start at a smaller scale by creating unit tests for `AchRecordProcessor`, which is solely responsible for parsing the ACH record with no database interaction. The simpler requirements for testing the `AchRecordParser` should mean that it takes less work to set up our unit tests. Once the parsing is verified at that level, we can take a step back to see the bigger picture and begin writing unit tests that involve the database, as outlined in this section.

While we should have all the parsing validated at this stage, we still need to update our `ach_file_processor` to handle the IAT batch and make use of all this wonderful code we just wrote.

11.5 IAT file processing

At this point, we have built the pieces of our IAT processing. We should feel fairly confident that we can handle the individual records and parsing them. Now, we need to be able to incorporate the parsing of an actual IAT batch within a file. We need to keep in mind that the work we did in previous sections tested individual pieces. For instance, we know that when we call the `_parse_iat_batch_header` method and pass it an IAT batch header record, it will be parsed and stored in the database. However, the

method is not called in the current flow of loading an ACH file through the POST call. As we work on adding the functionality to the parser, we should keep some goals and requirements in mind:

- Parsing a PPD batch still works as expected.
- Addenda records are all present.
- Addenda records are in the correct order.

In the next section, we begin ensuring that we have unit tests before we start making changes to the actual code.

11.5.1 Unit testing

We hope it is obvious that after adding the IAT batches, we should still be able to load PPD batches. So, do not forget to test and verify that we have not broken anything with the addition of IAT processing. This means that we want to ensure we implement some regression testing—the last thing we want to do is spend all our time working on the new processing and not validate the previous work.

With that in mind, the first order of business is to create a test_loading_ppd_files file and ensure that we have the correct record count when loading a PPD batch. We start with simple tests to get the count of the unparsed records. We test for the individual record counts and the total number of records, ensuring that we did not write any exceptions, as shown in the following listing.

Listing 11.10 Testing PPD batches

```
def test_good_unparsed_records_only(self, parser):
    filename = "ppd-mixed.ach"
    dir_path = os.path.dirname(os.path.realpath(__file__))
    file_path = os.path.join(dir_path, "data", filename)

    expected_exceptions_result: int = 0
    expected_total_records_result: int = 14

    ach_file_id = SqlUtils.
➥create_ach_file_record(filename, "123456789")

    parser.parse(ach_file_id, file_path)
    exceptions = SqlUtils.get_exceptions()

    with SqlUtils.get_db() as conn:
        record_count_type1 = conn.execute(
            "SELECT COUNT(*) FROM ach_records_type_1"
        ).fetchone()[0]
…
        record_count_type9 = conn.execute(
            "SELECT COUNT(*) FROM ach_records_type_9"
        ).fetchone()[0]
```

Defines a file name since it will be used in a few different places during the test

Ensures we can reference the file

Sets up some initial expected values

Sets up the test, parses the file, and returns any exceptions

Queries the database to get record counts for each of the tables that stores our unparsed records. This test contains a lot of repetitive code to obtain the count for each type.

```
        total_record_count = (
            record_count_type1
            + record_count_type5
            + record_count_type6
            + record_count_type7
            + record_count_type8
            + record_count_type9
        )
```

Adds the record counts together to get the total count, but it is somewhat redundant since we are testing the counts individually as well. However, how many times we would be asked about a total count if this was not here?

```
        assert record_count_type1 == 1,
                f"Expected 1, but got {record_count_type1}"
…
        assert (
            total_record_count == expected_total_records_result
        ), f"Expected {expected_total_records_result}, but got
{total_record_count}"
…
```

Asserts the record counts are correct

The previous test works well enough, but as it tends to happen with our development, we repurposed some other test to create the new one and ended up copying/pasting code to get the additional record counts and `assert` statement. In fact, Copilot is nice enough to fill in some of the code for us, so we did not even have to do much copying/pasting. There is one problem, though—we should be treating our test code as a first-class citizen, giving it the same attention we gave to our production code.

Let's look at how we might rework the above code to make it more concise and easier to understand. The changes in the following listing shortened the code from 53 to 45 lines, and while that is not the only metric, we should use it for judging whether code is good or bad. Early on in our careers, we were told by one of our mentors that it felt more productive when they were removing code rather than writing it.

Listing 11.11 Refactored unit test for unparsed records

```
with SqlUtils.get_db(row_factory=dict_row) as conn:
    record_counts = conn.execute(
    """
    SELECT
        record_count_type1,
        record_count_type5,
        record_count_type6,
        record_count_type7,
        record_count_type8,
        record_count_type9,
        record_count_type1 + record_count_type5 +
        record_count_type6 + record_count_type7 +
        record_count_type8 + record_count_type9
                             AStotal_record_count
    FROM (
     SELECT
       (SELECT COUNT(*) FROM ach_records_type_1)
                          AS record_count_type1,
       (SELECT COUNT(*) FROM ach_records_type_5)
```

Updates the get_db call to use a row_factory of dict_row

The record counts for each of our ACH tables

Computes the total records

The queries to get the counts of each row

```
                        AS record_count_type5,
       (SELECT COUNT(*) FROM ach_records_type_6)
                        AS record_count_type6,
       (SELECT COUNT(*) FROM ach_records_type_7)
                        AS record_count_type7,
       (SELECT COUNT(*) FROM ach_records_type_8)
                        AS record_count_type8,
       (SELECT COUNT(*) FROM ach_records_type_9)
AS record_count_type9
      ) AS counts
   """
   ).fetchone()
   record_counts["exception_count"] = len(exceptions)

assert expected_results == record_counts
```

The queries to get the counts of each row

Gets the single result

Adds the exception count into the record_counts dictionary

Compares the two dictionaries

We have a similar query to verify that parsed records exist in the table for all the parsed records as well. Ideally, we would expect an exception to be in the table if we had any problems parsing, but in case we have not yet coded for that or perhaps missed a condition that would cause a parse error, it is good to check these tables as well.

Listing 11.12 Unit test for parsed PPD records

```
SELECT
   record_count_type1,
   record_count_type5,
   record_count_type6,
   record_count_type7,
   record_count_type8,
   record_count_type9,
   record_count_type1 + record_count_type5 +
   record_count_type6 + record_count_type7 +
   record_count_type8 + record_count_type9
                                        AS total_record_count
FROM (
   SELECT
      (SELECT COUNT(*) FROM ach_file_headers)
                           AS record_count_type1,
      (SELECT COUNT(*) FROM ach_batch_headers)
                           AS record_count_type5,
      (SELECT COUNT(*) FROM ach_entry_ppd_details)
                                  AS record_count_type6,
      (SELECT COUNT(*) FROM ach_addenda_ppd_records)
                                  AS record_count_type7,
      (SELECT COUNT(*) FROM ach_batch_control_records)
                                  AS record_count_type8,
      (SELECT COUNT(*) FROM ach_file_control_records)
                                  AS record_count_type9
   ) AS counts
```

The selection logic can remain the same.

Note that the counts are now occurring on the parsed records and that the tables are specific to PPD batches.

Our goal was to ensure that we had enough tests to validate the files processed successfully. We should now be confident that the file has been processed in the database, that

their unit tests may validate those database fields, and that the files were parsed correctly. Remember from chapter 2 that we should test not only the happy path (success) but also the not-so-happy path (error handling). For now, we have enough to move forward with a similar test for IAT processing.

We can create a similar test for unparsed records that loads a file containing an IAT batch instead. We created one and named it (rather unimaginatively) iat.ach. The processing for an IAT file does not change when we consider only the unparsed records, and that is partially why we needed to test both the unparsed and parsed records. Of course, we still tested the unparsed records with a unit test, but the real work is for the IAT parsed records, as in the following listing.

Listing 11.13 Unit test for parsed IAT records

```
SELECT
   record_count_type1, record_count_type5, record_count_type6,
   record_count_type710, record_count_type711,
   record_count_type712, record_count_type713,
   record_count_type714, record_count_type715,
   record_count_type716, record_count_type8,
   record_count_type9, record_count_type1 +
   record_count_type5 + record_count_type6 +
   record_count_type710 + record_count_type711 +
   record_count_type712 + record_count_type713 +
   record_count_type714 + record_count_type715 +
   record_count_type716 + record_count_type8 +
                  record_count_type9 AS total_record_count
FROM (
   SELECT
      (SELECT COUNT(*) FROM ach_file_headers)
                                    AS record_count_type1,
      (SELECT COUNT(*) FROM ach_iat_batch_headers)
                                    AS record_count_type5,
      (SELECT COUNT(*) FROM ach_iat_entry_details)
                                    AS record_count_type6,
      (SELECT COUNT(*) FROM ach_iat_addenda_710_records)
                                    AS record_count_type710,
      (SELECT COUNT(*) FROM ach_iat_addenda_711_records)
                                    AS record_count_type711,
      (SELECT COUNT(*) FROM ach_iat_addenda_712_records)
                                    AS record_count_type712,
      (SELECT COUNT(*) FROM ach_iat_addenda_713_records)
                                    AS record_count_type713,
      (SELECT COUNT(*) FROM ach_iat_addenda_714_records)
                                    AS record_count_type714,
      (SELECT COUNT(*) FROM ach_iat_addenda_715_records)
                                    AS record_count_type715,
      (SELECT COUNT(*) FROM ach_iat_addenda_716_records)
                                    AS record_count_type716,
      (SELECT COUNT(*) FROM ach_batch_control_records)
                                    AS record_count_type8,
      (SELECT COUNT(*) FROM ach_file_control_records)
                                    AS record_count_type9
) AS record_counts
```

We select the desired records and sum them up to get the total count, which we also verified by pulling up the file (since it is just a text file) in any editor and reviewing the record counts.

We need to include the IAT addenda tables. Keep in mind the batch header, entry details, and addenda records will be different from PPD batches.

The previous test will fail because we have not yet updated the file parsing to make use of these detail tables. However, we are now confident that we have both PPD and IAT parsing covered by unit tests, which means we can move ahead to using these new IAT detail tables.

11.5.2 Updating file processing

With our passing PPD file load and the failing IAT file load, we begin working through the changes needed to support IAT. As we make changes, we will constantly be rerunning the unit tests to ensure that we are making progress with parsing IAT files and that we have not inadvertently broken anything. Since an ACH file is processed sequentially, the first record we need to address is the batch header record. We add a new field named `batch_type` that we will use when parsing the batch header. The following listing shows the simple `if/elif/else` processing that we have added, just calling the appropriate routine based on the SEC code and logging an exception when having a header we do not recognize.

Listing 11.14 Calling the appropriate parsing routine

```
if line[50:53] == "IAT":
    self._parse_iat_batch_header(ach_record_id, line)
    self.batch_type = "IAT"
elif line[50:53] == "PPD":
    self._parse_batch_header(ach_record_id, line)
    self.batch_type = "PPD"
else:
    self._add_exception(
        AchExceptionSchema(
            ach_files_id=ach_file_id,
            record_number=sequence_number,
            exception_code=AchExceptions.INVALID_BATCH_TYPE.value,
        ),
        line,
    )
```

The SEC code is in the same position, regardless of whether the batch header is for IAT or PPD. This line sets the batch_type flag appropriately for the SEC type.

Previously, we just called the _parse_batch_header method. Now, we ensure that we are dealing with a PPD batch.

We would like to log an exception when processing, and we have an unrecognized SEC code. This could be because of an invalid formatted file or a code we do not support yet.

With the proper parsing of the batch header, we should be making it a bit farther in our testing of IAT processing, and of course, PPD processing should not be broken. The next record we encounter is the type 6 record. With the introduction of the `batch_type` variable, it should follow the same pattern. Notice that we continue to write the unparsed record to the database like before. However, the need to parse the record and write it to the appropriate table depends on the `batch_type` flag. The following listing shows the required updates.

Listing 11.15 Parsing entry details

```
case "6":
    ach_record = AchRecordType6Schema(
        ach_records_type_5_id=current_batch_header_id,
        unparsed_record=line,
        sequence_number=sequence_number,
    )
    ach_record_id = AchRecordsSqlType6()
➥.insert_record(ach_record)
    if self.batch_type == "IAT":
        self._parse_iat_entry_detail(
            ach_file_id=ach_file_id,
            current_batch_header_id=current_batch_header_id,
            ach_records_type_6_id=ach_record_id,
            sequence_number=sequence_number,
            line=line,
        )
    elif self.batch_type == "PPD":
        self._parse_entry_ppd_detail(
            ach_file_id=ach_file_id,
            current_batch_header_id=current_batch_header_id,
            ach_records_type_6_id=ach_record_id,
            sequence_number=sequence_number,
            line=line,
        )
    else:
        self._add_exception(
            AchExceptionSchema(
                ach_files_id=ach_file_id,
                record_number=sequence_number,
                exception_code=AchExceptions.INVALID_SEC.value,
            ),
            line,
        )
```

The writing of our unparsed records is independent of the batch type we are working with.

When the batch_type is set to IAT, we will call the appropriate method to parse the entry.

The parameters for IAT and PPD parsing are exactly the same. Only the fields that need to be parsed are different.

Adds an exception when we encounter an unexpected batch_type

Parsing the addenda records is only slightly more complicated than parsing the batch header and entry detail records. This is because we now also deal with the different addenda record types being in the expected order. Similar to the `expected_record_type` variable, we also introduce an `expected_addenda_type` variable. Let's jump straight to the method where we parse a specific addenda type record, as shown in the following listing. Here, it is standard processing with the addition of the new `expected_addenda_type`. The next record we expect is another addenda record (type 7), and it should be of addenda type 11.

Listing 11.16 The `_parse_iat_addenda_710` method

The expected_record should be another addenda record.

```
def _parse_iat_addenda_710(self, ach_records_type_7_id: UUID, line: str):
     self.expected_record_types = ["7"]
```

```
        self.expected_addenda_type = "11"

        ach_iat_addenda_record = AchRecordProcessor().parse_iat_addenda_710(
            ach_records_type_7_id, line
        )
        AchIat710AddendaSql().insert_record(ach_iat_addenda_record)
```

The expected_addenda_type should be the next sequence as the records are required to be in a specific order. Note that we only expect one type of record, so there is no need for an array as with the expected_record_type.

All the addenda records will parse the same way, with the `expected_addenda_type` being set to the next record. But what happens when we get to the final addenda type (type 16)? The following listing shows how we reset our expected types once we reach that last record.

Listing 11.17 Resetting the expected types

There are some optional addenda records for IAT processing, so it is possible to encounter those, as well as a new entry or the end of the batch.

```
self.expected_record_types = ["6", "7", "8"]
self.expected_addenda_type = ""
```

There are no more mandatory addenda records, so the expected_addenda_type can be set to an empty string.

Now that we know how to parse the addenda records, it should be easy to put together how these individual methods would be called—as they follow a similar approach to how we parse the record types. The process is shown in the following listing.

Listing 11.18 Parsing the addenda records

Extracts the addenda type since we will be using it repeatedly

Defines our method for parsing. This is an umbrella that will call the other methods to parse specific records.

```
def _parse_iat_addenda(
    self, ach_records_type_7_id: UUID,
    line: str, sequence_number
):
    addenda_type = line[1:3]

    if addenda_type != self.expected_addenda_type:
        self._add_exception(
            AchExceptionSchema(
                ach_files_id=ach_records_type_7_id,
                record_number=sequence_number,
                exception_code=
                   AchExceptions.UNEXPECTED_ADDENDA_TYPE.value,
            )
```

If the addenda type is not what we expected, we need to log an exception.

```
            )
            Return

        if addenda_type == "10":
            self._parse_iat_addenda_710(ach_records_type_7_id, line)
…
        elif addenda_type == "16":
            self._parse_iat_addenda_716(ach_records_type_7_id, line)
        else:
            self._add_exception(
                AchExceptionSchema(
                    ach_files_id=ach_records_type_7_id,
                    record_number=sequence_number,
                    exception_code=
                        AchExceptions.INVALID_IAT_ADDENDA_TYPE.value,
                )
            )
```

Depending on the addenda type we are working with, calls the appropriate method. It logs a different exception if we have an unexpected addenda type.

With that, we should be able to parse files containing both IAT and PPD batches. Having these extra tables for IAT batches will have an effect on some of our APIs. In the next section, we will take a brief look at exactly what is affected.

11.6 Effects on the dashboard

How does the addition of IAT processing affect the functionality of our dashboard? That question would likely require more discussion with the business. From an ACH standards perspective, it is certainly possible to have IAT batches in the same file as PPD batches. However, the fields and interesting information for an IAT batch may be different, and certainly, the introduction of new tables plays into how the components function. Whether it makes sense for our components to include all batches or just select batches may be a business decision with input from end-users. For now, let's look at how we can fit the IAT batches into our current dashboard components.

11.6.1 The get_batches method

The `/{file_id}/batches` endpoint calls the `get_batches` method, which will need to be updated to incorporate the IAT batch headers. We know that the `ach_records_type_5_id` will exist in only one of the batch header tables. We employ that and the `COALESCE` command to use a field from the `ach_batch_headers` and if that is `NULL`, to use the `ach_iat_batch_headers` value instead. This works under the assumption that a field will not unexpectedly be `NULL`. The fields in question are all marked as `NOT NULL` in the database. Whether they stay that way indefinitely is another matter. For the time being, the following listing shows how we can update our query without being too invasive to the current API.

Listing 11.19 Updated `get_batches`

```
SELECT COALESCE(abh.company_name, '') AS company_name,
       COALESCE(abh.company_identification,
       aibh.originating_dfi_identification) AS company_id,
       art5.ach_records_type_5_id AS id,
       COALESCE(abh.batch_number, aibh.batch_number)
                                            AS batch_number,
       abcd.total_debit_entry_dollar_amount AS debit_total,
       abcd.total_credit_entry_dollar_amount AS credit_total,
       abcd.entry_addenda_count
  FROM ach_files AS af
…
LEFT JOIN ach_batch_headers AS abh
                USING (ach_records_type_5_id)
LEFT JOIN ach_iat_batch_headers AS aibh
                USING (ach_records_type_5_id)
    WHERE af.ach_files_id = %s
 ORDER BY COALESCE(abh.originating_dfi_identification,
                   aibh.originating_dfi_identification);
```

If the company_identification is not available, uses the originating_dfi_identification

We do not have access to a company name for IAT from the header record.

Picks the appropriate batch_number from the batch

Uses LEFT JOIN because the batch header is for one table or the other

Sorts by the appropriate identification number

As shown here, we can sometimes get away with updating the existing query; however, that is not the case in all situations. In fact, it is probably an exception. The next section discusses what to do when the query is too large or complicated to incorporate both PPD and IAT.

11.6.2 Batch entries

Gathering the batch entries from the endpoint `/{file_id}/batches/{batch_id}/entries` requires some work. The query to build the `AchBatchEntriesResponse` was sizeable, and it may make more sense to leave the current query in place and create a new query specific to IAT transaction entries. Therefore, we will take the `get_entries` method, move the existing query to a method `_get_ppd_entries`, and create a new `_get_iat_entries`. The results of the split are shown in the following listing.

Listing 11.20 Updated `get_entries`

```
@staticmethod
def get_entries(
    ach_file_id: UUID, ach_batch_id: UUID
    ) -> list[AchBatchEntriesResponse]:
    if AchFileSql._is_iat_batch(ach_file_id, ach_batch_id):
        return AchFileSql._get_iat_entries
➥(ach_file_id, ach_batch_id)
```

Method to return the IAT entries for a given file and batch

Method to determine whether the specified batch in the file is an IAT batch

```
    else:
        return AchFileSql._get_ppd_entries(ach_file_id,
                                           ach_batch_id)
```

The previous SQL query has been moved to its own method.

There are some slight changes when creating the `AchBatchEntriesResponse` for IAT records. With the original query for PPD records to get the addenda count, we had to gather that information ourselves. With an IAT batch, that information is part of the entry record; however, we still gather the count in the same fashion, in part to keep the queries similar, but more importantly, because we may want to perform verification on the field in the future, and this approach provides the mechanism to do that (listing 11.21).

Listing 11.21 Query from `_get_iat_entries`

```
WITH addenda_records_for_entry AS (
   SELECT art6.ach_records_type_6_id,
   COUNT(art7.*) AS addenda_count
   FROM ach_files AS af
…
   LEFT JOIN ach_records_type_7 AS art7 USING (ach_records_type_6_id)
   WHERE af.ach_files_id = %s
      AND art5.ach_records_type_5_id = %s
      GROUP BY (art6.ach_records_type_6_id)
)
SELECT art6.ach_records_type_6_id AS id,
       aied.transaction_code,
…
       aia10d.receiving_name AS individual_name,
       aied.amount,
       CONCAT(
          '**************',
          RIGHT(LPAD#D
➥(aied.foreign_receivers_account_number, 4, '0'), 4)
       ) AS account_number_last_4,
       arfe.addenda_count
…
      INNER JOIN ach_iat_entry_details AS aied
                 USING (ach_records_type_6_id)
      INNER JOIN addenda_records_for_entry AS arfe
                 USING (ach_records_type_6_id)
      INNER JOIN ach_iat_addenda_10_details AS aia10d
                 USING (ach_records_type_7_id)
…
```

We get the number of addenda records, which is also available on the record type 6 for IAT transactions. Eventually, we could use this approach to validate the addenda count that was passed.

We pass the receiving_name back as the individual_name to conform to our response, which needs to be retrieved from the addenda type 10 record.

Pulls the amount from the IAT entry record

The entry record also has the account number.

Joins the necessary IAT tables

Whether we incorporate IAT into the current dashboard or create brand new components is going to be a matter or preference by the business and users. One aspect that is not up for negotiation is remaining compliant with government regulations. The next section discusses how to address some regulatory concerns.

11.7 OFAC scanning

Financial institutions are subject to various regulations and compliance on a daily basis. Regulations such as the PATRIOT Act and Know Your Customer (KYC) hold the financial institution responsible for failing to report potential money laundering and terrorist-financing activities. The Office of Foreign Asset Control (OFAC) provides a "Specially Designated Nationals" list, or a SDN list, that includes individuals, companies, and assets such as ships/planes that are prohibited to do business with, and institutions may face penalties.

Note that there are other lists, as well as countries in general, that institutions are prohibited from doing business with. Doing business with any of these sanctioned entities can cost the bank significantly. A list of entities that violated US regulations and their associated fines is available at https://mng.bz/pKN8.

While there are third-party packages that specialize in this type of scanning, over the next few sections, we implement basic scanning for our dashboard to get a feel for the process. If you are interested in ML/AI and analytics, this is a great opportunity to dive in and build some comprehensive detection!

11.7.1 Sanctioned individuals and countries

There are XML files available for download from the OFAC website at https://sanctionslist.ofac.treas.gov/Home/SdnList and an online search tool as well at https://sanctionssearch.ofac.treas.gov/. Either of these is a great way to start familiarizing ourselves with the information available when working with sanctioned individuals.

For our purposes, we create a table that contains names and aliases for made-up individuals to avoid any potential problems with using actual data from the list. We also create a list of made-up countries we should also scan for. As we saw from the civil penalties, many times, companies find themselves in trouble just by doing business with companies/individuals in other countries. Listing 11.22 shows the `create` statements for tables to hold individual names as well as countries.

Listing 11.22 Creating tables for OFAC scanning

```
CREATE TABLE sdn_list
(
    sdn_id          UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    first_name      VARCHAR(255) NOT NULL,
    middle_name     VARCHAR(255) DEFAULT NULL,
    last_name       VARCHAR(255) NOT NULL,
    alias           VARCHAR(255) DEFAULT NULL,
    created_at      TIMESTAMP    NOT NULL DEFAULT NOW(),
    updated_at      TIMESTAMP    NOT NULL DEFAULT NOW()
);

CREATE TABLE sanctioned_countries
(
    sanctioned_country_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    country_name          VARCHAR(255) NOT NULL,
```

```
    country_code          VARCHAR(2)    NOT NULL,
    created_at            TIMESTAMP     NOT NULL DEFAULT NOW(),
    updated_at            TIMESTAMP     NOT NULL DEFAULT NOW()
);
```

Next, we populate the tables when the database is created, as shown in the following listing. Do not forget to add the `sdn_list` and `sanctioned_countries` tables to the `truncate_all` utilities method as you do not want this default data cleared during unit testing.

Listing 11.23 Populating the new tables

```
INSERT INTO
sdn_list(first_name, middle_name, last_name, alias)
VALUES ('Cash', DEFAULT, 'Steeler', 'Heister');
INSERT INTO
sdn_list(first_name, middle_name, last_name, alias)
VALUES ('Penny', DEFAULT, 'Pincher', 'Embezzler');
INSERT INTO
sdn_list(first_name, middle_name, last_name, alias)
VALUES ('Ben', 'E', 'Factor', '');
INSERT INTO
sdn_list(first_name, middle_name, last_name, alias)
VALUES ('Lou', DEFAULT, 'Pole', 'Evader');
INSERT INTO
sdn_list(first_name, middle_name, last_name, alias)
VALUES ('Mallory', DEFAULT, 'Practice', 'Biller');

INSERT INTO
sanctioned_countries(country_name, country_code)
VALUES ('Bribeland', 'BL');
INSERT INTO
sanctioned_countries(country_name, country_code)
VALUES ('Scamistan', 'SC');
INSERT INTO
sanctioned_countries(country_name, country_code)
VALUES ('Embezzlvania', 'EV');
INSERT INTO
sanctioned_countries(country_name, country_code)
VALUES ('Swindleland', 'SW');
INSERT INTO
sanctioned_countries(country_name, country_code)
VALUES ('Greedonia', 'GD');
```

With our data populated, we can now move on to building the API for scanning.

11.7.2 Scanning for individuals

Before we build an API, let's define the query that will return the results for the OFAC scan. For our needs, we will be scanning all our loaded files whenever someone clicks on the OFAC Scanning link in our UI. However, this feature will not be practical later when we have larger lists of suspects, more complex scanning algorithms, and more

files loaded. We are going to discuss some strategies to handle this problem, but for now, let's break down the query we used to gather the results.

As you know, the goal is to search for individual names on incoming ACH transactions and match them against known individuals on the lists provided by OFAC. Up until this point, when searching, we have used SQL keywords such as `ILIKE` and the `%` wildcard. Now, we get a little more advanced by introducing fuzzy matching to our query. By installing the `fuzzystrmatch` into our Postgres database, as in the following listing, we get additional search options.

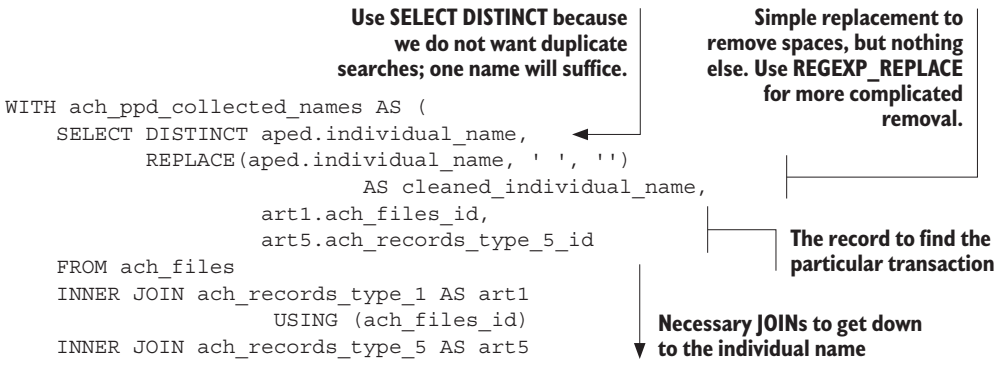
Listing 11.24 Installing `fuzzystrmatch`

```
CREATE EXTENSION IF NOT EXISTS "fuzzystrmatch";
```

This extension provides phonetic matching algorithms such as Soundex and Metaphone, as well as the Levenshtein distance algorithm, for measuring the similarity between two strings. These algorithms can help us create a more robust searching algorithm because, rather than searching for an exact or partial match, we can now expand our search to names that may by misspelt or slightly altered by a nefarious individual so that they can still perform transactions. Even though we are introducing feature for IAT transactions, it can be incorporated into any transactions done at a financial institution. Often, this (and various other checks) are done when a customer opens an account, and the database is often periodically scanned for any individuals. Therefore, we start with creating a common table expression (CTE) to gather names for PPD transactions. The main problem is to ensure we are collecting the names from the appropriate record, with ACH PPD transactions that will be on the type 6 record.

In the following listing, we want to only select distinct names to minimize some of the searching across batches if we have multiple names. We may want to consider additional information in case we have a false positive in regard to one customer but not another. We also used `REPLACE` to remove any spaces from the name, but could also use `REGEXP_REPLACE(aepd.individual_name, '[^a-zA-Z0-9]', '', 'g')` if we wanted to deal with any punctuation.

Listing 11.25 Collecting individual names from PPD transactions




```
WITH ach_ppd_collected_names AS (
    SELECT DISTINCT aped.individual_name,
           REPLACE(aped.individual_name, ' ', '')
                          AS cleaned_individual_name,
                  art1.ach_files_id,
                  art5.ach_records_type_5_id
    FROM ach_files
    INNER JOIN ach_records_type_1 AS art1
                    USING (ach_files_id)
    INNER JOIN ach_records_type_5 AS art5
```

```
                                USING (ach_records_type_1_id)
    INNER JOIN ach_records_type_6 AS art6
                                USING (ach_records_type_5_id)
    INNER JOIN ach_ppd_entry_details AS aped
                                USING (ach_records_type_6_id)
    GROUP BY art1.ach_files_id, art5.ach_records_type_5_id,
                     individual_name, cleaned_individual_name
),
```

Necessary GROUP BY statements because of the SELECT DISTINCT statement

A similar CTE is required for IAT names, as shown in the following listing. As for some of our other IAT processing, we need to retrieve the name from the addenda record instead of the entry record. We return the `receiving_name` as the individual name to maintain alignment between the CTEs.

Listing 11.26 Collecting individual names from IAT transactions

Simple replacement to remove spaces, but nothing else. Use REGEXP_REPLACE for more complicated removal.

At this time, we do not want duplicate searches because one name will suffice. We need to rename the field to individual_name or find a common name for the field.

```
ach_iat_collected_names AS (
    SELECT DISTINCT aia10d.receiving_name
                                        AS individual_name,
                    REPLACE(aia10d.receiving_name, ' ', '')
                           AS cleaned_individual_name,
                    art1.ach_files_id,
                    art5.ach_records_type_5_id
    FROM ach_files
    INNER JOIN ach_records_type_1 AS art1
               USING (ach_files_id)
    INNER JOIN ach_records_type_5 AS art5
               USING (ach_records_type_1_id)
    INNER JOIN ach_records_type_6 AS art6
               USING (ach_records_type_5_id)
    INNER JOIN ach_records_type_7 AS art7
               USING (ach_records_type_6_id)
    INNER JOIN ach_iat_addenda_10_details AS aia10d
                                        USING (ach_records_type_7_id)
    GROUP BY art1.ach_files_id, art5.ach_records_type_5_id,
             individual_name, cleaned_individual_name
)
```

The record to find the particular transaction

Necessary JOINs to get down to the individual name

Necessary GROUP BY statements because of the SELECT DISTINCT statement

The next CTE is used to collect the names of our suspects (listing 11.27). For display purposes, we handle concatenating the name together, keeping in mind the possibility of the `middle_name` being `NULL` with `CONCAT_WS`. Otherwise, using a standard `CONCAT` function will result in two blank spaces between the first and last names. Of course,

there are ways to handle this situation if the RDMS we are using does not have a function such as `CONCAT_WS`.

Listing 11.27 Collecting the SDN names

```
sdn_names AS (
    SELECT
        CONCAT_WS(' ', first_name, middle_name, last_name)
                                                AS sdn_name
        REPLACE(
           CONCAT(first_name, middle_name, last_name), ' ', '')
                                                 AS cleaned_sdn_name,
        alias,
        REPLACE(alias, ' ', '') as cleaned_sdn_alias
    FROM sdn_list
)
```

Concatenates the names and removes any spaces

The CONCAT_WS will skip NULL fields. It works perfectly for combining the first, middle, and last names into something we can use for display.

A similar approach for the alias

With the names of our customers and suspects gathered, we can now start comparing them. The first method we use is the function `LEVENSHTEIN` to compute the distance, which considers the number of additions, deletions, and updates needed to convert one string into another. We convert string into a number between 0 and 100, where 0 is no match and 100 is an exact match.

The other method is the `DAITCH_MOKOTOFF` function, used in listing 11.28. This function allows phonetic matching of names, attempting to determine if names sound alike. There are some additional algorithms available in the `fuzzystrmatch` module. In addition, there are other commercial searching algorithms. In a production environment, we would need additional pieces of information such as the alias, address, and similar to reduce the number of false positives that we may create by simply finding a phonetic match. However, using these methods is a good starting point.

Listing 11.28 Computing similarity scores for IAT individuals

```
computed_similarity_iat AS (
    SELECT
        ach_files_id,
        ach_records_type_5_id AS ach_batch_id,
        sdn.sdn_name,
        aicn.individual_name,
        alias,
        (1 - (LEVENSHTEIN(cleaned_individual_name,
sdn.cleaned_sdn_name)::FLOAT
/ GREATEST(LENGTH(cleaned_individual_name),
LENGTH(sdn.cleaned_sdn_name)))) * 100
AS similarity_score,
        CASE
           WHEN DAITCH_MOKOTOFF(cleaned_individual_name) &&
```

Converts the value returned by the LEVENSHTEIN function into a percentage that is easier to understand for end-users

Returns whether the names matched using the DAITCH_MOKOTOFF algorithm

```
            DAITCH_MOKOTOFF(sdn.cleaned_sdn_name) THEN TRUE
         ELSE FALSE
      END AS daitch_mokotoff_match_name,
      CASE
         WHEN daitch_mokotoff(cleaned_individual_name) &&
             DAITCH_MOKOTOFF (sdn.cleaned_sdn_alias) THEN TRUE
         ELSE FALSE
      END AS daitch_mokotoff_match_alias
  FROM ach_iat_collected_names aicn
  CROSS JOIN sdn_names sdn
```

We use CROSS JOIN so that every name is compared.

We select all results returned from both queries, as shown in the following listing.

Listing 11.29 Combining the results

```
computed_similarity AS (
    SELECT * FROM computed_similarity_ppd
    UNION ALL
    SELECT * FROM computed_similarity_iat
)
```

Pulls together all our computed scoring and matches

Finally, we filter the results as shown in the following listing, adding a unique row number for each row being returned to act as an identifier. There are many ways in which to optimize this and increase the performance of the queries. However, the clarity of gathering the names, comparing them, combining, and then filtering makes the process easier to follow in this example. We will discuss some strategies for actually producing this report when we discuss presenting of the OFAC results.

Listing 11.30 Filtering the results

```
SELECT ROW_NUMBER()
   OVER (ORDER BY ach_files_id, ach_batch_id) AS id, *
FROM computed_similarity
WHERE similarity_score >= 80
   OR daitch_mokotoff_match_name = TRUE
    OR daitch_mokotoff_match_alias = TRUE
ORDER BY similarity_score DESC
```

Use the ROW_NUMBER function to generate a unique id for the row.

Filters the results, returning only matches of a certain threshold or the ones that phonetically match the name or alias

We now have a good idea of what we will be returning based on the results we see when working with this query. We create a way for users to access these results in the next section.

11.8 Application programming interface

We now have two essential pieces: the data that exists in the database and a way to produce desired results. We continue to follow the well-established pattern for creating an API, starting with a unit test and working on the necessary pieces to ensure it passes.

11.8.1 Unit test

The unit test in the following listing loads one of our previously created files. Here we are concentrating on loading the data from a file called ofac_elemental_resources.ach and validating our API. Having previously created and loaded the database with the same file using the dashboard (or another unit test), we already know that our query should return three matches. So, the unit test simply needs to ensure the database is clean, load the file, and grab the results.

Listing 11.31 Unit test for OFAC API

```
class TestOfacApi:
    client: TestClient = TestClient(app)
    ach_files_id: Optional[str] = None

    # Get the directory of our file
    current_file_dir = Path(__file__).resolve().parent

    @pytest.fixture(autouse=True)
    def mock_client_host(self):
        with patch(
            "fastapi.Request.client",
            new_callable=lambda: type("Client", (),
                                      {"host": "127.0.0.1"}),
        ):
            Yield

    def get_absolute_path(self, relative_path):
        return self.current_file_dir / relative_path

    def setup_method(self, _method: Callable) -> None:
        ach_file = "ofac_elemental_resources.ach"
        absolute_path = self.get_absolute_path(
            Path("../data/ofac_elemental_resources.ach")
        )
        SqlUtils.truncate_all()
        self.ach_files_id = SqlUtils.create_ach_file_record(
            ach_file, str(randint(1, 99999999))
        )
        AchFileProcessor().parse
➥(self.ach_files_id, absolute_path)

    def test_get_ofac_api_for_ppd_batches(self):
        response = self.client.get("/api/v1/files/ofac")
        assert response.status_code == 200, response.text
        assert len(response.json()) == 3,
➥ "Should have 3 matches"
```

Ensures a host and IP address are passed with the client as our logging requires that

Ensures that we are picking the correct file

Prior to making our API call, we need to clear the database and load the file.

Once we have the database set up and the client ready to make a request, we can pull the OFAC results from the API and ensure they are correct. We are only doing a quick check of the number of items returned, but want to dive deeper into the results to ensure the expected data is returned.

Having the unit test defined allows us to continually check that our code returns the desired results. Let's make sure that happens!

11.8.2 Creating the endpoint

We first want to create the endpoint for the test (listing 11.32). We want to define the path users will be accessing. This endpoint was defined within the fiels.py router definition, which prefixes everything with `/api/v1/files`, so the path parameter only shows `/ofac`. Otherwise, we are simply returning an empty list, which should be enough to ensure that our unit test passes the first assert with a `200`-response code.

Listing 11.32 Barebones endpoint

```
@router.get(
    path="/ofac",
)
@log_message("Performed OFAC Scan on loaded ACH files")
async def read_files(request: Request):
    return []
```

Defines the path and a log message

Simply returns an empty list to ensure we receive a 200-response code and are able to pass the first assert statement in our unit test

After we have the barebones endpoint working, we can fill in the blanks with a return type and call an actual method to send back the results, as shown in the following listing.

Listing 11.33 Updated endpoint

```
@router.get(
    path="/ofac",
    response_model=list[OfacScanResults],
    summary="Scan for OFAC issues in loaded ACH Files",
    description=
➥"Perform an OFAC scan and return the results",
    response_description="Results of OFAC scan.",
    tags=["OFAC"],
)
@log_message("Performed OFAC Scan on loaded ACH files")
async def read_files(request: Request)
➥ -> list[OfacScanResults]:
    return OfacSql().get_scan_results()
```

Our API is documented and ready for users from one of our documentation endpoints.

We have defined the type of data we intend to return and a method to create it.

The endpoint will no longer work because of the newly added objects of `OfacScanResults` and `OfacSql`, which we will define next.

11.8.3 Finishing the API

With the API and the SQL to power it already defined, we only need to complete two housekeeping steps to finish the API and have a passing unit test. First, we want to

define the `OfacScanResults`, as shown in the following listing. This is the natural progression from the data returned in our `SQL` query to its Pydantic equivalent.

Listing 11.34 The `OfacScanResults` source

```
class OfacScanResults(BaseModel):
    id: int = Field(title="ID", description="The ID of the result")
    ach_files_id: UUID4 = Field(title="ACH Files ID",
                            description="The ACH Files ID")
    ach_batch_id: UUID4 = Field(title="ACH Batch ID",
                            description="The ACH Batch ID")
    sdn_name: str = Field(title="SDN Name",
➥ description="The SDN Name")
    individual_name: str = Field(
        title="Individual Name",
➥description="The Individual Name"
    )
    alias: str = Field(title="Alias",
                       description="The Alias from the SDN List")
    similarity_score: Decimal = Field(
        title="Similarity Score",
                       description="The Similarity Score"
    ) #C
    daitch_mokotoff_match_name: bool = Field(
        title="Daitch Mokotoff Match Name",
        description="Daitch Mokotoff Match Name"
    )
    daitch_mokotoff_match_alias: bool = Field(
        title="Daitch Mokotoff Match Alias",
        description="Daitch Mokotoff Match Alias"
    )
```

With these fields, users can jump directly to the file or batch that contained the match for further research.

The name/alias of the suspect and the name it matched from the ACH file

These Boolean values are intended to show a check mark or some other indicator that the record was matched by this method as well.

The numeric score that was computed. For now, it is the Levenshtein score but could potentially reflect a score computed from a more complicated algorithm.

With a place to store the results of our query, all that is needed is to create the `OfacSql` class and the associated `get_scan_results` method (listing 11.35). This will execute the SQL we created earlier, so this should be straightforward.

Listing 11.35 The `OfacSql` class

```
class OfacSql:
    def get_scan_results(self) -> list[OfacScanResults]:
        with get_db_connection(row_factory=class_row(OfacScanResults))
                                                                as conn:
            result = conn.execute(
                """
...
                """,
```

The SQL query we designed previously

```
        [],
    ).fetchall()          <-- Gets all the results from the query

return result             <-- Returns the results
```

At this point, the API should is functional, and we should have a passing unit test. We should also circle back and update our unit test to ensure that we are diving into the results and validating some of the fields as we were originally checking the size of an array. In theory, any endpoint that returned three items would pass that test. Next, we look at the final piece of the puzzle—presenting of our results.

11.9 User interface

With the data and the ability to retrieve it in place, the final step is to present the information to the user. Staying with our general approach, we create a page responsible for making an API call, the outcome of which is passed down to a component that will display formatted results. Figure 11.6 shows the result of pulling a sample report. The report provides the name of the suspect, their alias, and the customer's name. We also include the score (how close of a match) and whether the system matched the name or alias. There are also options to find the file or the batch that include the suspected customer.

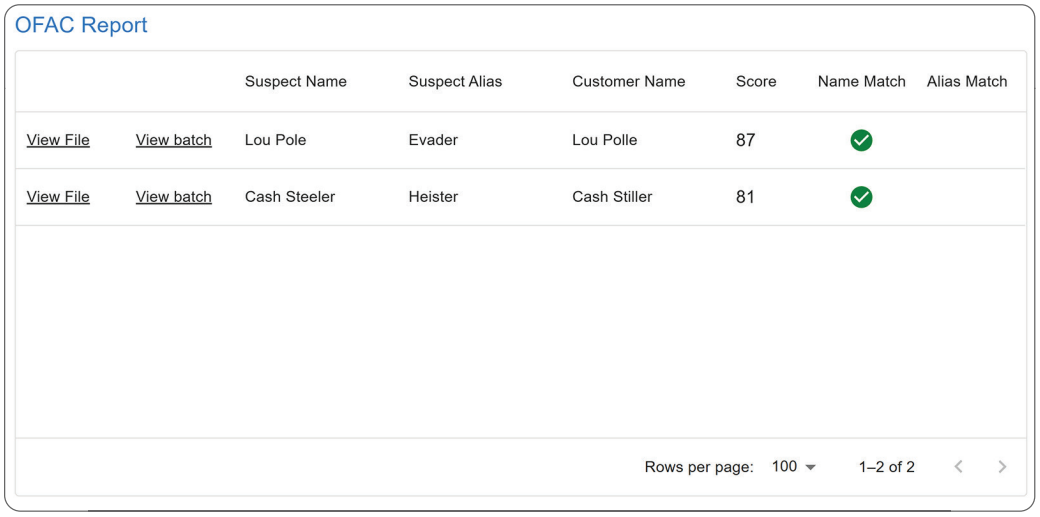


Figure 11.6 Sample OFAC report screen

11.9.1 The OFAC page

As shown in the following listing, we start by creating the main page and retrieving the data from the API. We could start by creating a placeholder page like in previous sections and ensuring that we can navigate to it before we try to populate the data. In this

instance, since we have used similar approaches for exceptions (chapter 9) and company information (chapter 10), we jump right to building the page. The IDE should complain because we do not have an `OfacResponse` or `OfacRecords` component, but we will tackle those next.

Listing 11.36 Sample page and API call

```
export default function OfacPage() {

    const [entries, setEntries] = useState<OfacResponse[]>([]);

    useEffect(() => {
        const apiUrl = process.env.NEXT_PUBLIC_API_URL ?? '';
        axios.get<OfacResponse[]>(`${apiUrl}/files/ofac`, {
            headers: {
                'Content-Type': 'application/json'
            }
        })
            .then(response => {
                setEntries(response.data);
            })
…
    return (
…
                    <OfacRecords records={entries}/>
…
    );
```

Uses axios to make the API request for an OFAC scan

Passes the results to our component to display them nicely for the user

As mentioned, this has become somewhat of a boilerplate process. Depending on the task at hand and how many pages need to be created, we may sometimes create those placeholders just to get a feel for the navigation and flow. Or we may choose to add pages as we begin working. We should be flexible in our approach.

11.9.2 OFAC components

We must create the `OfacResponse` to hold the API results. As we have worked with generative AI, we found that it can really help improve our productivity by generating some of this boilerplate code. We may use it to generate a Pydantic definition from a SQL `CREATE TABLE` statement. Then, we generate the TypeScript interface from the Pydantic class, which usually does a good job of capturing the needed fields. The following listing shows the `OfacResponse` interface.

Listing 11.37 The `OfacResponse` interface

```
export interface OfacResponse {
  id: number;
  ach_files_id: string;
  ach_batch_id: string;
  sdn_name: string;
  individual_name: string;
```

```
  alias: string | null;
  similarity_score: number;
  daitch_mokotoff_match_name: boolean;
  daitch_mokotoff_match_alias: boolean;
}
```

We now look at creating the component itself using a MUI DataGrid like we did for previous components in our dashboard.

Listing 11.38 The `OfacRecords` component

```
interface OfacRecordsProps {
    records: OfacResponse[];
}

export default function OfacRecords({records}:
➥Readonly<OfacRecordsProps>) {

    const columns: GridColDef[] = [
        {
            field: 'ach_files_id',
            headerName: '',
            width: 100,
            renderCell: (params) => (
                <Link href={`/fileDetails/${params.value}`}
                      color="inherit">
                    View File
                </Link>
            ),
        },
        {
            field: 'ach_batch_id',
            headerName: '',
            width: 100,
            renderCell: (params) => (
                <Link
href={`/fileDetails
       /${params.row.ach_files_id}/batchDetails
       /${params.value}`}
                      color="inherit">
                    View batch
                </Link>
            ),
        },
…
        {field: 'similarity_score',
                  headerName: 'Score', width: 75,
                  renderCell: (params) => (
                <Typography>{Math.floor(params.value)}</Typography>
            )},
        {
            field: 'daitch_mokotoff_match_name',
            headerName: 'Name Match',
            width: 100,
```

The ach_files_id is used to create a link users can click to navigate to the file where the match was.

To navigate to the batch that has the match, we need to pull the ach_files_id from the row object, as well as the params.value, although we could have also used the params.row.ach_batch_id field.

Displays the similarity score; we use Math.floor to remove any decimals just as a display preference.

Based on the Boolean flag that was passed, displays a green circle with a check mark or nothing

```
            renderCell: (params) => (
…
                    {params.value ?
                        <CheckCircle sx={{color: "green"}}/> : null}
…
        },
        {
            field: 'daitch_mokotoff_match_alias',
            headerName: 'Alias Match',
            width: 100,
            renderCell: (params) => (
…
                     {params.value ?
                          <CheckCircle sx={{color: "green"}}/> : null}
…
        },
    ];
…
                        <DataGrid columns={columns} rows={records}/>
…
```

Based on the Boolean flag that was passed, displays a green circle with a check mark or nothing

When the alias was matched, displays a green circle with a check mark

The columns and rows we defined are passed to the DataGrid componenet.

With the component created, we should be able to pull the results of an OFAC scan and display them in our component. That should be sufficient to meet the requirements for the story. In the next section, we wrap up the OFAC searching by outlining some potential problems.

11.9.3 OFAC results

We now have a component that can present the results of an OFAC scan and have met the requirements for our story. This will more than likely suffice for the short term, but in a production environment, we will have to address some of the design choices made.

First, the scan is done dynamically whenever the page is loaded. While this works great for limited data, the page will likely slow to a crawl in a production environment. The SDN list contains over 15,000 records, and each will be scanned for every ACH transaction we have. Remember that there is a large volume of payments flowing through the ACH system (currently over 31 billion payments per year), and while we will not be required to scan them all, apparently, it may result in a lot of scanning. The situation will also be exacerbated when we have multiple users trying to view the reports, since each page requests results in another scan.

Second, there is no way to save or export our results. Because the scanning is done upon request (when a user visits the page) using the data currently stored in the database, we do not have a way to pull previous scan results. The SDN and other lists provided by OFAC are revised constantly with individuals being added, removed, and updated. The scans we do today may yield different results tomorrow, which can be a compliance nightmare in the current setup. If the SDN list changes, and we can no longer show that a particular individual was matched on a certain date or transaction, we may find ourselves facing penalties, as outlined at the beginning of the chapter.

Third, in production environments, scanning is typically done at the time of the file load in an ACH environment. The database can also be periodically scanned to identify potential customers and transactions that may be in violation. All these scans produce a copy of the results that can be reviewed and archived in cold storage for later retrieval as needed. Scanning at the time of the load can help reduce constant requests that burden the system while providing real-time feedback on suspicious activity. However, for large files, this operation can still take some time, and with our current architecture, we would likely want to introduce a status for the file, so that some of these tasks could be done asynchronously.

Those are just a few of the challenges we may face with OFAC scanning, mentioning nothing of creating a more complex algorithm to detect suspicious activity. Although it presents several challenges, we can review them as opportunities to explore other areas of ACH, finance, compliance, and security that we need to touch on with this subject.

In this chapter, we explored a new ACH layout for international ACH transactions. The introduction of this new type of transaction for our dashboard also caused an increased risk for our organization by potentially violating US laws and regulations for dealing with sanctioned individuals. To remain in compliance and reduce our risk, we explored scanning our ACH file for customers that may be on a list of sanctioned individuals provided by OFAC. To accomplish all this, we enhanced the parsing of our ACH files, performed fuzzy matching on customer names, provided an API to power it all, and presented the results in a UI component.

Summary

- International ACH transactions (IAT) introduce unique regulatory requirements and require precise formatting, which underscores the need for careful planning and compliance verification in financial systems.
- Expanding database structures to accommodate IAT involves integrating new tables tailored for specific ACH components, reinforcing the importance of planning for system scalability.
- Generating IAT batches requires adapting current solutions to handle additional header and entry information, highlighting the complexity of international transaction standards.
- Comprehensive unit testing ensures new IAT functionalities align with existing PPD processes, while preventing disruptions, showcasing the importance of regression testing in software development.
- Streamlining code through refactoring enhances maintainability and reduces redundancy, indicating the continual need for codebase optimization as projects evolve.
- Integrating IAT processing in dashboards requires understanding both technical constraints and user needs, which emphasizes cross-functional alignment in feature development.

- Designing APIs to handle IAT data ensures consistent data retrieval and processing, demonstrating the necessity for flexible yet robust system interfaces.
- OFAC compliance is crucial when processing IAT to mitigate risks associated with sanctioned individuals, which highlights the role of regulatory adherence in transaction processing.
- Implementing fuzzy matching algorithms aids in detecting minor variations in suspect names, thus increasing accuracy in identifying sanctioned transactions.
- Effective UI components for OFAC results delivery underscore the value of clear, actionable insights for users navigating complex compliance landscapes.
- Persisting scan results addresses compliance demands by providing traceable audit trails, which stresses the need for reliable data storage and retrieval in financial applications.
- The addition of IAT processing elevates organizational risk awareness, which underscores the critical need for stringent compliance frameworks to navigate international regulations effectively.

12 Where to go from here

This chapter covers

- Expanding the dashboard
- Enhancing our ACH processing
- Diving further into FinTech

In this book, we worked through an entire program increment—starting from research spikes, we derived a minimum viable product (MVP) and enhanced the product based on feedback from our stakeholders. Along the way, we explored programming with Python using FastAPI to document and power our APIs. Furthermore, we used Next.js and Material UI to present the data stored in our Postgres database. Working on this project expanded our knowledge of FinTech and Automated Clearing House processing.

However, we only scratched the surface of what is possible. By processing ACH transactions, we were able to drive a lot of interesting features you are likely to encounter on a regular basis. In other words, the project is now at a point where you can expand it to different areas of software development and finance. This chapter shares some ideas you can use to improve the project, and figure 12.1 shows an overview of various possibilities.

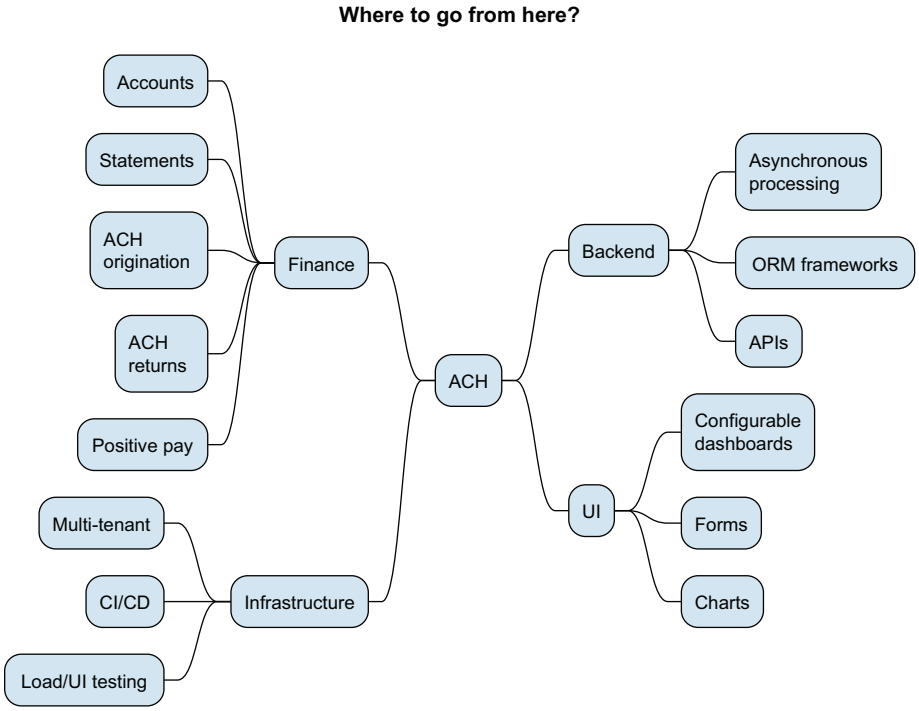


Figure 12.1 Expanding our project

12.1 Backend processing

We built the backend processing using Python, FastAPI Framework, and a database running Postgres. While we had a chance to touch on all those areas, we did not really examine them in depth, but now it's a perfect time for you to investigate some of the possibilities we skipped or were not able to cover. This section touches on some areas of the backend that you may find both interesting and valuable.

12.1.1 Asynchronous processing

You should have a better understanding of ACH processing at this point, so let's investigate some possibilities for enhancing the parsing of uploaded files. When uploading a file, we currently must wait for the file to load and process before a response is returned. For large files, this is simply not going to be a nice experience as the process may time out and users may not realize the file is still loading in the background. This downside may result in users trying to process the file a second, third, or fourth time. Depending on the safeguards we have in place, it could very well lead to duplicate transactions being posted to accounts if there is no way to identify the file is the same (such as computing a hash).

We need to worry not only about large files, but also about the ancillary processing that goes along with file loading. We saw a piece of this with OFAC scanning and how we would likely need to integrate scans into the processing of an ACH file itself. You would also need to post the transactions to the account, debiting or crediting the account as appropriate. When OFAC scanning and the accounts are separate services or require additional API calls, you will see a degradation of the user experience when larger files or remote services are involved.

One way to deal with this problem is to introduce asynchronous processing to the file load and then process the file in steps. By introducing this type of processing, you'll also have additional opportunities to improve the user experience. Imagine that a user loads a file. As soon as you can confirm the file is uploaded successfully, you can respond to the user at that point. The file can then be processed and the user alerted through a message on the dashboard, an email, or text. To implement this feature, you will need to introduce a file status and track the file through the various stages.

12.1.2 ORM frameworks

An object-relational mapper (ORM) allows mapping objects to a database and can simplify the interactions between our classes and the database. ORMs such as SQL-Alchemy and Peewee are popular options for Python and work with a variety of databases, including our Postgres database. So, why didn't we use an ORM?

We believe that ORMs are very handy, but the level of abstraction that comes with them gives them a learning curve. Of course, learning pure SQL has its own learning curve! Now that you have learned some SQL concepts and created a fair number of queries, it's a great time to see whether you could drop in an ORM and replace some of the handwritten SQL.

12.1.3 Application programming interfaces

RESTful APIs are ubiquitous in the industry at this point, and FastAPI gave us a great way to build APIs in our application. It also allowed us to easily generate OpenAPI documentation, so there were plenty of opportunities to learn about API development. We also took a brief look at API managers such as WSO2.

We built enough APIs to demonstrate the ACH processing and some dashboard functionality, but to have a fully functioning dashboard, we would need to generate additional APIs and functionality. For instance, the dashboard should provide the ability to add/update/delete companies, delete uploaded files, and undo anything related to the deleted file (e.g., OFAC-related scans). You could expand or enhance the current APIs in plenty of places in the project.

Another option would be to start incorporating GraphQL. While RESTful APIs are still the predominant API standard, there is still a place for GraphQL. Regardless of whether we want to migrate away from REST or use it in conjunction with our RESTful APIs, GraphQL is an interesting option worth exploring. If nothing else, you can gain

some hands-on experience, so that you have an informed opinion when another team wants to implement it.

12.2 Infrastructure

Most companies and projects have a robust infrastructure—it supports the long-term sustainability of the project. For example, projects need to be built, packaged, and deployed in some form. This process may require code merges, reviews, and scanning for vulnerabilities or just ensuring the code meets existing standards. In this section, we look at how to improve the infrastructure of the ACH dashboard.

12.2.1 Docker

With the ACH dashboard, we relied heavily on Docker, not just because container technology is important to understand and get comfortable with, but also because it is useful when exploring other technologies and products.

Companies such as Atlassian have Docker images that allow us to run their products, which facilitates evaluating them in a sandbox environment. We saw how easy it was to get Postgres up and running with a Docker image. In addition, the WSO2 API Manager was also available to download from Docker. You should keep in mind that whenever you want to explore something new, there may be a container that lets you start working quickly. Whether it is an official image from Docker on Golang, MongoDB, or a Debian Linux distribution, you can explore a wealth of things even without installing a bunch of products on your machine—just use a container and get rid of it once you are done.

12.2.2 Multitenant environment

Although there are a few flavors of multitenant environments, we consider that a multitenant environment involves two distinct entities (financial institutions) sharing the same database table. We only considered this application being used by Futuristic FinTech. But what if you wanted or needed to support additional financial institutions? This can be tackled in several ways. First, since the application runs as separate services, you could simply look at spinning up additional containers to handle the new financial institution. You should be able to do that without any additional setup. Each financial institution would be running on their own instance, and you would have different DNS names pointing to those locations, such as http://bankA.dashboard.com or http://bankB.dashboard.com.

Another option would be to update the database so that the tables also utilize a tenant ID. which could be the ABA number for the bank or some other identifier. This option allows the bank to exist in the same database. The identifier does not necessarily need to be added to all the tables (e.g., the individual ACH record tables do not need an identifier because they are dependent on the ACH file table), meaning the ACH file itself can be tied to a particular bank and then all the records within that file will

automatically be associated with that bank. However, you would need to review the queries to ensure that it is not possible to pull bank data using a UUID of another bank.

There are pros and cons to using a single database versus multiple databases. Depending on the industry, company, and governing regulations, customer may be required to be isolated in databases and even restricted to using their own hardware. However, adding other banks that can use the dashboard with their own routing numbers, companies, and similar, would let you explore different types of architectures and a chance to address these types of problems.

12.2.3 Continuous integration and delivery

Our project used several containers running under Docker. This approach worked well enough for our needs, especially with running our database in a container. We were able to bring the container down and start with a fresh database as needed. A challenge that we encountered was taking our project from our IDE or a source control system such as GitHub and deploying it to those containers because that is currently a manual process.

A great direction to explore (especially if you have not had much opportunity to do so) is to look at various ways to automate those tasks. This gets you moving in a continuous integration (CI) and continuous deployment (CD) direction. For instance, you could automate moving and/or building the code into the appropriate containers from your IDE. You could also create an instance of a CI/CD server such as Jenkins and learn to build and deploy the code from there. Furthermore, you could also use GitHub Actions to deploy the code, although with the current infrastructure setup to be running in a local Docker container, using a local Jenkins server may be easier.

A build server such as Jenkins allows you to automate other tasks as part of your build, such as running unit tests, scanning for security vulnerabilities, and performing a slew of other tasks. You are likely to find these types of build servers at companies of any size and in any industry.

12.2.4 Testing

We tried to introduce several methods of testing the application, drawing mostly on TDD and BDD. We did not have a chance to revisit load testing of our APIs using Locust after it was introduced. We also did not do too much UI testing, besides a few tests in Jest and Playwright.

Expanding the tests and reviewing the test coverage would be a great way to familiarize yourself with the testing process. You could also ensure the tests are robust enough to work when being run through a CI/CD server. A lot can be learned from writing tests and ensuring that they run through a release pipeline.

12.2.5 Database

We used Postgres as our database of choice, but you are certainly not limited to using Postgres. Changing the database can be tied to exploring ORM frameworks as they

provide a level of abstraction that can be helpful when the underlying database needs to be swapped. If the business has another vendor or database product they prefer, you often must go with that database. If you are interested in selling a product and allowing customers to install it in their environment, having flexibility in the database choice can be a big selling point. If a company uses Oracle database and your product requires a Postgres database, customers may not be interested in the overhead associated with managing another database product.

An alternative is not just to swap out the RDMS from Postgres to some other vendor but also to consider exploring a NoSQL database such as MongoDB. While Postgres was a great choice for storing the data for the ACH dashboard, it may be interesting to start moving some of the tables over to MongoDB. You could begin with something small, such as the `bank_routing_numbers` table and then look for other tables that can be moved. Do you need MongoDB for the ACH dashboard? No, but you already understand the table structure and data that may make exploring and transitioning to a new database interesting.

12.3 User interface

A lot can be done from the UI perspective if you want to explore the other end of development. Throughout the project, we used React and the Next.js framework to build the interface. You can use other React frameworks or switch gears and use Angular or even plain old HTML, CSS, and JavaScript! It all depends on where your interests and needs lie.

12.3.1 Charts and forms

We used Recharts to create some scatterplots, bar graphs, and pie charts. However, we did not go in depth, and you could certainly work on adding new charts and visualizing other aspects of the ACH files. Figures 12.2 and 12.3 show two sample charts, visualizing ACH files over a period of time. As the current charts focus on the existing contents of the files, they would be a great addition for our users.

You could also let users choose the type of chart they want to see the data displayed in. It would also be nice to have some interactivity added to the charts, such as dynamically adjusting the data or navigating to other pages in response to user actions. By adding additional charts and interactivity, you may improve the user experience and make the dashboard even more practical.

Another aspect of the dashboard that could use some updating is the basic management of information through using forms. For instance, the company information was displayed, but it was not possible to update it. You could enable updating, perform validation, or try different methods of presenting the data to the user. We also used a lot of DataGrid elements to display our data. You may continue to use those or design other components to display data. This may lead you to explore concepts such as those in *Data-Oriented Programming* (2022, Manning) by Yehonathan Sharvit.

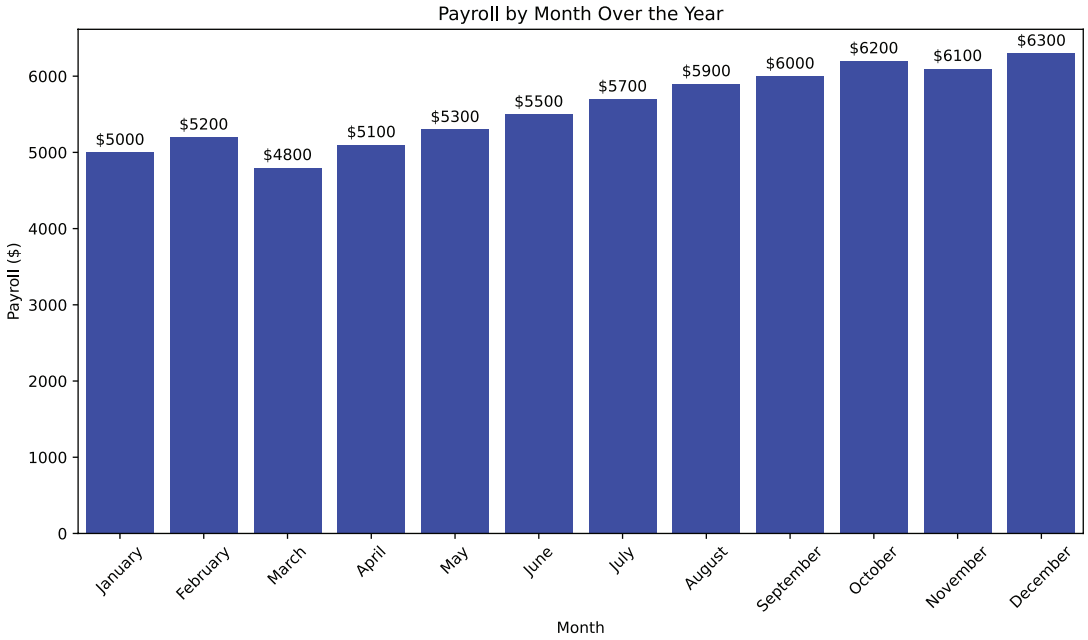


Figure 12.2 Payroll by month over the year

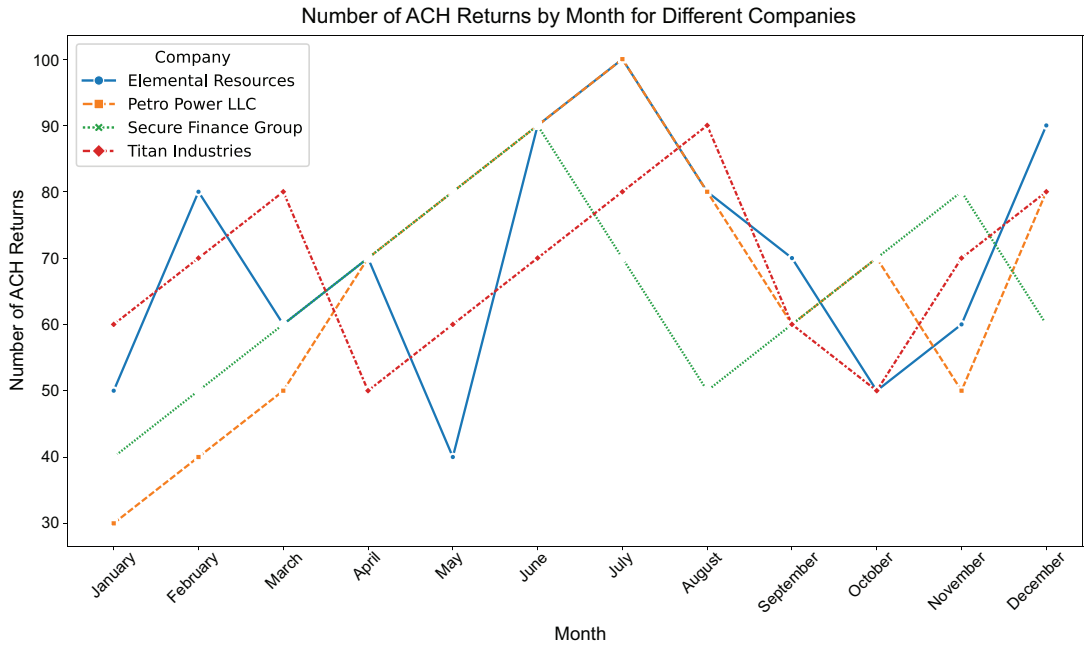


Figure 12.3 ACH returns by company

12.3.2 Configurable dashboards

You saw that there are at least three levels of progression (and possibly more) when it comes to configurable dashboards. You could create some more prepackaged components that users could drag onto the dashboard screen to configure what they want to see when they view the main dashboard and company level. While that would give users more flexibility and could greatly benefit them, we can provide them with even more customization.

Instead of choosing from prepackaged components, what if you allowed them to customize and configure their own components? By allowing the user to pick the fields that are being displayed and generating charts based on that information, you give them finer control over the UI.

The next progression would be to allow users to write their own components using an API provided by the dashboard or backend. Products such as Jenkins or the Atlassian suite let third parties to write tools that extend the product functionality. Imagine the ACH dashboard UI had a marketplace where custom user components are integrated into the dashboard using the type of customization we previously described. Perhaps users want to publish messages to Microsoft Teams or a Slack/Discord channel to alert when an ACH file is loaded, an exception occurred processing a file, or there is a transaction for a VIP customer.

12.3.3 Mobile version

We used APIs to ensure the backend and the UI were separated. Not only is using an API a great way to separate components, but it also lets you create other ways (or channels) for users to communicate with the application. For instance, you could use React Native, Dart, Swift, Kotlin, or anything you desired to design a mobile version of the dashboard.

Another approach for a mobile application would be to allow customers (companies) to create batches and enter transactions for the ACH system right from their Mobile App! So, even when dealing with the ACH and FinTech, we can still work in this area as well.

12.4 Finance

If we already have a lot of knowledge in the technical side of things and want to continue to expand the capabilities of the application, there are numerous possibilities for that as well.

12.4.1 Accounts and customers

One of the first tasks we would suggest undertaking is the addition of accounts to the application. Whether they exist in the same database or as a separate service is a decision that can be made as part of the overall design. The addition of accounts brings up a few possibilities, not the least of which is that if you have account numbers, those accounts must belong to customers. The relationship between customers and accounts

is no trivial matter, and you can look at handling relationships on accounts such as the primary/secondary owners or authorized signers, to name a few.

With accounts in the database (or posted via an API to a core banking product), the ACH transactions can be posted to the account. This has a cascading effect of causing you to handle insufficient fund conditions (also known as NSF), errors with the wrong or incorrect account number, interest accruals, and statements.

When an account has insufficient funds to cover a transaction, this can create a whole new world of transfers for accounts. You may have overdraft protection on the account or may transfer funds from another account to cover the charges. Furthermore, you may want to look at scheduled transfer or bill payments.

When a transaction does not post because of insufficient funds or an invalid account number or any number of other conditions, it may be returned or unposted. The way you need to deal with returned or unposted items may vary from one customer to another, as well as with the circumstances of the situation.

All of this account activity—such as transactions posting, overdraft transfers taking place and associated fees—must be reflected on the customer's monthly statement. These statements and notices may be mailed via snail mail, email, or available online for viewing.

Finally, most of what is previously done also needs to be reversed. For instance, if a transaction posts multiple times to an account because of some error, you need to be able to safely reverse that debit or credit. This could also cause a cascading effect of having to undo the transfer for an overdraft, as well as any associated fees the account may have incurred due to the erroneous transaction postings.

12.4.2 ACH origination and returns

ACH origination is the process of creating ACH transactions such as credits and debits that will be processed by the ACH network. In our project, we focused on receiving and processing ACH files within the dashboard. Financial institutions may originate ACH files to help facilitate movement of funds between the bank and their customers. We kind of touched on the process of originating files with the creation of our ACH files for testing. However, with the addition of companies to our dashboard, some additional opportunities for development arise when it comes to originating files. The companies that have defined the database may want to originate files for paying their employees (payroll) or other businesses/companies. You would need to provide the ability to create batches with transaction entries/addenda as necessary. The financial institution would then send these batches on behalf of the company.

Another aspect of ACH processing is the need to process returns. An ACH return happens when the financial institution rejects the transaction. Examples of why this might happen are incorrect account information, insufficient funds, or a stop payment on the transaction. With an ACH return, you would be returning the funds to an account they came from and possibly charging a fee.

By adding the functionality for ACH origination and handling of returns, you are beginning to build a more complete ACH-processing system.

12.4.3 Positive Pay

Another feature you could consider is building a Positive Pay system. Positive Pay is a system of fraud prevention originally created for check processing, but it has been expanded to other transaction processing areas as well. Basically, with Positive Pay customers would enter transactions on the dashboard or upload a list of expected transactions through the dashboard. The system would then check fields such as date, dollar amount, account number, names, and similar against the company-provided data to ensure transactions were added.

When transactions coming into the system do not match an item, they are flagged and rejected for processing. Basically, this is meant to prevent unauthorized transactions from being processed by the bank, reducing risk and fraud for both the company and the bank. Positive Pay is often offered to commercial customers, but in theory, it could be used by any customer.

12.4.4 Service charges

This is a broad area where customers may be assessed service charges or fees for types of transactions or problems with their account. For instance, most people are painfully familiar with an overdraft or insufficient funds fee, which can happen when a transaction comes in for more than our available balance, and the bank assesses a fee. Obviously, if you are processing ACH transactions, you could certainly encounter this situation. Other fees may be associated with different actions, perhaps originating ACH transactions, processing IAT transactions, creating a stop payment, or a monthly maintenance fee, to name just a few.

Given that you have companies, customers, and accounts (depending on how many additional features you have implemented), there are ample opportunities to apply different types of service charges to customers. Sticking with the ACH theme, you may choose to also charge companies for the number of transactions they have processed or originated, or you may charge them for returned or reversed items. While service charges start to tackle the broader FinTech area, there are plenty of opportunities to apply charges to ACH specific areas.

12.5 Final thoughts

We have made it to the end of our journey and would like to thank you for sticking with us! The main goal for this book was to pique your interest in software development and finance, hoping that you continue exploring other areas or dive deeper into a topic that we have already touched upon. In our experience, what separates good software developers from great software developers is curiosity and the willingness to expand one's knowledge to continue to grow. Taking on this book and making it to the end should show you that you have what it takes to be a great programmer.

In this book, you had an opportunity to work through a FinTech project following an agile methodology. We wanted to provide you with a project based on the real world and follow the same process that you are likely to see in any company you will work for.

From processing an ACH file, to creating APIs, and presenting everything in a UI, you worked on a lot of areas and covered a lot of additional topics. We hope you enjoyed the journey!

Summary

- Whether it is expanding the existing code, swapping out the FastAPI framework, or moving away from Python entirely, a plethora of things can be done to continue expanding the backend processing of our project. Rewriting functionality from one language to another often provides a great learning experience, and the ability to use language or framework features not available in the original language can be illuminating.
- Sometimes, pieces of code or entire applications need to be rewritten because of resource constraints. We have seen applications rewritten into a new language because they were legacy software written in a nonstandard language, and the team simply did not have the bandwidth or resources to devote to learning it just to maintain that piece of software.
- Well-established products often have plenty of infrastructure around them. Such infrastructure is often homegrown, especially if we are dealing with legacy software that may have existed before commercial tools were available to fill some of the niches.
- Knowing tools that are out there can be helpful, as homegrown tools eventually need to be replaced. Often, as team members mature, there is a need to replace the existing home-grown tools with a standard product so that new team members have a reduced learning curve.
- ACH is widely used in the United States, and as we saw, it has international capabilities as well. Processing ACH transactions provides some real-world exposure to what happens billions of times a year without much fanfare. Creating a simple system to begin processing ACH transactions and see just what goes on during the process gives us a great base to expand from.

index

Numbers

201 status code 103
404 error 208, 263, 285, 287
404 NOT FOUND error message 291
405 METHOD NOT ALLOWED status 103

A

accept status 243
accounts 401
ACH (automated clearing house) 5, 29
  batches APIs 207
  batch view 182–186
  converting recent order to recent ACH uploads 170–175
  dashboard 5–6
  supporting batches in dashboard 213
ACH (Automated Clearing House) files
  creating 221–224
  creating for companies 327
  creating Pydantic schema for parsed ACH records 160
  creating Pydantic schema for unparsed ACH records 159
  creating with exceptions 244–247
  designing database 128–247
  lessons learned 156–159
  modernizing legacy software 30
  parsing 32–41
  storing 127, 132–144, 152–156
  unit test changes 160
  uploading 147–152
ACH (automated clearing house) file uploads, mocking data 175–181
  accessing endpoints with axios 179
  adding WireMock to Docker 176
  setting up mappings in WireMock 177–179
ACH (Automated Clearing House) network 31
  origination and returns 401
ACH (Automated Clearing House) payment network
  file header record (type 1) 45
ACH (Automated Clearing House) transactions
  international 378–383
  OFAC scanning 378–383
  searching 283
AchBatchEntriesResponse 276, 376, 377
AchBatchesResponse 208–209
AchBatchesScatterPlotProps interface 187
AchBatchesScatterTooltip component 188
ACH class 34
ACH dashboard 10–13
  tools of trade 13

AchExceptionsSql class 253
AchFileProcessor 132, 153, 205
AchFileSql class 154, 208
AchFilesResponse class 200
AchFilesResponse interface 203
ACH operators or clearing houses 31
AchParsingValidationError 254
AchRecordBaseSchema 159
AchRecordParser 367
AchRecordProcessor 367
AchTransactionSearchResponse interface 292
AchTransactionSearchResults component 293
AchUpload interface 198
addenda records 51
add method 215
Address type 328
Alpine Linux 87
analytics 96
AnyUrl data type 299
APIs (application programming interfaces) 4, 94, 96, 383–387, 395
  adding to Docker 104–106
  API-driven design 95
  building with FastAPI 97–104
  calling 79–83
  creating container 74
  design, company information 319
  documenting 106–109
  endpoints 385
  ensuring accessible 220
  finishing 386
  implementing company limits API 346–349
  improving code 121–125
  load testing with Locust 116–120
  managing 109–114
  unit tests 384
apk (Alpine Package Keeper) 88
app folder 217
apt-get command 87, 88
args parameter 192
ARIA (Accessible Rich Internet Applications) 168
arithmetic errors 198
ASGI (Asynchronous Server Gateway Interface) 75, 105
asynchronous processing 394
auditing
  adding APIs for audit log 308
  creating page 303
  creating page components 304–308
  middleware for logging 301–303
  search API 286
  searching, ACH transactions 283
  viewing audit log 303–309
auditing user interactions 297–301
  auditing logic 300
  database design 297
  TDD testing for auditing 298–300
AuditLog class 299
AuditLogRecord class 299, 300
AuditLogRecords component 304
AuditModal component 307
AuditResponse interface 304
Authorization header 227
automated testing 42
AWS Secrets Manager 132
axios 165
  accessing endpoints with 179
  call 203

B

B2B (business-to-business) payments 3
backend processing 394–396
  APIs 395
  asynchronous processing 394
  ORM frameworks 395
Bahree, Amit 18
Base64-encoded string 329
BaseModel record 160
batch control record 52
batch header record (type 5) 48
batch rejection 56
BatchSearchResponse object 335
batch statistics component 271
BDD (behavior-driven development) 48, 49, 204, 284
BETWEEN keyword 290
black command 45
BYTEA type 314

C

CA (certificate authority) 229
carta-ach package 34

certificates 230
ChatGPT 15–18
  parsing ACH files 34–38
CI/CD (continuous integration and delivery) 101, 397
CIDR (Classless Inter-Domain Routing) 306
CloudBeaver, creating container 70
CMD 75, 92
cognitive load 41
CompaniesListing component 323
company dashboard 326
  creating ACH files for companies 327
  navigation for 326
CompanyDetailRecord class 328, 329
company information 311, 313
  API design 319
  Company Information component 328–331
  Company Limits component 344–353
  database design 313
  Expected Files component 337–343
  generating mock data 317
  Recent Batches component 333–336
  user interface 322
Company Limits component 344–353
  displaying company limits 349–353
  implementing company limits API 346–349
  storing company limits 345
COMPANY_LIMITS_EXCEEDED exception 347
CompanyLimitsSql class 347
CompanyOverviewRecord 320
CompanySearchSql class 340
components, defined 164
component testing 100
computeBatchStatistics function 273
computeBatchTotals function 272
CONCAT function 382
configuration files, cleaning up 217
constraints on fields 129
Copilot, parsing ACH files 38
COPY command 75, 122
CORS (cross-origin resource sharing) 81, 217, 233
COTS (commercial off-the-shelf tools) 297
createData function 179
createListItems function 273
CREATE TABLE statement 138, 247–248
credential provider 225
criteria parameter 286
CRUD (Create, Read, Update, and Delete) methods 154
CSR (client-side rendering), defined 164
CSS (Cascading Style Sheets), defined 164
CSV (comma-separated value) 24
CTE (common table expression) 268, 380
curl command 114
CurrencyTick component 188
cursor methods 144
customers 401
customer satisfaction 30
CVE (Common Vulnerabilities and Exposures) 60

D

DAITCH_MOKOTOFF function 382
dashboard, international ACH transactions
  batch entries 376
  effects on 375–377
  get_batches method 375
dashboard.conf 230
Dashboard.tsx file 168
databases 398
  company information 313
  designing 128–132, 297
database trigger, defined 157
database view, defined 157
DataGrid 277
data integrity 129
Data-Oriented Programming (Sharvit) 398
data ownership 24–26
DBA (database analyst) 132
DbUtils class 151
DDA (Demand Deposit Accounts) 277
debugging 18–21
Decimal data type 199
defect backlog 30
DELETE request 102
describe keyword 168
dive command-line tool 88
Docker 65, 396
  adding APIs to 104–106
  adding WireMock to 176
  calling API 79–83
  connecting containers 76–79
  container security 83–85

deploying dashboard in 192
docker-compose.yml file 68–76
health checks 92
optimizing 86–89
removing hardcoded username/password 89–91
where to begin 66
docker-compose commands 71, 74, 76, 87, 106, 111, 134, 218, 339
docker-compose.yml file 68–76
creating API container 74
creating CloudBeaver container 70
creating PostgreSQL container 72–74
creating web server container 76
Dockerfile 229
docker image ls command 86
Docker in Action (Nickoloff and Kuenzli) 67
docker ps command 84, 92
Docker Scout 88
Docker squash 89
documentation 95
DOM (Document Object Model), defined 164

E

EIN (Employer Identification Number) 314
ELT (extract–load–transform) 133
enabler story 30, 156
endpoints 385
entry detail record (type 6) 50
entry rejection 57
ENUM type 247, 315
environment variables 91, 92, 132
escaped defects 30
Essential TypeScript 5 (Freeman) 165
ETL (extract–transform–load) 133
EULA (End-User License Agreement) 26
exceptions 55–57, 145–147, 241–243, 239
adding application 275
batch rejection 56
batch statistics component 271
creating ACH files with 244–247
creating exceptions component 270
creating unit tests for 251
entry rejection 57
exception handling, additional UI changes 269
expanding ACH processing for 247–250
file rejection 55
making UI responsive 278
planning enhancements 240–241
supporting in parser 252–255
UI (user interface) 263–269
updating ACH parser 251–256
updating APIs 256–263
validation in Pydantic 255
EXPLAIN ANALYZE 332

F

FastAPI 78, 97– 104
adding file with POST /files 103–104
retrieving files with GET /files 99–102
file header record (type 1) 45, 138–143
full example 139–143
using generative AI 138
file rejection 55
/files APIs 197–207
testing 204–207
file trailer record (type 9) 53
financial software
development, user interface 398–400
international ACH transactions and OFAC scanning 378–383
testing with Playwright 231–233
FinTech
overview of 3
SAFe Agile 7
for-each loop 57, 115
FOREIGN KEY 152
constraints 150
foreign keys 129
format argument 92
Freeman, Adam 165
FROM command 75
Futuristic FinTech project 4
fuzzystrmatch 380, 382

G

generative AI 14–27
ChatGPT 15–18
data ownership 24–26
debugging 18–21
ownership 26
parsing ACH files 40–41

privacy 26
rubber duck debugging 14
security 26
storing file header record 138
syntax 21–24
Generative AI in Action (Bahree) 18
get(field, default value) function 245
GET /files 99–102
GET request 114
gitignore command 90
Given/When/Then 49, 101
GraphQL 95
Grid container 278

H

handleDrop function 216
handleSearch function 294
HashiCorp Vault 132
health checks 73, 76, 77, 82, 92
hooks, defined 164
HTML (Hypertext Markup Language), defined 164
HTTP POST request 148

I

IAT (international ACH transactions) 355, 357
batches, overview of 357
creating IAT batches 359–363
file processing 368–375
record parsing 364–367
IBM Websphere MQ 95
id 330
IDE (Integrated Development Environment) 40
if/elif/else processing 372
ILIKE keyword 290, 380
ImageBase64 323, 324
IndustryType enumeration 320
INET data type 297, 299, 306
infrastructure
continuous integration and delivery 397
database 398
Docker 396
multitenant environment 396
testing 397
INNER JOIN 329, 351
INNER JOINs 329, 335, 351
INSERT statement 153
integration testing 100
with Playwright 295
international ACH (automated clearing house) transactions 355
creating IAT batches 359–363
effects on dashboard 375–377
OFAC scanning 378–383
record parsing 364–367
sprint planning 356
user interface 387–391
INTERVAL function 340
InvalidTextRepresentation error 261
IPvAnyNetwork data type 299, 306
ITIN (Individual Tax Identification Number) 314
it keyword 168

J

Jest tests 174

K

key/value pairs 115
KISS (Keep OT Simple, Stupid) 351
KPIs (key performance indicators) 30
Kuenzli, Stephen 67
KYC (Know Your Customer) 378

L

L10n (localization) 172
latest tag 86
Lean Startup, The (Ries) 195
LEFT JOIN 329, 335, 351
legacy software, modernizing 30
LEVENSHTEIN function 382
LIKE keyword 290
LLMs (large language models) 14, 144
load testing 116–120
localhost 230
Locust 116–120
logging, middleware for 301–303
login page 224–228

M

main.py 124
manual and exploratory testing 100
map function 172

Measure-Object command 42
middleware 224, 301–303
Missing Credentials error 113
MLP (minimum lovable product) 195
MMF (minimum marketable feature) 195
MMP (minimum marketable product) 195
mock data, generating 317
ModuleNotFoundError 122
monitoring 96
MUI (Material UI) 264
multipart-formdata 148
multitenant environment 396
musl libc library 87
MVP (minimum viable product) 73, 194, 393
  ACH batches APIs 207
  adding basic login page 224–228
  adding /files/id/batches API 207
  adding TLS to dashboard 229–231
  building out /files APIs 197–207
  creating ACH files 221–224
  different types of 195
  preparing for 196
  putting puzzle together 217–221
  supporting ACH batches in dashboard 213
  testing with Playwright 231–233
  troubleshooting 233–235
  uploading files 215

N

Nacha (National Automated Clearing House Association) 31, 38, 42, 50
NameError 79
NaN (not a number) 197, 203
NDA (nondisclosure agreement) 30
NextAuth.js 224
Next.js
  converting recent order to recent ACH uploads 170–175
  deploying dashboard in Docker 192
  expanding UI with ACH batch view 182–186
  mocking ACH uploads data 175–181
  overview 164–165
  refactoring navigation 168–170
  ScatterPlot component 186–192
  upload pages 166–168
NEXT_PUBLIC_API_URL 192, 219
nginx 229, 230
Nickoloff, Jeff 67
NotImplementedError 102, 103
NOT NULL constraint 248
NPI (nonpublic information) 25, 26, 262
NPM (Node Package Manager) 164
npm run test command 168
npx command 164
NSF (Insufficient funds) 246, 401
null values 198
NUMERIC type 199

O

ODFI (Originating Depository Financial Institution) 56
OFAC (Office of Foreign Asset Control) scanning 355, 378–383
  API 383–387
  sanctioned individuals and countries 378
  scanning for individuals 380–383
  user interface 388–391
OfacRecords component 388, 389
OfacResponse component 388
OfacSql class 386
onChange event 294
onClick event 294
ON DELETE CASCADE 345
onRowClick 323
ON UPDATE CASCADE 345
OpenAPI 106
  version 111
OpenSSL 229, 230
OpenTelemetry 96, 126, 303
optimizing Docker 86–89
  general maintenance 86
  optimizing build time 88
  optimizing image size 87
origination, defined 401
ORM (object-relational mapper) 132
  frameworks 395
ORM (object-relational model) 154
orphan records 129
os.getenv command 91
os module 91
OWASP (Open Worldwide Application Security Project) 26
ownership, explained 26

P

P2P (person-to-person) payments 3
pages
  creating 303
  creating components 304–308
page.tsx file 166
parsing ACH files 32–41
  generative AI 40–41
  with ChatGPT 34–38
  with Copilot 38
parsing errors 198
parsing payments 29
  adding addenda records 51
  batch control record 52
  batch header record (type 5) 48
  entry detail record (type 6) 50
  file header record (type 1) 45
  file trailer record (type 9) 53
  handling exceptions 55–57
  interpreting code 57–60
  parsing records 42
  testing number of records read 42
payments
  ACH network 31
  automated testing 42
  modernizing legacy software 30
  parsing 29, 32–41, 57–60
pend status 243
PEP (Python Enhancement Proposal) 59
perldoc 24
Phone type 328
PII (personally identifiable information) 26
PI (program increment) 8
PlantUML 96
Playwright 231–233
  integration testing with 295
PoLP (principle of least privilege) 85
Positive Pay 402
POST call 368
POST /files 103–104
PostgreSQL 78
  creating container 72–74
Postman, testing with 114–116
POST request 102, 103, 119, 120, 197, 227
PPD (Prearranged Payment and Deposit) 31, 129
prefixes 121
PRIMARY KEY 152
primary keys, defined 129
privacy, defined 26
prompt engineering 17
prune command 86
psycopg 77, 79, 144, 211
PUT request 102
Pydantic 122
  creating schema for parsed ACH records 160
  creating schema for unparsed ACH records 159
  troubleshooting 125
  validation with 124–125, 255
Pydantic class 328
pylint 211
pyNacha package 34
pytest 136, 168, 219
pytest-bdd 101, 219
pytest.fixture 137
pytest fixtures 137, 155
pytest-postgresql 137
python-ach package 34
python-multipart package 148

Q

QA (quality assurance) 9, 48, 100, 114–115, 204, 227

R

RabbitMQ 95
RadioButtonUnchecked icon 343
RAG (retrieval-augmented generation) 144
RDBMS (relational database management system) 247
read_files function 99
Readonly feature 187
Recent Batches component 333–336
  displaying recent batches 336
  implementing recent batches API 334–335
recharts 165
records, storing with integrity 152
Redoc 108
refactoring, Next.js 168–170
referential integrity 128
reject status 243
release cadence 30
replace function 25

requirements.txt file 77, 105
response class 208
REST API 302
REST (Representational State Transfer) 95
returns, defined 401
RFC-5322 24
Ries, Eric 195
routing transit number 31
rubber duck debugging 14
RUN command 75

S

SAFe Agile 7
scaling 95
ScatterPlot component 186–192
  custom ticks 189
  custom tooltip 189
  overview 187
  using colors for chart 190–192
SCHEDULE_TYPE 338
SDLC (software development lifecycle) 7
SDN (Specially Designated Nationals) 355
searchCriteria field 294
searching
  ACH transactions 283
  BDD testing for 284
  search API 286
searching and auditing 281
  auditing 303–309
  auditing user interactions 297–301
  integration testing with Playwright 295
  sprint planning 282
SEC (standard entry class) code 50, 359
secure coding 59
security 26, 96
SELECT statement 321, 339
server configuration 230
service charges 402
Sharvit, Yehonathan 398
SLA (Service Level Agreement) 95
slim tag 87
SMEs (subject matter experts) 54, 132
SOAP (Simple Object Access Protocol) 95
some method 115
SonarLint 41
sourceMap 219
sprint planning 282, 312, 356
SQL (Structured Query Language) 4, 132–137
  adding records to ach_files table 133–137
  commands 77, 317
  query 340
SqlUtils 153
  SqlUtils class 144, 150
  SqlUtils.truncate_all() function 248
  SqlUtils.truncate_all method 160
  SqlUtils.truncate() method 146
SSL (Secure Sockets Layer) 230
SSN (Social Security Number) 314
SSR (server-side rendering) 164
state, defined 164
status_code 104
step definitions 49
storing ACH files 143–144
  creating Pydantic schema for parsed ACH records 160
  creating Pydantic schema for unparsed ACH records 159
  design and different approaches 161
  exceptions 145–147
  full example 139–143
  lessons learned 143
  records with integrity 152
  storing file header record 138–143
  unit test changes 160
  using generative AI 138
stripSubnet method 306
Swagger 107
switch statement 57
syntax 21–24

T

TaskAlt icon 343
TDD (test-driven development) 40, 241
  testing for auditing 298–300
TestClient 100
testing 397
  automated 42
  payment parsing, testing number of records read 42
  with Playwright 231–233
  with Postman 114–116
TextField 294

Then clause 206, 285
ticks, scatterplot custom 189
TLS (Transport Layer Security) 229–231
toFixed function 172
toggleDrawer function 170
tooltips, scatterplot custom 189
ToS (Terms of Service) 26
totalCreditAndDebits 187
TransactionSearchResponse class 287
TransactionSearchSql class 288
troubleshooting Pydantic 125
TSX (TypeScript) 166
type hints 59

U

UI (user interface) 4, 163, 322, 387–391, 398–400
  charts and forms 398
  configurable dashboards 400
  exceptions 263–269
  exception handling, additional UI changes 269
  expanding with ACH batch view 182–186
  making responsive 278
  mobile version 400
  OFAC components 388
  OFAC page 388
  OFAC results 390
  testing 100
Unable to resolve column error 153
UndefinedColumn 152
undefined values 198
unit tests 100, 384
  changes 160
  creating for exceptions 251
unparsed records 262–263
uploading files 215
upload pages, testing 166–168
UPPER function 290
URL field 299
urlPathTemplate 183
use cases 95
UUID (Universally Unique Identifier) 131
Uvicorn 105
UX (user experience) 4

V

validation 96, 124–125
  in Pydantic 255
valueGetter function 265
VARCHAR type 199
verification 24
versioning 95, 121
version tag 71
VOLUME keyword 176

W

web server container 76
When clause 259, 285
WireMock
  adding to Docker 176
  defining data for ACH batch view 182
  setting up mappings in 177–179
with statement 135
WSO2 API manager 110
  creating API in 112
  OpenAPI version 111
  troubleshooting 113

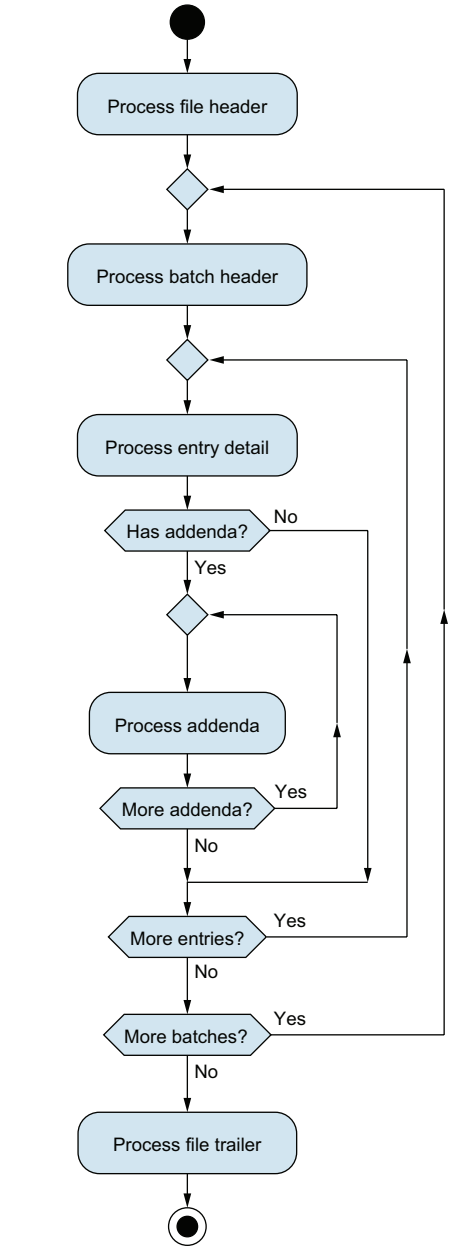
X

XP (extreme programming) 40

Y

YAGNI (You aren't going to need it) principle 73

Automated Clearing House (ACH) file processing



SOFTWARE DEVELOPMENT/PYTHON

BUILD

Financial Software with Generative AI

(FROM SCRATCH)

Kardell • Brouwer

The financial industry is awash with regulatory and compliance challenges, complex technical requirements, and stringent security demands. There's a huge demand for developers who can create financial services software and this book will get you started. You'll build your own FinTech app from the ground up—with a big productivity boost from Generative AI!

Build Financial Software with Generative AI (From Scratch) guides you through modernizing a full-stack Automated Clearing House (ACH) application, layer-by-layer. You'll start with a quick review of FinTech basics and an introduction to Gen AI tools. Then, you'll develop a data visualization dashboard with React, containerize components with Docker, create and refine APIs, implement backend processing, and even design a custom database. Throughout, you'll see how AI tools aid with coding, testing, research, security, documentation, and even Agile practices.

What's Inside

- Learn the core concepts of FinTech development
- Create and document APIs using Generative AI
- Build an awesome data visualization dashboard

Examples are in Python. No experience with generative AI or financial services required.

Christopher Kardell and **Mark Brouwer** have both spent more than 20 years working in the Fintech industry.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

"Filled with practical insights, real-world examples, and clear explanations. Indispensable."
—Murugan Lakshmanan
Fidelity Investments

"An insightful guide to the intersection of Generative AI and financial software development."
—Aditya Pant, Capco

"Thorough and fascinating! An effective dive into FinTech and generative AI—complete with a realistic project."
—Matthew O'Connor, Floor and Decor, Inc.

"A journey into FinTech infused with the power of AI."
—Saranya Balasubramaniyan
Amazon

ISBN-13: 978-1-63343-662-6