Community Experience Distilled

WordPress Web Application Development

Develop powerful web applications quickly using cutting-edge WordPress web development techniques

Rakhitha Nimesh Ratnayake

[PACKT] open source* PUBLISHING
community experience distilled

WordPress Web Application Development

Develop powerful web applications quickly using cutting-edge WordPress web development techniques

Rakhitha Nimesh Ratnayake

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

WordPress Web Application Development

First published: November 2013

Production Reference: 1111113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-075-9

www.packtpub.com

Cover Image by William Kewley (william.kewley@kbbs.ie)

Credits

Author
Rakhitha Nimesh Ratnayake

Reviewers
Rudolf Boogerman
Michael Cannon
Geert De Deckere
Benjamin Moody
Doug Sparling

Acquisition Editors
Saleem Ahmed
Sam Wood

Lead Technical Editor
Balaji Naidu

Technical Editors
Manan Badani
Pankaj Kadam
Krutika Parab

Copy Editors
Mradula Hegde
Gladson Monteiro
Kirti Pai
Lavina Pereira
Laxmi Subramanian

Project Coordinators
Abhijit Suvarna

Proofreaders
Maria Gould
Simran Bhogal
Paul Hindle

Indexer
Hemangini Bari

Graphics
Ronak Dhruv
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Rakhitha Nimesh Ratnayake is a freelance web developer, writer, and an open source enthusiast.

He also provides technical consultation on large scale web applications to one of the leading software development firms in Sri Lanka.

Rakhitha is the creator of `www.innovativephp.com`, where he writes tutorials on the latest web development and design technologies. He is also a regular contributor to a number of popular websites such as 1stwebdesigner, the Tuts+ network, and the SitePoint network. *Building Impressive Presentations with impress.js* was his first book, also published by Packt Publishing.

In his spare time, he likes to watch cricket and spend time with his family. Make sure you visit him online at `www.innovativephp.com` and follow him on Google+ at `http://goo.gl/UiEf5B`.

I would like to thank my parents and my brother for the encouragement and help they provided throughout the various tasks in this book. Also, I would like to thank my family members and friends at Providence for consistently motivating me to complete my second book for Packt Publishing.

I would also like to thank the Packt Publishing team members; Parita Khedekar for inviting me to write this book, Erol Staveley and Saleem Ahmed for the support throughout the book as Acquisition Editors, Abhijit Suvarna for being the Project Coordinator of the book, and the reviewers for providing honest feedback to improve the book.

Finally, I would like to thank you for reading my book and being one of the most important people who helped me make this book a success.

About the Reviewers

Rudolf Boogerman is of Dutch origin, born in England, and has lived in Belgium since 2007. He first studied graphic arts, but learned several programming/scripting languages hands-on, starting from 1990. From then on, he primarily developed multimedia presentations for a wide variety of projects worldwide.

In 1997 he created his first site, `www.raboo.info`, to show his own artwork. After this first experience on the Web, he got very excited and decided to combine his artistic background with web development. For the past 5 years, he has been primarily working with WordPress and Joomla! and developing plugins/extensions for both platforms.

Rudolf is the founder of Raboo Design (a visual communication agency), Footprint Visual Communication (visual communication agency), Footprint add-ons (extensions for Joomla!), WP 21 century (plugins for WordPress), and Miracle Tutorials (a blog with step-by-step advice on video and audio on the Web).

Michael Cannon, Peichi's smiling man, is an adventurous water rat, Chief People Officer, cyclist, full stack developer, poet, WWOOFer, and a world traveler. At his core, he is the happiest of all being productive, doing something different, living simply, and sharing with people.

Through Aihrus, he supports TYPO3 and WordPress clients; develops software-based products; and provides business, IT, and software development mentoring.

As Axelerant's Chief People Officer, he gets to collaborate with awesome teammates and smart folks as part of exciting open source projects, even as he continues to explore the world.

Germany, India, and Taiwan have been his homes since 2008. He has visited an average of five countries every year since 2005. In 2012, it was 14 countries. In 2013, nine countries have padded his feet or his bicycle wheels so far.

Geert De Deckere is a web developer living in Belgium. Around 10 years ago, he wrote his first lines of PHP. He loves to code and play around with ever-evolving web technologies such as WordPress. Apart from that, he also enjoys cycling (especially in the mountains), cooking, working in his vegetable garden, and playing an occasional game of chess. His personal website can be found at `http://geertdedeckere.be/`.

Geert was also a technical reviewer of *Kohana 3.0 Beginner's Guide*, which is published by Packt Publishing.

Benjamin Moody is a web developer who started using WordPress way back in 2005. Since then, he has developed several custom plugins and themes for clients across North America, as well as a number of WordPress-based web applications. When not coding for clients, Benjamin works on personal application projects and can be found providing support at Toronto WordCamp Happiness Bar.

Doug Sparling works as a web and mobile software developer for Andrews McMeel Universal, a publishing and syndication company in Kansas City, MO. As a long-time employee of the company, he has built everything from the GoComics Android app to its registration, e-commerce systems, web services, and various websites using Ruby on Rails. He's now busy building another site in Rails and porting a Perl-based e-mail system to Go. Some of the AMU properties include GoComics.com (`http://www.gocomics.com/`), PuzzleSociety.com (`http://puzzlesociety.com/`), Doonesbury.com (`http://doonesbury.slate.com/`), and Dilbert.com (`http://dilbert.com/`).

He is also the Director of Technology for a small web development firm called New Age Graphics (`www.newage-graphics.com`). After creating a custom CMS using C# and ASP.NET, all his work has moved to WordPress since the time WordPress 3.0 was released, eliminating the need to ever run Windows again.

Doug is a passionate advocate for WordPress and has written several WordPress plugins, can be found on the WordPress forums answering questions (and writing sample code) under the username "scriptrunner", and occasionally plays the role of a grammar nerd as a volunteer in the WordPress Codex Documentation team.

His other experience includes PHP, JavaScript, jQuery, Erlang, Python, Magento, and Perl. Doug was also the co-author for a Perl book (*Instant Perl Modules*) and is a reviewer for other Packt Publishing books, including *Mastering Android 3D Game Development* and *jQuery 2.0 Animation Beginner's Guide*, as well as *The Well Ground Rubyist*, *2nd Edition* for Manning Publications.

In his less than ample spare time, Doug enjoys spending time with his family. Other passions include photography, writing music, hitting drums and cymbals with sticks, playing briscola, meditation, and watching countless reruns of Firefly, Buffy the Vampire Slayer, and Doctor Who.

Many thanks to Packt Publishing for giving me the opportunity to review a book on my favorite technology, and a big thanks to the WordPress community for being the best I've ever come across.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface **1**

Chapter 1: WordPress As a Web Application Framework **7**

WordPress as a CMS **8**

WordPress as a web development framework **9**

MVC versus event-driven architecture 9

Simplifying development with built-in modules **10**

Identifying the components of WordPress **12**

The role of the WordPress theme 12

The structure of a WordPress page layout 13

Customizing the application layout 13

The role of the admin dashboard 14

Admin dashboard 14

Posts and pages 14

Users 15

Appearance 15

Settings 15

The role of plugins 15

The role of widgets 16

A development plan for a portfolio management application **18**

Application goals and a target audience 18

Planning the application 19

User roles of the application 19

Planning application features and functions 20

Understanding limitations and sticking with guidelines **21**

Building a question-answer interface **22**

Prerequisites 23

Creating questions 23

Changing the status of answers 25

Saving the status of answers 30
Generating the question list 33
Summary 35
Chapter 2: Implementing Membership Roles, Permissions, and Features 37
Introduction to user management 38
Preparing the plugin 38
Getting started on user roles 39
Creating user roles for a application 40
What is the best action for adding user roles? 40
Knowing the default roles 41
How to choose between default and custom roles 42
Removing existing user roles 42
Understanding user capabilities 43
Creating your first capability 43
Understanding default capabilities 44
Registering application users 45
Implementing the frontend registration 46
Shortcode implementation 46
Pros and cons of using shortcodes 46
Page template implementation 47
Pros and cons of page templates 47
Custom template implementation 47
Building a simple router for user modules 47
Creating the routing rules 48
Adding query variables 49
Flushing the rewriting rules 49
Controlling access to your functions 51
What are the advantages of using do_action? 52
Creating custom templates 54
Designing the registration form 54
Planning the registration process 56
Handling registration form submission 56
Activating system users 61
Creating a login on the frontend 63
Displaying a login form 65
Time to practice 68
Summary 69
Chapter 3: Planning and Customizing the Core Database 71
Understanding the WordPress database 72
Exploring the role of existing tables 72
User-related tables 73

Post-related tables 74
Term-related tables 75
Other tables 76
Adapting existing tables into web applications 77
User-related tables 78
Post-related tables 78
Scenario 1 – an online shopping cart 79
Scenario 2 – hotel reservation system 79
Scenario 3 – project management application 79
Term-related tables 79
Other tables 80
Extending a database with custom tables 81
Planning the portfolio application tables 82
Types of tables in web applications 82
Creating custom tables 83
Querying the database 85
Querying the existing tables 85
Inserting records 86
Updating records 86
Deleting records 86
Selecting records 86
Querying the custom tables 87
Working with posts 88
Extending WP_Query for applications 88
Limitations and considerations 90
Transaction support 90
Post revisions 90
How to know whether to enable or disable revisions 91
Autosaving 91
Using metatables 92
Summary 92
Chapter 4: The Building Blocks of Web Applications 93
Introduction to custom content types 94
The role of custom post types in web applications 94
Planning custom post types for the application 94
Projects 95
Services 96
Articles 97
Books 97
Implementing custom post types for a portfolio application 97
Implementing the custom post type settings 100

Creating a projects class 102
Assigning permissions to projects 104
Creating custom taxonomies for technologies and types 105
Assigning permissions to project type 108
Introduction to custom fields with meta boxes 110
What is a template engine? 113
Configuring Twig templates 114
Creating your first Twig template 116
Persisting custom field data 119
Customizing custom post type messages 123
Passing data to Twig templates 124
Introduction to custom post type relationships 126
Pods framework for custom content types 128
Should you choose Pods for web development? 132
Time to practice 133
Summary 133
Chapter 5: Developing Pluggable Modules 135
A brief introduction to WordPress plugins 136
Understanding the WordPress plugin architecture 136
WordPress plugins for web development 137
Create reusable libraries with plugins 138
How to use AJAX in WordPress 138
Creating an AJAX request using jQuery 138
Defining AJAX requests 139
Planning the AJAX plugin 140
Creating the plugin 141
Including plugin scripts for AJAX 143
Creating reusable AJAX requests 146
Extensible plugins 148
Planning the file uploader for portfolio application 148
Creating the extensible file uploader plugin 149
Converting file fields with jQuery 151
Integrating the media uploader to buttons 152
Extending the file uploader plugin 155
Customizing the allowed types of images 155
Saving and loading project screens 157
Pluggable plugins 159
Time to practice 162
Summary 162
Chapter 6: Customizing the Dashboard for Powerful Backends 163
Understanding the admin dashboard 164
Customizing the admin toolbar 164
Removing the admin toolbar 165

Managing the admin toolbar items 166
Customizing the main navigation menu 169
Creating new menu items 171
Adding features with custom pages 171
Building options pages 172
Automating option pages with SMOF 173
Customizing the options page to use as a generic settings page 175
Building the application options panel 176
Using the WordPress Options API 178
Using feature-packed admin list tables 180
Building extended lists 181
Using an admin list table for following developers 182
Step 1 – defining the custom class 182
Step 2 – defining instance variables 182
Step 3 – creating the initial configurations 182
Step 4 – implementing custom column handlers 183
Step 5 – implementing column default handlers 184
Step 6 – displaying the checkbox for records 184
Step 7 – listing the available custom columns 185
Step 8 – defining the sortable columns of the list 185
Step 9 – creating a list of bulk actions 186
Step 10 – retrieving list data 186
Step 11 – adding the custom list as a menu page 187
Step 12 – displaying the generated list 187
An awesome visual presentation for the admin dashboard 190
The responsive nature of the admin dashboard 195
Time for action 197
Summary 197
Chapter 7: Adjusting Themes for Amazing Frontends 199
Introduction to a WordPress application's frontend 200
Basic file structure of a WordPress theme 200
Understanding template execution hierarchy 201
Template execution process of web application frameworks 203
Web application layout creation techniques 205
Shortcodes and page templates 205
Custom templates with custom routing 206
Using pure PHP templates 206
The WordPress way of using templates 207
Direct template inclusion 207
Theme versus plugin templates 208
Template engines 209
Building a portfolio application's home page 210
What is a widget? 211

Widgetizing application layouts **212**
Creating widgets 213
Creating a custom template loader **217**
Designing the home page template **219**
Generating an application's frontend menu **221**
Creating a navigation menu 222
Displaying user-specific menus on the frontend 224
Creating pluggable and extendable templates **225**
Pluggable or extendable templates 226
Extendable templates in web applications 226
Pluggable templates in WordPress 227
Comparing WordPress templates with Twig templates 229
Extending the home page template with action hooks **229**
Customize widgets to enable extendable locations 230
Planning action hooks for layouts **232**
Time for action **234**
Summary **234**

Chapter 8: Enhancing the Power of Open Source Libraries and Plugins **237**

Why choose open source libraries? **238**
Open source libraries inside the WordPress core **238**
Open source JavaScript libraries in the WordPress core **239**
What is Backbone.js? 240
Understanding the importance of code structuring 241
Integrating Backbone.js and Underscore.js 242
Creating a developer profile page with Backbone.js 243
Structuring with Backbone.js and Underscore.js 247
Displaying the projects list on page load 249
Creating new projects from the frontend 253
Integrating events to the Backbone.js views 254
Validating and creating new models on the server 255
Creating new models on the server 256
Using PHPMailer for custom e-mail sending **259**
Usage of PHPMailer within the WordPress core 260
Creating a custom version of the pluggable wp_mail function 260
Loading PHPMailer inside plugins and creating custom functions 261
Implementing user authentication with Opauth **263**
Configuring login strategies 265
Building a LinkedIn app 266
Process of requesting the strategies 270
Initializing the Opauth library 270
Authenticating users in our application 273

Using third-party libraries and plugins **276**
Time for action **277**
Summary **277**
Chapter 9: Listening to Third-party Applications **279**
Introduction to APIs **280**
Advantages of having an API 280
WordPress XML-RPC API for web applications **281**
Building the API client **281**
Creating a custom API **285**
Integrating API user authentication **287**
Integrating API access tokens **289**
Providing the API documentation **293**
Time for action **294**
Summary **295**
Chapter 10: Integrating and Finalizing the Portfolio Management Application **297**
Integrating and structuring a portfolio application **298**
Step 1 – deactivating all the plugins used in this book 298
Step 2 – creating a new standalone plugin 299
Step 3 – moving all the plugins into wpwa-web-application 299
Step 4 – removing plugin definitions 300
Step 5 – creating common folders 300
Step 6 – loading components to the main plugin 302
Step 7 – creating the template loader 302
Step 8 – reusing the autoloader 303
Step 9 – defining main plugin functions 303
Step 10 – building the template router 304
Step 11 – building the activation controller 306
Step 12 – building the script controller 307
Step 13 – building the admin menu controller 309
Step 14 – creating class initializations 310
Step 15 – initializing application controllers 310
Restructuring the custom post manager **312**
Integrating a template loader into the user manager **312**
Working with a restructured application **313**
Building the developer model 314
Designing the developer list template 315
Enabling AJAX-based filtering 316
Updating a user profile with additional fields **320**
Updating values of profile fields 322

Scheduling subscriber notifications **325**
Notifying subscribers through an e-mail 327
Lesser-known WordPress features **330**
Caching 330
Transients 332
Testing 332
Security 333
Time for action **334**
Final thoughts **334**
Summary **335**
Appendix: Configurations, Tools, and Resources **337**
Configure and set up WordPress **337**
Step 1 – downloading WordPress 337
Step 2 – creating the application folder 337
Step 3 – configuring the application URL 338
Creating a virtual host 338
Using a localhost 338
Step 4 – installing WordPress 339
Step 5 – setting up permalinks 342
Step 6 – downloading the Responsive theme 343
Step 7 – activating the Responsive theme 343
Step 8 – activating the plugin 343
Step 9 – using the application 344
Open source libraries and plugins **344**
Online resources and tutorials **345**
Index **347**

Preface

Developing WordPress-powered websites is one of the standout trends in the modern web development world. The flexibility and power of the built-in features offered by WordPress has made developers turn their attention to the possibility of using it as a web development framework. This book will act as a comprehensive resource for building web applications with this amazing framework.

WordPress Web Application Development is a comprehensive guide focused on incorporating the existing features of WordPress into typical web development. This book is structured towards building a complete web application from scratch. With this book, you will build a portfolio management application with a modularized structure supported by the latest trending technologies.

This book provides a comprehensive, practical, and example-based approach for pushing the limits of WordPress to create web applications beyond your imagination.

It begins by exploring the role of existing WordPress components and discussing the reasons for choosing WordPress for web application development. As we proceed, more focus will be put into adapting WordPress features into web applications with the help of an informal use-case-based model for discussing the most prominent built-in features. While striving for web development with WordPress, you will also learn about the integration of popular client-side technologies such as Backbone.js, Underscore, jQuery, and server-side technologies and techniques such as template engines and OpenAuth integration.

This book differentiates from the norm by creating a website that is dedicated to providing tutorials, articles, and source codes to continue and enhance the web application development techniques discussed throughout this book. You can access the website for this book at `http://www.innovativephp.com/wordpress-web-applications`.

After reading this book, you will possess the ability to develop powerful web applications rapidly within limited time frames with the crucial advantage of benefitting low-budget and time-critical projects.

What this book covers

Chapter 1, *WordPress As a Web Application Framework*, walks you through the existing modules and techniques to identify their usage in web applications. Identification of WordPress features beyond the conventional CMS and planning portfolio management application are the highlights of this chapter.

Chapter 2, Implementing Membership Roles, Permissions, and Features, begins the implementation of a portfolio management application by exploring the features of the built-in user management module. Working with various user roles and permissions as well as an introduction to the MVC process through routing are the highlights of this chapter.

Chapter 3, *Planning and Customizing the Core Database*, serves as an extensive guide for understanding the core database structure and the role of database tables in web applications. Database querying techniques and coverage of the planning portfolio management application database are the highlights of this chapter.

Chapter 4, *The Building Blocks of Web Applications*, explores the possibilities of extending WordPress posts beyond their conventional usage to suit complex applications. Advanced use of custom post types and an introduction to template engines are the highlights of this chapter.

Chapter 5, *Developing Pluggable Modules*, introduces the techniques of creating highly reusable and extensible plugins to enhance the flexibility of web applications. Implementing various plugins for explaining these techniques is the highlight of this chapter.

Chapter 6, *Customizing the Dashboard for Powerful Backends*, walks you through the process of customizing the WordPress admin panel for adding new features as well as changing existing features and design. Building reusable grids and designing an admin panel are the highlights of this chapter.

Chapter 7, *Adjusting Themes for Amazing Frontends*, dives into the techniques of designing amazing layouts, thereby opening them for future extension. Widgetizing layouts and building reusable templates are the highlights of this chapter.

Chapter 8, *Enhancing the Power of Open Source Libraries and Plugins*, explores the use of the latest trending open source technologies and libraries. Integrating open authentication in to your web application and structuring the application at the client side are the highlights of this chapter.

Chapter 9, *Listening to Third-party Applications*, demonstrates how to use WordPress XML-RPC API to create a custom API for your web application. Building a simple yet complete API with all the main features is the highlight of this chapter.

Chapter 10, *Integrating and Finalizing The Portfolio Management Application*, guides you through the integration of modules developed throughout this book into a standalone plugin while introducing you to important concepts such as caching, security, and testing.

Appendix, *Configurations, Tools, and Resources*, provides an application setup guide with necessary links to download the plugins and libraries used throughout the book.

What you need for this book

Technically, you need a computer, browser, and an Internet connection with the following working environment:

- The Apache web server
- PHP Version 5.2 or higher
- WordPress Version 3.6
- MySQL Version 5.0 or higher

Once you have the preceding environment, you can download the Responsive theme from `http://wordpress.org/themes/responsive` and activate it from the Themes section. Finally, you can activate the plugin developed for this book to get things started.

Please refer to *Appendix*, *Configurations, Tools, and Resources*, for the application setup guide, required software, and plugins.

Who this book is for

This book is intended for WordPress developers and designers who have the desire to go beyond conventional website development to develop quality web applications within a limited time frame and for maximum profit.

Experienced web developers who are looking for a framework for rapid application development will also find this to be a useful resource.

Prior knowledge of WordPress is preferable as the main focus will be on explaining methods for adapting WordPress techniques for web application development rather than explaining basic skills with WordPress.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "WordPress passes existing MIME types as the parameter to this function. Here, we have modified the `$mimes` array to restrict the image types to JPG."

A block of code is set as follows:

```
function filter_mime_types($mimes) {
    $mimes = array(
      'jpg|jpeg|jpe' => 'image/jpeg',
    );

    return $mimes;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public function flush_application_rewrite_rules() {
  $this->manage_user_routes();
  flush_rewrite_rules();
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Once the user fills the data and clicks on the **Register** button, we have to execute quite a few tasks in order to register a user in the WordPress database."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
WordPress As a Web Application Framework

In recent years, WordPress has matured from being the most popular blogging tool to the most popular content management system. Thousands of developers around the world are making a living from WordPress design and development. As more and more people become interested in using WordPress, there are discussions and arguments for exploring the possibilities of using this amazing framework for web application development.

The future seems bright as WordPress has already got dozens of built-in modules, which can be easily adapted to web application development using some slight modifications. Since you are already reading this book, you are probably someone who is really excited to see how WordPress fits into web application development. Throughout this book, we are going to learn how we can inject the best practices of web development into the WordPress framework to build web applications in a rapid process.

Basically, this book will be important for developers from two different perspectives. On one hand, WordPress developers of beginner level to intermediate level can get the knowledge of cutting edge web development technologies and techniques to build complex applications. On the other hand, web development experts who are already familiar with popular PHP frameworks can learn WordPress for rapid application development. So let's get started!

In this chapter, we are going to cover the following topics:

- WordPress as a CMS
- WordPress as a web development framework
- Simplifying development with built-in modules
- Identifying the components of WordPress

- A development plan for a portfolio management application
- Understanding limitations and sticking to guidelines
- Building a question-answer interface

In order to work through this book, you should be familiar with WordPress themes, plugins, and its overall process. Developers who are experienced in PHP frameworks can work with this book while using the reference sources to learn WordPress. By the end of this chapter, you will have the ability to make the decision on choosing WordPress for web development.

WordPress as a CMS

Way back in 2003, WordPress released its first version as a simple blogging platform and continued to improve, until it became the most popular blogging tool. Afterwards, it continued to improve as a **CMS** (**Content Management System**) and has a reputation now for being the most popular CMS. These days everyone sees WordPress as a CMS rather than just a blogging tool.

Now, the question is where will it go next?

No one really knows the answer for this question. But recent versions of WordPress have included popular web development libraries such as `Backbone.js` and `Underscore.js`, and developers are even building different types of applications with WordPress. So we can assume that it's moving towards the direction of building applications. It's important to keep an eye on the next few versions to see what WordPress offers for web applications.

Before we consider the application development aspects of WordPress, it's a good idea to figure out the reasons for it being such a popular framework. The following are some of the reasons behind the success of WordPress as a CMS:

- Plugin-based architecture for adding independent modules and the existence of over 20,000 open source plugins
- Super simple and easy-to-access administration interface
- Fast learning curve and comprehensive documentation for beginners
- Rapid development process involving themes and plugins
- Active development community with awesome support
- Flexibility in building websites with its themes, plugins, widgets, and hooks

These reasons prove why WordPress is the top CMS for website development. But experienced developers who work with full stack web applications don't believe that WordPress has a future in web application development. While it's up for debate, let's see what WordPress has to offer for web development.

Once you complete reading this book, you will be able to decide whether WordPress has a future in web applications. I have been working with full stack frameworks for several years and I certainly believe in the future of WordPress for web development.

WordPress as a web development framework

In practice, the decision to choose a development framework depends on the complexity of your application. Developers will tend to go for frameworks in most scenarios. It's important to figure out why we go with frameworks for web development. Here is a list of possible reasons that frameworks become a priority in web application development:

- They provide stable foundations for building custom functionalities
- Stable frameworks usually have a large development community with active support
- Frameworks contain built-in modules to address the common aspects of application development such as routing, language support, form validation, user management, and more
- They have a large number of utility functions to address repetitive tasks

Full stack development frameworks such as Zend, CodeIgniter, and CakePHP adhere to the points mentioned in the preceding section, which in turn become the framework of choice for most developers. Now let's take a look at how WordPress fits into the boots of the web application framework.

MVC versus event-driven architecture

A vast majority of web development frameworks are built to work with the **MVC** (**Model-View-Controller**) architecture, where the application is separated into independent layers called models, views, and controllers. In MVC, we have a clear understanding of what goes where and when each of the layers will be integrated in the process.

So, the first thing most developers will look at is the availability of MVC in WordPress. Unfortunately WordPress is not built on top of the MVC architecture. This is one of the main reasons that developers refuse to choose it as a development framework. Even though it is not MVC, we can create a custom execution process to make it work like an MVC application. Unlike other frameworks, it won't have the full capabilities of MVC. But unavailability of an MVC architecture doesn't mean that we cannot develop quality applications with WordPress.

WordPress relies on procedural event-driven architecture with its action hooks and filter system. Once the user makes a request, these actions will get executed in a certain order to provide the response to the user. You can find the complete execution procedure at `http://codex.wordpress.org/Plugin_API/Action_Reference`.

In an event-driven architecture, both the model and controller code gets scattered throughout the theme files. In the upcoming chapters, we are going to look at how we can separate these concerns with the event-driven architecture, in order to develop maintainable applications.

Simplifying development with built-in modules

As we discussed in the previous section, the quality of a framework depends on its core modules. The more quality you have in the core, the better it will be for developing quality and maintainable applications. It's surprising to see the availability of a number of WordPress modules directly related to web development, even though it was meant to create websites.

Let's get a brief introduction about the WordPress core modules to see how they fit into web application development:

- **User module**: The built-in user management module is quite advanced, catering to the most common requirements of any web application. Its user roles and capability handling makes it much easier to control the access to specific areas of your application. Most full stack frameworks don't have a built-in user module and hence this can be considered as an advantage of using WordPress.
- **File management**: File uploading and managing is a common and time-consuming task in web applications. Media Uploader, which comes built-in with WordPress, can be effectively used to automate the file-related tasks without writing much source code. This super-simple interface makes it so easy for application users to handle file-related tasks.

- **Template management**: WordPress offers a simple template management system for its themes. It is not as complex or fully featured as a typical templating engine, but it does offer a wide range of capabilities from the CMS development perspective, which we can extend to suit web applications.
- **Database management**: In most scenarios, we will be using the existing database table structure for our application development. WordPress database management functionalities offer a quick and easy way of working with existing tables with their own style of functions. Unlike other frameworks, WordPress provides a built-in database structure and hence most of the functionalities can be used to directly work with these tables without writing custom SQL queries.
- **Routing**: Comprehensive support for routing is provided through plugins. WordPress makes it simple to change the existing routing and choose your own routing, in order to build search engine friendly URLs.
- **XMR-RPC API**: Building an API is essential for allowing third-party access to our application. WordPress provides a built-in API for accessing CMS-related functionality through its XML-RPC interface. Also developers are allowed to create custom API functions through plugins, making it highly flexible for complex applications.
- **Caching**: Caching in WordPress can be categorized into two sections; persistent and nonpersistent cache. Nonpersistent caching is provided by the WordPress cache object, while persistent caching is provided through its transient API. Caching techniques in WordPress is a simple comrade to other frameworks, but it's powerful enough to cater for complex web applications.
- **Scheduling**: As developers, you might have worked with cron jobs for executing certain tasks at specified intervals. WordPress offers the same scheduling functionality through built-in functions, similar to a cron job. Typically, it's used for scheduling future posts. But it can be extended to cater to complex scheduling functionality.
- **Plugins and widgets**: The power of WordPress comes with its plugin mechanism, which allows us to dynamically add or remove functionality without interrupting other parts of the application. Widgets can be considered as a part of the plugin architecture and will be discussed in detail in the remainder of this chapter.

An overall collection of modules and features provided by WordPress can be effectively used to match the core functionalities provided by full stack PHP frameworks.

Identifying the components of WordPress

WordPress comes up with a set of prebuilt components, which are intended to provide different features and functionality for an application. A flexible theme and powerful admin module act as the core of WordPress websites while plugins and widgets extend the core with application-specific features. As a CMS, we all have a pretty good understanding of how these components fit into a WordPress website.

Here our goal is to develop web applications with WordPress, and hence it is important to identify the functionality of these components in the perspective of web applications, so we are going to look at each of the following components, how they fit into web applications, and how we can take advantage of them to create flexible applications through a rapid development process:

- The role of the WordPress theme
- The role of the admin dashboard
- The role of plugins
- The role of widgets

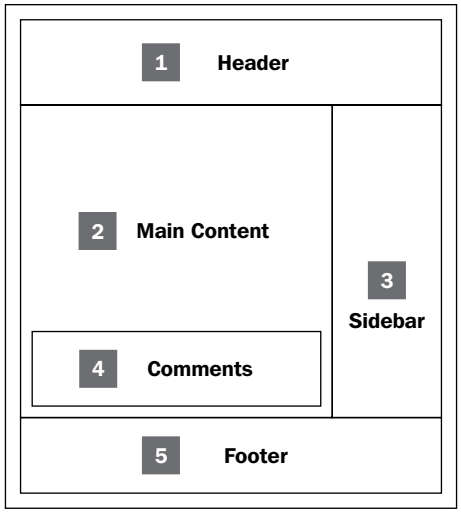
The role of the WordPress theme

Many of us are used to seeing WordPress as a CMS. In its default view, a theme is a collection of files used to skin your web application layouts. In web applications, it's recommended to separate different components into layers such as models, views, and controllers. WordPress doesn't adhere to the MVC architecture, but we can easily visualize the theme or templates as the presentation layer of WordPress.

In simple terms, views should contain the HTML needed to generate the layout and all the data it needs should be passed to the views. WordPress is built for creating content management systems and hence it doesn't focus on separating views from its business logic. Themes contain views, also known as template files, as a mix of both HTML code and PHP logic. As web application developers, we need to alter the behavior of existing themes, in order to limit the logic inside templates and use plugins to parse the necessary model data to the views.

The structure of a WordPress page layout

Typically, posts or pages created in WordPress consist of five common sections. Most of these components will be common across all the pages in the website. In web applications, we also separate the common layout content into separate views to be included inside other views. It's important for us to focus on how we can adapt the layout into a web-application-specific structure. Let's visualize the common layout of WordPress using the following diagram:



Having looked at the structure, it's obvious that the **Header**, **Footer**, and **Main Content** areas are mandatory even for web applications, but the footer and comments section will play a less important role in web applications, compared to web pages. The **Sidebar** is important in web applications, even though it won't be used with the same meaning. It can be quite useful as a dynamic widget area.

Customizing the application layout

Web applications can be categorized as projects and products. A project is something we develop that targets the specific requirements of a client. On the other hand, a product is an application created based on the common set of requirements for a wide range of users. Therefore, customizations will be required on layouts of your product based on different clients.

WordPress themes make it super simple to customize the layout and features using child themes. We can make the necessary modifications in the child theme while keeping the core layout in the parent theme. This will prevent any code duplications in customizing layouts. Also the ability to switch themes is a powerful feature that eases the layout customization process.

In situations where we need to provide dynamic layouts based on user types, devices, or browsers, we can dynamically switch between different themes by writing custom functions.

The role of the admin dashboard

The administration interface of an application plays one of the most important roles behind the scenes. WordPress offers one of the most powerful and easy-to-access admin areas compared to other competitive frameworks. Most of you should be familiar with using the admin area for CMS functionalities, but we will have to understand how each component in the admin area suits the development of web applications.

Admin dashboard

The dashboard is the location where all the users get redirected, once they are logged in to the admin area. Usually it contains dynamic widget areas with the most important data of your application. The dashboard could play a major role in web applications, compared to blogging or CMS functionality. We can remove the existing widgets related to CMS and add application-specific widgets to create a powerful dashboard.

Posts and pages

Posts in WordPress are built for creating content such as articles and tutorials. In web applications, posts will be the most important section to create different types of data. Often, we will choose custom post types instead of normal posts for building advanced data creation sections. On the other hand, pages are typically used to provide static content of the site. Usually we have static pages such as **About Us**, **Contact Us**, and **Services**.

Users

User management is a must-use section for any kind of web application. User roles, capabilities, and profiles will be managed in this section by authorized users.

Appearance

Themes and application configurations will be managed in this section. Widgets and theme options will be the important sections related to web applications.

Settings

This section involves general application settings. Most of the prebuilt items in this section are suited to blogs and websites. We can customize this section to add new configuration areas related to our plugins that are used in web application development.

There are some other sections such as links, pages, and comments, which will not be used frequently in complex web application development. The ability for adding new sections is one of the key reasons for its flexibility.

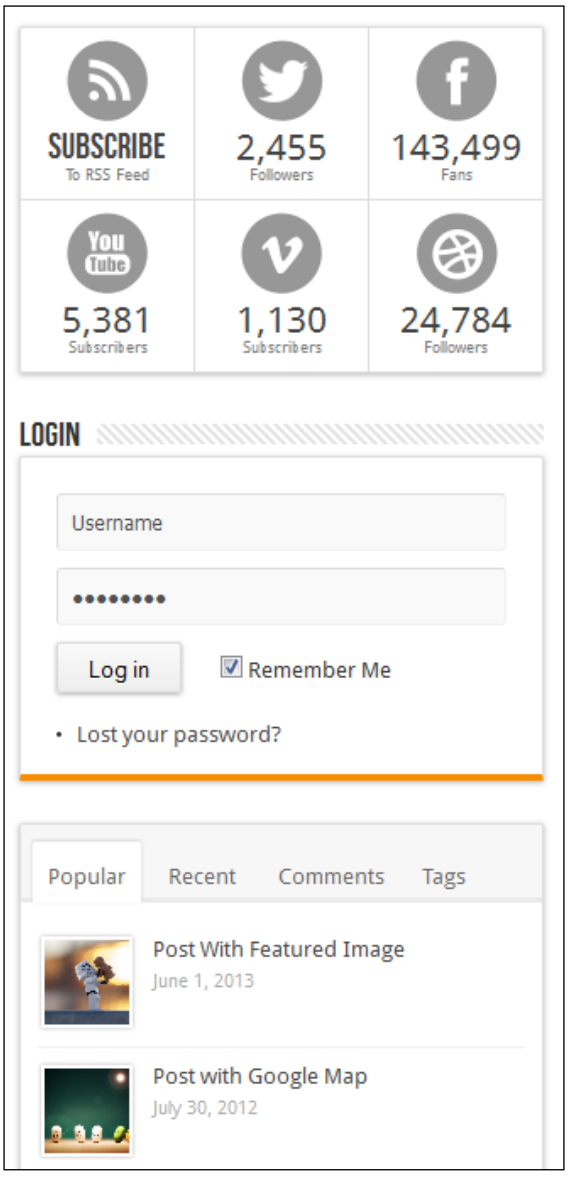
The role of plugins

In normal circumstances, WordPress developers use functions, which involve application logic scattered across theme files and plugins. Some developers even change the core files of WordPress, which is considered to be a very bad practice. In web applications, we need to be much more organized. In the section *The role of the WordPress theme*, we discussed the purpose of having a theme for web applications. Plugins will be and should be used to provide the main logic and content of your application. In short, anything you develop for web applications should go inside plugins, instead of theme files.

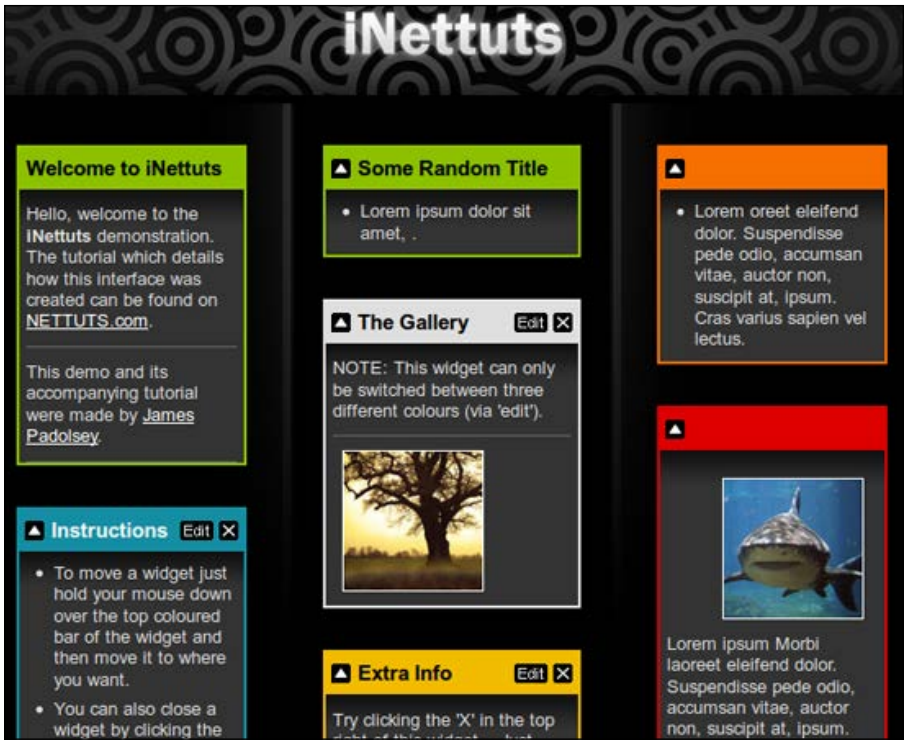
The plugin architecture is a powerful way to add or remove features without affecting the core. Also, we have the ability to separate independent modules into their own plugins, making it easier to maintain. On top of that, plugins have the ability to extend other plugins.

The role of widgets

The official documentation of WordPress refers to widgets as a component that adds content and features to your Sidebar. In typical blogging or from a CMS user's perspective, it's a completely valid statement. In fact, widgets offer more in web applications by going beyond content that populates Sidebars. The following screenshot shows a typical widgetized Sidebar of a website:



We can use dynamic widgetized areas to include complex components as widgets, making it easy to add or remove features without changing the source code. The following screenshot shows a sample dynamic widgetized area. We can use the same technique to develop applications with WordPress:



Throughout these sections, we covered the main components of WordPress and how they fit into the actual web application development. Now we have a good understanding of the components in order to plan the application that we will be developing throughout this book.

A development plan for a portfolio management application

Typically a WordPress book consists of several chapters, each of them containing different practical examples to suit each section. In this book, our main goal is to learn how we can build full stack web applications using built-in WordPress features, therefore, I thought of building a complete application, explaining each and every aspect of web development.

Throughout this book, we are going to develop an online portfolio management system for web development related professionals. This application can be considered as a mini version of a basic social network. We will be starting the development of this application from *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*.

Planning is a crucial task in web development in which we will save a lot of time and avoid potential risks in the long run. First, we need to get a basic idea of the goal of this application, and its features, functionalities, and the structure of components to see how it fits into WordPress.

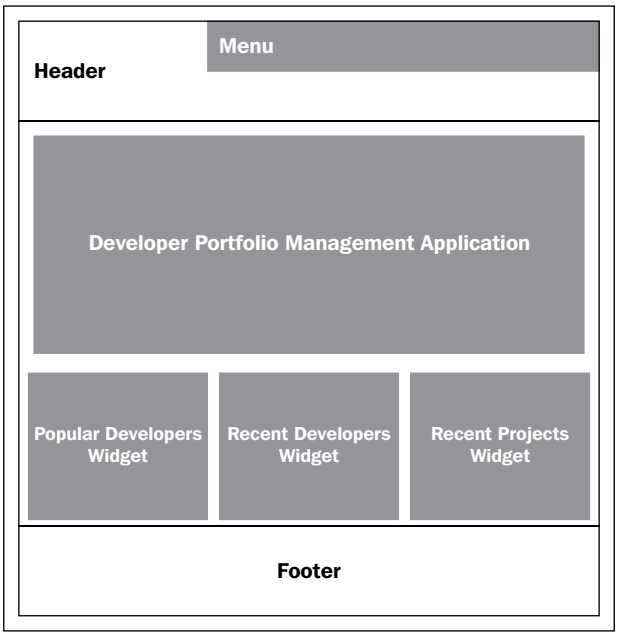
Application goals and a target audience

Developers and designers who work online as freelancers know the importance of a personal profile to show your skills for an improved reputation. But most people, including experts who work full time jobs, don't maintain such profiles and hence remain unnoticed among co-developers. The application developed throughout this book is intended to provide the opportunity for web professionals to create their public profiles and connect with the experts in the field.

This application is aimed at those who are involved in web development and design. I believe that both the output of this application and the contents of the book will be ideal for PHP developers who want to jump into WordPress application development.

Planning the application

Basically, our application consists of both frontend and backend, which is common to most web applications. In the frontend, both registered and unregistered users will have different functionalities based on their user roles. The following screenshot shows the structure of our application's home page:



The backend will be developed by customizing the built-in admin module of WordPress. The existing and new functionalities of the admin area will be customized based on the user role permissions.

User roles of the application

The application consists of four user roles, including the built-in admin role. The following are the user roles and their respective functionalities:

- **Admin**: This manages the application's configurations, settings, and the capabilities of the users.
- **Developer**: This is the user role common to all web professionals who want to make profiles. All the developers will be able to create complete profile details to enhance their reputation.

- **Members**: These are normal users who want to use plugins, themes, books, articles, and applications created by developers and designers. They will be able to access and download the work made public by developers. Basically, the members will have more permission to directly interact with developers, compared to subscribers. We could also implement a premium content section in the future for paid members.
- **Subscribers**: This is also a normal user who wants to follow the activities of their preferred developers. These users will be notified whenever their preferred developers create a new activity within the application.

Registration is required for all the four user roles in the Portfolio Management Application.

Planning application features and functions

Our main intention of building this application is to learn how WordPress can be adapted to advanced web application development. Therefore, we will be considering various small requirements, rather than covering all aspects of a similar system. Each of the functionalities will be focused on explaining various modules in web applications and the approach of WordPress in building similar functionality. Let's consider the following list of functions, which we will be developing throughout this book:

- **Developer profile management**: Users who register as developers will be given the opportunity to construct their profiles by completing content divided into various sections such as services, portfolios, articles, and books.
- **Frontend login and registration**: Typically, web applications contain the login and registration in the frontend, whereas WordPress provides it in the admin area. Therefore, custom implementation of login and registration will be implemented in the application frontend.
- **Settings panel**: A comprehensive settings panel will be developed for administrators to configure general application settings from the backend.
- **XML API**: A large number of popular web applications come up with a fully functional API to allow access to third-party applications. In this application, we will be developing a simple API to access the developer details and activities from external sources.
- **Notification service**: A simple notification service will be developed to manage subscriptions as well as manage updates about the application activities.

- **Responsive design**: With the increase of Internet browsing using mobile devices, more and more applications are converting their apps to suit various devices. So we will be targeting different devices for fully responsive design from the beginning of the development process
- **Third-party libraries**: Throughout this book, we will be creating functionalities such as `open auth login`, RSS feed generation, and template management to understand the use of third-party libraries in WordPress.

While these are our main functionalities, we will also develop small features and components on top of them to explain the major aspects of web development.

If you are still not convinced, you can have a look at various types of WordPress-powered web applications at `http://www.innovativephp.com/demo/packt/wordpress_applications`.

Understanding limitations and sticking with guidelines

As with every framework, WordPress has its limitations in developing web applications. Developers need to understand the limitations before deciding to choose a framework for application development.

In this section, we are going to learn the limitations while building simple guidelines for choosing WordPress for web development. Let's get started:

- **Lack of support for MVC**: We talked about the architecture of WordPress and its support for MVC in one of the earlier sections. As a developer, you need to figure out ways to work with WordPress in order to fit with your web applications. If you are someone who cannot work without MVC, WordPress may not be the best solution for your application.
- **Database migration**: If you are well experienced in web development, you will have a pretty good idea about the importance of choosing databases considering the possibilities of migrating to another one in later stages. This could be a limitation in WordPress as it's built-in to work with the MySql database. Using it with another database will be quite difficult, if not impossible. So if you need the database to be migrated to some other database, WordPress will not be the best solution.

- **Performance**: Performance of your application is something we get to experience in later stages of the project when we go into a live environment. It's important to plan ahead on the performance considerations as it can come through internal and external reasons. WordPress has a built-in database structure and we are going to use it in most of the projects. It's designed to suit CMS functionality and sticking with the same tables for different types of projects will not provide an optimized table structure. Therefore, performance might be a limitation for critical applications interacting with millions of records each day, unless you optimize your caching, indexing, and other database optimization strategies. WordPress runs on an event-driven architecture, packed with features. Often developers misuse the hooks without proper planning, affecting the performance of the application, so you have to be responsible in planning the database and necessary hooks in order to avoid performance overheads.
- **Regular updates**: WordPress has a very active community involving its development for new features and fixing the issues in the existing features. Once a new version of core is released, plugin developers will also update their plugins to be compatible with the latest version. Hence, you need to perform additional tasks for updating the core, themes, and plugins, which can be a limitation when you don't have a proper maintenance team.
- **Object-oriented development**: Experienced web developers will always look for object-oriented frameworks for development. WordPress started its coding with procedural architecture and is now moving rapidly towards object-oriented architecture, so there will be a mix of both procedural and object-oriented codes. WordPress also uses hook-based architecture for providing functionality for both procedural and object-oriented codes. Developers who are familiar with other PHP frameworks might find it difficult to come to terms with the mix of procedural and object-oriented code, as well as hook-based architecture, so you have to decide whether you are comfortable with its existing coding styles.

If you are a developer or designer, who thinks these limitations could cause major concerns for your projects, WordPress may not be the right solution for you.

Building a question-answer interface

Throughout the previous sections, we learned the basics of web application frameworks while looking at how WordPress fits into web development. By now, you should be able to visualize the potential of WordPress for application development and how it can change your career as developers. Being human, we always prefer a practical approach to learning new things over the more conventional theoretical approach.

So, I am going to complete this chapter by converting the default WordPress functionality into a simple question-answer interface such as Stack Overflow, to show you a glimpse into what we are going to develop throughout this book.

Prerequisites

We will be using Version 3.6 as the latest stable version, available at the time of writing this book. I suggest you to set up a fresh WordPress installation for this book, if you haven't already done so.

Also we will be using the TwentyTwelve theme, which is available with the default WordPress installation. Make sure to activate the TwentyTwelve theme in your WordPress installation.

First, we have to create an outline containing the list of tasks to be implemented for this scenario:

- Create questions using the admin section of WordPress
- Allow users to answer questions using comments
- Allow question creators to mark each answer as correct or incorrect
- Highlight the correct answers for each question
- Customize the question list to include the number of answers and number of correct answers

Now it's time to get things started.

Creating questions

The goal of this application is to let people submit questions and get answers from various experts in the same field. First, we need to create a method to add questions and answers. By default, WordPress allows us to create posts and submit comments to the posts. In this scenario, the post can be considered as the question and the comments can be considered as the answers. Therefore, we have the capability of directly using normal post creation for building this interface.

However, I would like to choose a slightly different approach by using custom post types in order to keep the default functionality of posts and let the new functionality be implemented separately without affecting the existing ones.

We are going to create a plugin to implement the necessary tasks for our application. First, create a folder called `wpwa-questions` inside the `/wp-content/plugins` folder and add a new file called `index.php`. Next, we need to add the block comment to define our file as a plugin:

```
/*
Plugin Name: WP Questions
Plugin URI: -
Description: Question and Answer interface for developers
Version: 1.0
Author: Rakhitha Nimesh
Author URI: http://www.innovativephp.com/
License: GPLv2 or later
*/
```

Having created the main plugin file, we can move into creating a custom post type called `wp_question` using the following code snippet. Include the code snippet in your `index.php` file of the plugin:

```
add_action('init', 'register_wp_questions');

function register_wp_questions() {

  $labels = array(
    'name' => __( 'Questions', 'wp_question' ),
    'singular_name' => __( 'Question', 'wp_question' ),
    'add_new'  => __( 'Add New', 'wp_question' ),
    'add_new_item' => __( 'Add New Question', 'wp_question' ),
    'edit_item' => __( 'Edit Questions', 'wp_question' ),
    'new_item'  => __( 'New Question', 'wp_question' ),
    'view_item' => __( 'View Question', 'wp_question' ),
    'search_items' => __( 'Search Questions', 'wp_question' ),
    'not_found' => __( 'No Questions found', 'wp_question' ),
    'not_found_in_trash' => __( 'No Questions found in Trash',
    'wp_question' ),
    'parent_item_colon' => __( 'Parent Question:', 'wp_question'
    ),
    'menu_name' => __( 'Questions', 'wp_question' ),
    );

  $args = array(
    'labels'  => $labels,
    'hierarchical' => true,
    'description' => __( 'Questions and Answers', 'wp_question' ),
    'supports'   => array( 'title', 'editor', 'comments' ),
```

```
        'public'                => true,
        'show_ui'               => true,
        'show_in_menu'          => true,
        'show_in_nav_menus'     => true,
        'publicly_queryable'    => true,
        'exclude_from_search'   => false,
        'has_archive'           => true,
        'query_var'             => true,
        'can_export'            => true,
        'rewrite'               => true,
        'capability_type'       => 'post'
        );

    register_post_type( 'wp_question', $args );
}
```

This is the most basic and default code for custom post type creation and I assume that you are familiar with the syntax. We have enabled title, editor, and comments in the support section of the configuration. These fields will act in the role of question title, question description, and answers. Other configurations contain the default values and hence explanations will be omitted. If you are not familiar, make sure you have a look at the documentation on custom post creation at `http://codex.wordpress.org/Function_Reference/register_post_type`.

Beginner to intermediate level developers and designers tend to include the logic inside the `functions.php` file in the theme. It is considered bad practice as it becomes extremely difficult to maintain as your application becomes larger, so we will be using plugins to add functionality throughout this book, and the drawbacks of the `functions.php` technique will be discussed in the later chapters.

Once the code is included, you will get a new section in the admin area for creating questions. Add a few questions and put some comments in using different users, before we move on to the next stage.

Changing the status of answers

Once users provide their answers, the creator of the question should be able to mark them as correct or incorrect. So, we are going to implement a button for each answer to mark its status. Only the creator of the questions will be able to mark the answers. Once the button is clicked, an AJAX request will be made to store the status of the answer in the database.

First, we need to customize the existing comments list to suit the requirements of the answers list. By default, WordPress will use the `wp_list_comments` function inside the `comments.php` file to show the list of answers for each question. We need to modify the answers list in order to include the answer status button.

So we implement our own version of `wp_list_comments` using a custom function. First, you have to open the `comments.php` file of the theme and look for the call to the `wp_list_comments` function. You should see something similar to the following code:

```
<?php wp_list_comments( array( 'callback' =>
'twentytwelve_comment', 'style' => 'ol' ) ); ?>
```

This function is used to generate a comments list for all types of posts, but we need a slightly modified version to suit the answers list. So we call the `wp_list_comments` function with different arguments, as shown in the following code:

```
<?php
if( get_post_type( $post ) == "wp_question" ){
wp_list_comments(  array( 'type' => 'comment', 'callback' => 'wpwa_
comment_list', 'style' => 'ol' ) );
} else{

wp_list_comments(  array( 'type' => 'comment', 'callback' =>
'twentytwelve_comment', 'style' => 'ol' ) );
}
?>
```

Arguments of the `wp_list_comments` function can be either an array or a string. Here we have preferred array-based arguments over string-based arguments.

Here we include a conditional check for the post type in order to choose the correct answer list generation function. When the post type is `wp_question`, we call the `wp_list_comments` function with the callback parameter defined as `wpwa_comment_list`, which will be the custom function for generating the answers list.

Implementation of the `wpwa_comment_list` function goes inside the `wpwa-questions.php` file of our plugin. This function contains lengthy code, which is not necessary for our explanations, so I'll just be explaining the important sections of the code. It's best to work with the full code for the `wpwa_comment_list` function from the source code folder. Have a look at the following code snippet:

```
function wpwa_comment_list( $comment, $args, $depth ) {
  global $post;

  $GLOBALS['comment'] = $comment;

  // Get current logged in user and author of question
```

```
    $current_user           = wp_get_current_user();
    $author_id              = $post->post_author;
    $show_answer_status     = false;

    // Set the button status for authors of the question
    if ( is_user_logged_in() && $current_user->ID == $author_id ) {
      $show_answer_status = true;
    }

    // Get the correct/incorrect status of the answer
    $comment_id = get_comment_ID();
    $answer_status = get_comment_meta( $comment_id,
    "_wpwa_answer_status", true );

    // Rest of the Code

  }
```

`wpwa_comment_list` is used as the callback function of the comments list and hence it will contain three parameters by default. Remember that the button for marking the answer status should only be visible to the creator of the question.

First, we get the current logged in user from the `wp_get_current_user` function. Also we can get the creator of the question using the global `$post` object. Next, we check whether the logged in user created the question. If so, we set the `$show_answer_status` variable to `true`. Also, we have to retrieve the status of the current answer by passing the comment ID and the `_wpwa_answer_status` key to the `get_comment_meta` function.

Then we will have to include the common code for generating the comments list with the necessary condition checks. Open the `wpwa-questions.php` file of the plugin and go through the rest of the `wpwa_comment_list` function to get an idea of how the comments loop works.

Next, we have to highlight the correct answers for each question and I'll be using an image as the highlighter. In the source code, we use the following code after the header tag to show the correct answer highlighter:

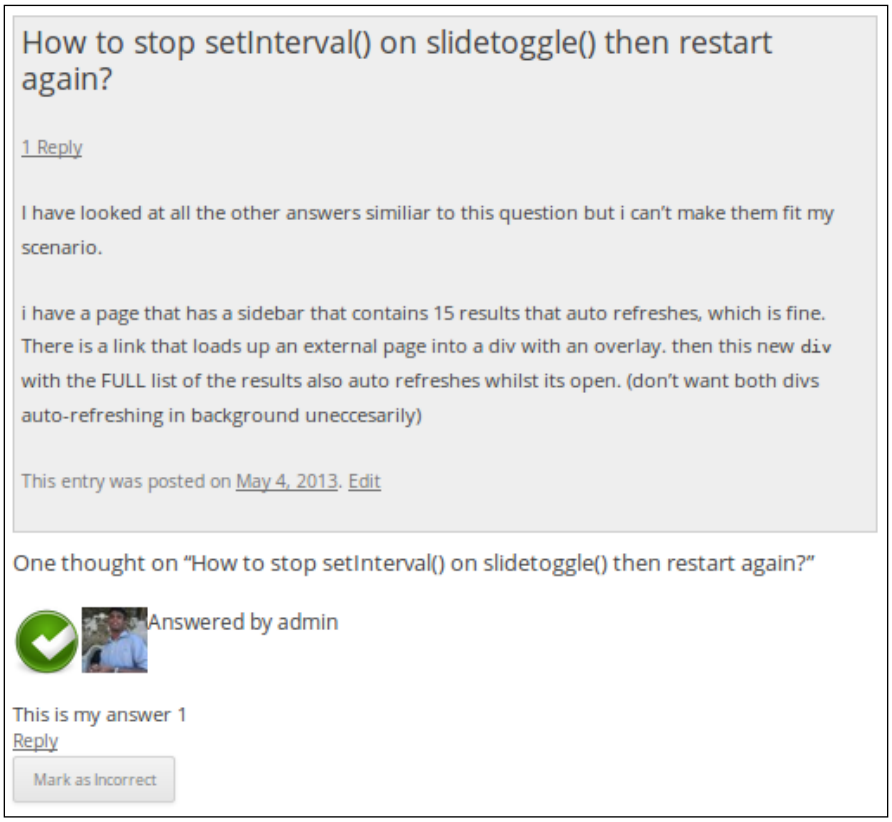
```
<?php
  // Display image of a tick for correct answers
  if ( $answer_status ) {
    echo "<div class='tick'><img src='".plugins_url(
    'img/tick.png', __FILE__ )."' alt='Answer Status' /></div>";
  }
?>
```

In the source code, you will see a DIV element with the class reply for creating the comment reply link. We need to insert our answer button status code right after that, as shown in the following code:

```
<div>
  <?php
  // Display the button for authors to make the answer as correct
  or incorrect
  if ( $show_answer_status ) {

    $question_status = '';
    $question_status_text = '';
    if ( $answer_status ) {
      $question_status = 'invalid';
      $question_status_text = 'Mark as Incorrect';
    } else {
      $question_status = 'valid';
      $question_status_text = 'Mark as Correct';
    }

  ?>
  <input type="button" value="<?php echo $question_status_text; ?>
  "  class="answer-status answer_status-<?php echo $comment_id;
  ?>"
    data-ques-status="<?php echo $question_status; ?>" />
  <input type="hidden" value="<?php echo $comment_id; ?>"
  class="hcomment" />

  <?php
  }
  ?>
</div>
```

If the `$show_answer_status` variable is set to `true`, we get the comment ID, which will be our answer ID, using the `get_comment_ID` function. Then we get the status of the answer as true or false using the `_wpwa_answer_status` key from the `commentmeta` table. Based on the returned value, we define buttons for either `Mark as Incorrect` or `Mark as Correct`. Also, we specify some CSS classes and HTML5 data attributes to be used later with jQuery. Finally, we keep the comment ID in a hidden variable called `hcomment`.

Once you include the code, the button will be displayed for the author of the question, as shown in the following screenshot:



Next, we need to implement the AJAX request for marking the status of the answer as true or false. Before that, we need to see how we can include our scripts and styles into WordPress plugins.

Here is the code for including custom scripts and styles for our plugin. Copy the following code into the `wpwa-questions.php` file of your plugin:

```
function wpwa_frontend_scripts() {

  wp_enqueue_script( 'jquery' );
  wp_register_script( 'wp-questions', plugins_url(
  'js/questions.js', __FILE__ ), array('jquery'), '1.0',true );
  wp_enqueue_script( 'wp-questions' );

  wp_register_style( 'questions', plugins_url(
  'css/questions.css', __FILE__ ) );
```

```
    wp_enqueue_style( 'questions' );

    $config_array = array(
      'ajaxURL' => admin_url( 'admin-ajax.php' ),
      'ajaxNonce' => wp_create_nonce( 'ques-nonce' )
    );

    wp_localize_script( 'wp-questions', 'wpwaconf', $config_array );
  }
  add_action( 'wp_ajax_mark_answer_status',
  'wpwa_mark_answer_status' );
```

WordPress comes in-built with an action hook called `wp_enqueue_scripts`, for adding JavaScript and CSS files. `wp_enqueue_script` is used to include script files into the page while `wp_register_script` is used to add custom files. Since jQuery is built into WordPress, we can just use `wp_enqueue_script` to include jQuery into the page. We also have a custom JavaScript file called `questions.js`, which will contain the functions for our application.

Inside JavaScript files, we cannot access the PHP variables directly. WordPress provides a function called `wp_localize_script`, to pass PHP variables into script files. The first parameter contains the handle of the script for binding data, which will be `wp-questions` in this scenario. The second parameter is the variable name to be used inside JavaScript files to access these values. The third and final parameter will be the configuration array with the values.

Then we can include our `questions.css` file using the `wp_register_style` and `wp_enqueue_style` functions, which will be similar to JavaScript, file inclusion syntax. Now everything is set up properly to create the AJAX request.

Saving the status of answers

Once the author clicks on the button, the status has to be saved to the database as true or false depending on the current status of the answer. Let's go through the jQuery code located inside the `questions.js` file for making the AJAX request to the server.

```
$jq =jQuery.noConflict();

$jq(document).ready( function() {

  $jq(".answer-status").click( function() {

    // Get the button object and current status of the answer
    var answer_button = $jq(this);
```

```
    var answer_status  = $jq(this).attr("data-ques-status");

    // Get the ID of the clicked answer using hidden field
    var comment_id = $jq(this).parent().find(".hcomment").val();
    var data = {
      "comment_id":comment_id,
      "status": answer_status
    };

    // Create the AJAX request to save the status to database
    $jq.post( wpwaconf.ajaxURL, {
      action:"mark_answer_status",
      nonce:wpwaconf.ajaxNonce,
      data : data,
    }, function( data ) {
      if("success" == data.status){
      /*Changes the display text of answer status button and
      toggles the answer status between valid and invalid to be
      displayed in frontend.*/
        if("valid" == answer_status){
          $jq(answer_button).val("Mark as Incorrect");
          $jq(answer_button).attr("data-ques-status","invalid");
        }else{
          $jq(answer_button).val("Mark as Correct");
          $jq(answer_button).attr("data-ques-status","valid");
        }
      }
    }, "json");
  });
});
```

The preceding code creates a basic AJAX request to the `mark_answer_status` action. Most of the code is self-explanatory and code comments will help you to understand the process.

The important thing to note here is that we have used the configuration settings assigned in the previous section, using the `wpwaconf` variable. Once the server returns the response with success status, the button will be updated to contain the new status and display the text.

The next step of this process is to implement the server-side code for handling AJAX requests. First, we need to define AJAX handler functions using the WordPress `add_action` function. Since logged in users are permitted to mark the status, we don't need to implement the `add_action` function for `wp_ajax_nopriv_{action}`.

```
add_action( 'wp_ajax_mark_answer_status',
'wpwa_mark_answer_status' );
```

Implementation of the `wpwa_mark_answer_status` function is given in the following code:

```
function wpwa_mark_answer_status() {
  $data = isset( $_POST['data'] ) ? $_POST['data'] : array();
  $comment_id     = isset( $data["comment_id"] ) ?
  absint($data["comment_id"]) : 0;
  $answer_status  = isset( $data["status"] ) ? $data["status"] :
  0;

  // Mark answers in correct status to incorrect
  // or incorrect status to correct
  if ("valid" == $answer_status) {
    update_comment_meta( $comment_id, "_wpwa_answer_status", 1 );
  } else {
    update_comment_meta( $comment_id, "_wpwa_answer_status", 0 );
  }

  echo json_encode( array("status" => "success") );
  exit;
}
```

We can get the necessary data from the `$_POST` array and use it to mark the status of the answer using the `update_comment_meta` function. This example contains the most basic implementation of the data saving process. In real applications, we need to implement necessary validations and error handling.

Now, the author of the question has the ability to mark answers as correct or incorrect, so we have implemented a nice and simple interface for creating a question-answer site with WordPress. The final task of the process will be the implementation of the questions list.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Generating the question list

Usually WordPress uses the `archive.php` file of the theme for generating post lists of any type. We can use a file called `archive-{post type}.php` for creating different layouts for different post types. In this file we are going to create a customized layout for our questions. Make a copy of the existing `archive.php` file of the TwentyTwelve theme and rename it `archive-wp_question.php`. In this file, you will find the following code section:

```
get_template_part( 'content', get_post_format() );
```

The TwentyTwelve theme uses a separate template for generating the content of each post type. We cannot modify the existing `content.php` file as it affects all kinds of posts, so create custom templates called `content-questions.php` by duplicating the `content.php` file and change the preceding code to the following:

```
get_template_part( 'content-questions', get_post_format() );
```

Finally, we need to consider the implementation of the `content-questions.php` file. In the questions list, only the question title will be displayed and therefore we don't need the content of the post, so we have to either remove or comment the functions `the_excerpt` and `the_content` in the template.

Then we also have to remove the `twentytwelve_entry_meta` function and create our own metadata using the following code:

```
<div class="answer_controls"><?php comments_popup_link(__('No
Answers &darr;', 'responsive'), __('1 Answer &darr;',
'responsive'), __('% Answers &darr;', 'responsive')); ?>
</div>

<div class="answer_controls">
<?php wpwa_get_correct_answers(get_the_ID()); ?>
</div>

<div class="answer_controls">
<?php echo   get_the_date(); ?>
</div>
<div style="clear: both"></div>
```

The first container will make use of the existing `comments_popup_link` function to get the number of answers given for the questions. Then we need to display the number of correct answers for each question. The custom function called `wpwa_get_correct_answers` is created to get the correct answers. The following code contains the implementation of the `wpwa_get_correct_answers` function inside the plugin:

```
function wpwa_get_correct_answers( $post_id ) {
  $args = array(
    'post_id'    => $post_id,
    'status'     => 'approve',
    'meta_key'   => '_wpwa_answer_status',
    'meta_value'=> 1,
  );
  // Get number of correct answers for given question
  $comments = get_comments( $args );
  printf(__('<cite class="fn">%s</cite> correct answers'), count(
  $comments ) );
}
```

We can set the array of arguments to include the conditions for retrieving the approved answers of each post, which also contains the correct answers. The number of results generated from the `get_comments` function will be returned as correct answers. Now you should have a question list similar to the following screenshot:

Throughout this section we looked at how we can convert the existing functionalities of WordPress for building a simple question-answer interface. We took the quick-and-dirty path for this implementation by mixing the HTML and PHP code inside both themes and plugins.

I suggest you go through the `Chapter 1` source code folder and try this implementation on your own test server. This demonstration was created to show the flexibility of WordPress. Some of you might not understand the whole implementation. Don't worry as we will be developing a web application from scratch using detailed explanation in the upcoming chapters.

In the upcoming chapters, we'll see the limitations in this approach in complex web applications and how we can organize things better to write high quality, maintainable code.

Summary

Our main goal was to find out how WordPress fits into web application development. We started this chapter by identifying the CMS functionalities of WordPress. We explored the features and functionalities of popular full stack frameworks and compared them with the existing functionalities of WordPress.

Then we looked at the existing components and modules of WordPress and how each of these components fit into a real-world web application. We also planned the portfolio management application requirements and identified the limitations in using WordPress for web applications.

Finally, we converted the default interface into a question-answer interface in a rapid process using the existing functionalities.

By now, you should know how to:

- Decide whether to choose WordPress for your web application
- Visualize how your requirements fit into the components of WordPress
- Identify and minimize the limitations

In the next chapter, we are going to start the development of the portfolio management application with the user module. Before we go there, I suggest you do some research on user management modules of other frameworks and look at your previous projects to identify the functionalities.

2
Implementing Membership Roles, Permissions, and Features

The success of any web application or website depends heavily on its user base. There are plenty of great web applications that go unnoticed by many people due to the lack of user interaction. As developers, it's our responsibility to build simple and interactive user management processes, as visitors decide whether to stay on or leave a website by looking at the complexity of initial tasks such as registration and login.

In this chapter, we will be mainly concentrating on adapting existing user management functionalities into typical web applications. In order to accomplish our goal, we are going to execute some tasks outside the box to bring user management features from the WordPress core to WordPress themes.

While striving to build a better user experience, we will also take a look at advanced aspects of web application development such as routing, controlling, and custom templating.

In this chapter, we will cover the following topics:

- Introduction to user management
- Understanding user roles and capabilities
- Creating a simple MVC-like process
- Implementing registration on the frontend
- Implementing login on the frontend

Before we get started, I suggest you refer to *Appendix*, *Configurations, Tools, and Resources*, and configure the WordPress environment and setup required for this book. I assume that you are familiar with default user management features and necessary coding techniques in WordPress. So, let's get started.

Introduction to user management

Usually, other popular PHP development frameworks such as Zend, CakePHP, CodeIgniter, and Laravel don't provide built-in user modules. Developers tend to build their own user management modules and use them across many projects of the same framework. WordPress offers a built-in user management system to cater to common user management tasks in web applications. Such things include:

- Managing user roles and capabilities
- Built-in user registration
- Built-in user login
- Built-in forgot password

Developers are likely to encounter these tasks in almost all web applications. On most occasions, these features and functions can be effectively used without significant changes to the code. However, web applications are much more advanced, and hence we might need various customizations on these existing features. It's important to explore the possibility of extending these functions so that they are compatible with advanced application requirements. In the upcoming sections, we are going to learn how to extend these common functionalities to suit varying scenarios.

Preparing the plugin

As developers, we have the option of building a complete application with standalone plugins or using various independent plugins to cater for specific modules. Throughout this book, we will be developing several independent modules as specific plugins to make things simple. We are going to create a specific plugin for user-related functionalities of our application. So, let's get started by creating a new folder named `wpwa_user_manager` inside the `/wp-content/plugins` folder.

Then, create a PHP file inside the folder and save it as `class-wpwa-user-manager.php`. Now it's time to add the plugin definition as shown in the following code:

```
<?php
/*
  Plugin Name: WPWA User Manager
  Plugin URI:
```

```
  Description: User management module for the portfolio management
  application.
  Author: Rakhitha Nimesh
  Version: 1.0
  Author URI: http://www.innovativephp.com/
*/
```

In the previous chapter, we created a plugin with procedural functions calls. Now we are going to go one step further by building an object-oriented plugin. Basically, this plugin consists of one main class that handles the plugin initialization. The following code shows the implementation of the plugin class inside the `wpwa-user-manager.php` file:

```
class WPWA_User_Manager {
  public function __construct() {
    // Initialization code
  }
}
$user_manege = new WPWA_User_Manager();
```

Once the class is defined, we can make an object to initialize the plugin within the same file. All the initialization code resides in the plugin constructor.

Getting started on user roles

In simple terms, user roles define the types of users in a system. WordPress offers built-in functions for working with every aspect of user roles. In this section, we are going to look at how we can manage these tasks by implementing the user roles for our application. We can create a new user role by calling the `add_role` function. The following code illustrates the basic form of user role creation:

```
$result = add_role( 'role_name', 'Display Name', array(
  'read' => true,
  'edit_posts' => true,
  'delete_posts' => false,
) );
```

The first parameter takes the role name, which is a unique key to identify the role. The second parameter will be the display name, which will be shown in the admin area. The final parameter will take the necessary capabilities of the user role. In this scenario, `read`, `edit_posts`, and `delete_posts` will be the capabilities, while `true` and `false` are used to enable and disable status.

Creating user roles for a application

As planned earlier, we will need three types of user roles for our application to handle subscribers, developers, and members. So, we will update our plugin by adding a specific function to create the user roles as shown in the following code:

```
public function add_application_user_roles() {
  add_role( 'follower', 'Follower', array( 'read' => true ) );
  add_role( 'developer', 'Developer', array( 'read' => true ) );
  add_role( 'member', 'Member', array( 'read' => true ) );
}
```

User roles for applications are created with the default capability of read, which is used by all user roles in WordPress. Initialization of this function should be done inside the constructor of our plugin.

What is the best action for adding user roles?

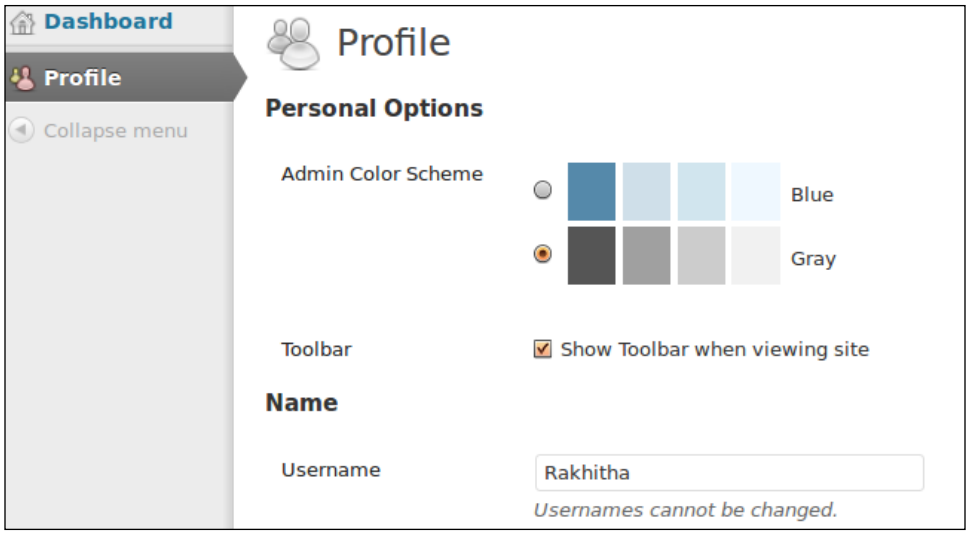
As these user roles will be saved in a database as settings, only a single call to this function is required throughout the life cycle of an application. Plugin activation is the most suitable option to call these kinds of functions for eliminating duplicate executions. So, we need to include the call to the `add_application_users` function inside the constructor as shown in the following code:

```
register_activation_hook( __FILE__ , array( $this,
'add_application_user_roles' ) );
```

The `register_activation_hook` function will be called when the plugin is activated, and hence avoids the duplicate calls to the database.

A good rule of thumb is to prevent the inclusion of such settings inside the `init` action as it will get executed in each request, making unnecessary performance overhead.

You might have noticed that all three user roles of the application are created with the read capability. WordPress is built for creating websites, and hence most of the default capabilities will be related to CMS features. In web applications, we need custom capabilities more often than not. Therefore, we can keep the basic read capability and add new custom capabilities as we move on. All the users will get a very basic admin area containing the dashboard and profile information as shown in the following screenshot:



Knowing the default roles

WordPress comes with six built-in user roles, including superadmin, which will not be displayed in the user creation screen by default. As a developer, it's important to know the functionality of each of these types in order to use them in web applications. First, we'll take a look at the default user roles and their functionalities:

- **Superadmin**: A user with this role has the administration permission in WordPress multisite implementation
- **Admin**: A user with this role has the permission to all administration activities inside a single site
- **Editor**: A user with this role can create, publish, and manage posts, including the posts of other users
- **Author**: A user with this role can create and publish his/her own posts
- **Contributor**: A user with this role can create posts but cannot publish on his/her own
- **Subscriber**: A user with this role can read posts and manage profiles

As you can see, most of the existing user types are used for blogging and content management functionality. Therefore, we might need to create our own user roles for web applications apart from the default superadmin and admin user roles.

How to choose between default and custom roles?

This is an interesting question which doesn't have a correct answer. Choosing between these two types of roles naturally comes with experience. First, you need to figure out how these built-in roles relate to your application. Let's consider two scenarios to help demonstrate the practical usage of these user roles.

Scenario 1

Usually, the roles such as editor, author, and contributor are mainly focused on publishing and managing blog posts. If you are developing an online shopping cart, these roles will not have any relation to the roles of such applications.

Scenario 2

Now think of a scenario where we have a job posting site with three access levels called admin, companies, and individuals. Here, individuals can create job posts, while approvals are given by the admin. So, they are similar to contributors. Similar to authors, companies can create and publish their own job posts. Admin can play the role of the editor as well in the default system.

Even though we can match certain aspects of our portfolio application roles with the existing roles, we are going to work with custom roles to keep things simple and clear. All application users will be created as custom roles with read capability by default and necessary capabilities will be added as we move on.

It doesn't matter whether you choose existing ones or new ones as long as you are comfortable and the roles have a specific meaning within your application. Since we choose custom roles, it's not necessary to keep the unused default roles. Let's see how we can remove roles when necessary.

Removing existing user roles

We should have the ability to remove existing or custom user roles when necessary. WordPress offers the `remove_role` function for deleting both custom and existing user roles. In this case, we want to get rid of existing user roles. Also, there can be situations where you use a plugin with specific user roles and suddenly you want to disable the functionality of the plugin. In both cases, we need to remove the user roles from the database. Let's create a function that removes unnecessary user roles from the system as described in the following code:

```
public function remove_application_user_roles(){
  remove_role( 'author' );
  remove_role( 'editor' );
  remove_role( 'contributor' );
  remove_role( 'subscriber' );
}
```

As mentioned earlier, the `remove_role` function involves database operations, and hence it's wise to use it with the `register_activation_hook` function as shown in the following code:

```
register_activation_hook( __FILE__, array($this,
'remove_application_user_roles') );
```

In this section, we looked at how user roles work in WordPress. Now we need to see how we can associate capabilities with these user roles.

Understanding user capabilities

Capabilities can be considered as tasks that users are permitted to perform inside the application. A single user role can perform many capabilities, while a single capability can be performed by many user roles. Typically, we use the term access control for handling capabilities in web applications. Let's see how capabilities work inside WordPress.

Creating your first capability

Capabilities are always associated with user roles, and hence we cannot create new capabilities without providing a user role. Let's look at the following code for associating a custom capability with a follower user role created in the *Creating user roles for a application* section:

```
public function add_application_user_capabilities(){
  $role = get_role( 'follower' );
  $role->add_cap( 'follow_developer_activities' );
}
```

First, we need to retrieve the user role as an object using the `get_role` function. Then, we can associate a new or existing capability using the `add_cap` function. We need to continue this process for each user role until we assign all the capabilities to necessary user levels. Also, make sure to call this function on activation with the `register_activation_hook` function.

Understanding default capabilities

You can find over fifty built-in capabilities in the WordPress default database. Most of these capabilities are focused on providing permissions related to website or blog creation. Therefore, it's a must to create our own capabilities when developing web applications. If you are curious to learn, you can look at the `wp_user_roles` option inside the `wp_options` table for all the available user roles and their capabilities.

```
select option_value from wp_users where
option_name='wp_user_roles'
```

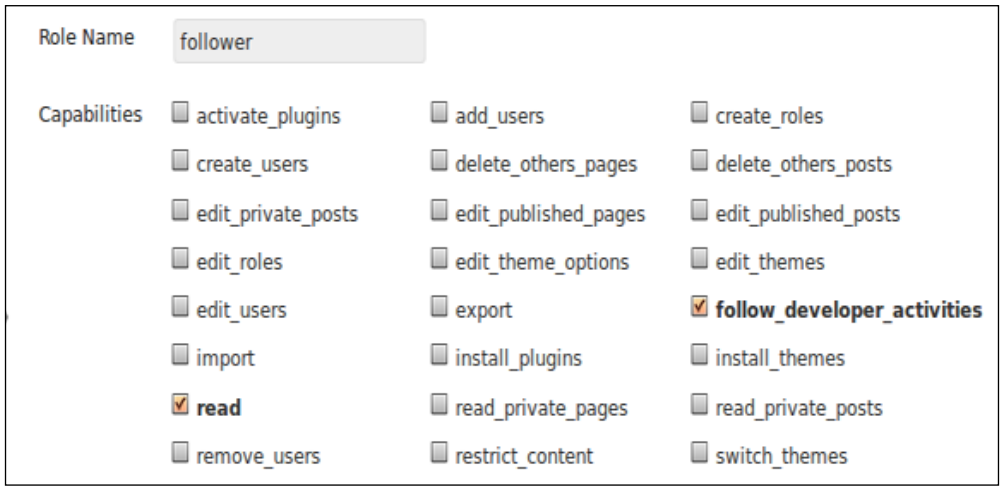
You should see a serialized array like the following:

```
a:10:{s:13:"administrator";a:2:{s:4:"name";s:13:"Administrator";s:
12:"capabilities";a:67:{s:13:"switch_themes";b:1;s:11:"edit_themes
";b:1;s:16:"activate_plugins";b:1;s:12:"edit_plugins";b:1;s:10:"ed
it_users";b:1;s:10:"edit_files";b:1;s:14:"manage_options";b:1;s:17
:"moderate_comments";b:1;s:17:"manage_categories";b:1;s:12:"manage
_links";b:1;s:12
```

A part of the value contained in the `wp_user_roles` row is displayed in the preceding code. It's quite confusing and not practical to understand the capabilities of each user role by looking at this serialized array. Therefore, we can take advantage of an existing WordPress plugin to view and manage user roles and capabilities.

There are plenty of great and free plugins for managing user roles and permissions. My favorite is the Members plugin by *Justin Tadlock*, as it's quite clean and simple. You can grab a copy of this plugin at `http://wordpress.org/plugins/members/`.

Let's see how capabilities are displayed for the follower role in our application using the following screenshot of the plugin:



All the capabilities that are assigned to specific user roles will be ticked by default. As expected, the **follow_developer_activities** capability added in the previous section is successfully assigned to the follower role.

Up to now, we have learned how to use WordPress roles and capabilities in the context of web applications. We will be updating the capabilities while creating new functionalities in the upcoming chapters. Next, we are going to see how user registration works in WordPress.

Registering application users

The administration panel is a built-in WordPress framework that allows us to login through the admin screen. Therefore, we have a registration area which can be used effectively to add new users by providing a username and e-mail. In web applications, registration can become complex compared with the simple registration process in WordPress. Let's consider some typical requirements of a web application registration process in comparison with WordPress:

- **A user-friendly interface**: Applications can have different types of user roles. Until the registration is complete, everyone is treated as a normal application user with the ability to view public content. Typically, users are used to seeing fancy registration forms inside the main site rather than a completely different login area like WordPress. Therefore, we need to explore the possibilities of adding the WordPress registration to the frontend.
- **Requesting detailed information**: Most web applications will have at least 4-5 fields in the user registration form for grabbing detailed information about users. Therefore, we need to look for the possibility of adding new fields to the existing WordPress registration form.
- **Activating user accounts**: In some applications, you will be asked to verify and activate your account after successful registration. WordPress doesn't offer this feature by default. Hence, we need to see how we can extend the current process to include user activation.

These are the most common requirements of the registration process in web applications. Complex applications may have more requirements in this process. Therefore, we need to extend the WordPress registration process in order to cater for varying requirements. In the next section, we are going to address the issues mentioned here by creating a WordPress registration from the frontend.

Implementing the frontend registration

Fortunately, we can make use of the existing functionalities to implement registration from the frontend. We can use regular HTTP requests or AJAX-based techniques for implementing this feature. In this book, I am going to focus on a normal process instead of using AJAX. Our first task is creating the registration form on the frontend.

There are various ways to implement such forms on the frontend. Let's look at some of the possibilities as described in the following sections:

- Shortcode implementation
- Page template implementation
- Custom template implementation

Now let's look at the implementation of each of these techniques.

Shortcode implementation

Shortcodes are the quickest way of adding dynamic content to your pages. In this situation, we need to create a page for registration. Therefore, we need to create a shortcode that generates the registration form as shown in the following code:

```
add_shortcode( "register_form", "display_register_form" );
function display_register_form(){
  $html = "HTML for registration form";
  return $html;
}
```

Then, you can add the shortcode inside the created page using the following code snippet to display the registration form:

```
[register_form]
```

Pros and cons of using shortcodes

- Easy to implement in any part of your application
- Hard to manage the template code assigned using the PHP variables
- Possibility of getting the shortcode deleted from the page by mistake

Page template implementation

Page templates are a widely used technique in modern WordPress themes. We can create a page template to embed the registration form. Consider the following code for a sample page template:

```
/*
* Template Name :  Registration
*/
HTML code for registration form
```

Next, we have to copy the template inside the theme folder. Finally, we can create a page and assign the page template to display the registration form. Now let's look at the pros and cons of this technique.

Pros and cons of page templates

- A page template is more stable than a shortcode.
- Generally, page templates are associated to the look of the website rather than providing dynamic forms. Full width page, 2 column page, and left sidebar page are some of the common implementations of page templates.
- A page template is managed separately from logic without using PHP variables.
- Whether to use this technique depends on the theme and the need to update on theme switching.

Custom template implementation

Experienced web application developers will always look to separate business logic from the view templates. This will be the perfect technique for such people. In this technique, we create our own independent templates by intercepting the WordPress default routing process. Implementation of this technique starts from the next section on routing.

Building a simple router for user modules

Routing is one of the most important aspects in advanced application development. We need to figure out ways of building custom routes for specific functionalities. In this scenario, we are going to create a custom router to handle all the user-related functionalities of our application.

Let's list the requirements for building a router:

- All the user-related functionalities should go through a custom URL such as `http://www.example.com/user`
- Registration should be implemented at `http://www.example.com/user/register`
- Login should be implemented at `http://www.example.com/user/login`
- Activation should be implemented at `http://www.example.com/user/activate`

Make sure to set up your permalinks structure to post name for the examples in this book. If you prefer a different permalinks structure, you will have to update the URLs and routing rules accordingly.

As you can see, the user section is common for all the functionalities. The second URL segment changes dynamically based on the functionality. In MVC terms, the user acts as the controller and the next URL segment (`register`, `login`, and `activate`) acts as the action. Now let's see how we can implement a custom router for the given requirements.

Creating the routing rules

There are various ways and action hooks to create custom rewrite rules. We are going to choose the `init` action to define our custom routes for the user section as shown in the following code:

```
public function manage_user_routes() {
  add_rewrite_rule( '^user/([^/]+)/?',
  'index.php?control_action=$matches[1]', 'top' );
}
```

Based on the discussed requirements, all the URLs for the user section will follow the `/user/custom action` pattern. Therefore, we define the regular expression to match all the routes in the user section. Redirection is made to the `index.php` file with a query variable named `control_action`. This variable will contain the URI segment after the `/user` segment. The third parameter of the `add_rewrite_rule` function will decide whether to check this rewrite rule before the existing rules or after the existing rules. The value of the `top` parameter will give higher precedence, while the value of the `bottom` parameter will give lower precedence.

We need to complete two other tasks to get these rewriting rules to take effect, as follows:

- Add query variables to WordPress: `query_vars`
- Flush the rewriting rules

Adding query variables

WordPress doesn't allow you to use any type of variable in the query string. It will check for query variables within the existing list and all other variables will be ignored. Whenever you want to use a new query variable, make sure you add it to the existing list. To do this, we first need to update our constructor with the following filter to customize query variables:

```
add_filter( 'query_vars', array( $this,
'manage_user_routes_query_vars' ) );
```

This filter on `query_vars` will allow us to customize the existing variables list by adding or removing entries from an array. Now consider the implementation for adding a new query variable:

```
public function manage_user_routes_query_vars( $query_vars ) {
  $query_vars[] = 'control_action';
  return $query_vars;
}
```

As this is a filter, existing `query_vars` will be passed as an array. We modify the array by adding a new query variable named `control_action` and return the list. Now we have the ability to access this variable from the URL.

Flushing the rewriting rules

Once rewrite rules are modified, it's a must to flush the rules in order to prevent the 404 page generation. Flushing existing rules is a time consuming task which impacts the performance of the application, and hence should be avoided in repetitive actions such as `init`. It's recommended to perform such tasks in plugin activation as we did earlier with user roles and capabilities. So, let's implement the function for flushing rewrite rules on plugin activation:

```
public function flush_application_rewrite_rules() {
  flush_rewrite_rules();
}
```

As usual, we need to update the constructor to include the following action to call the `flush_application_rewrite_rules` function:

```
register_activation_hook( __FILE__, array( $this,
'flush_application_rewrite_rules' ) );
```

Now go to the admin panel, deactivate the plugin, and activate the plugin again. Then go to the `http://www.example.com/user/login` URL and check if it works. Unfortunately, you will still get the 404 error for the request.

You might be wondering what went wrong. Let's go back and think about the process in order to understand the issue. We flushed the rules on plugin activation. So, the new rules should be persisted successfully. However, we define the rules on the `init` action, which is only executed after the plugin is activated. Therefore, new rules will not be available at the time of flushing.

Consider the updated version of the `flush_application_rewrite_rules` function for a quick fix to our problem:

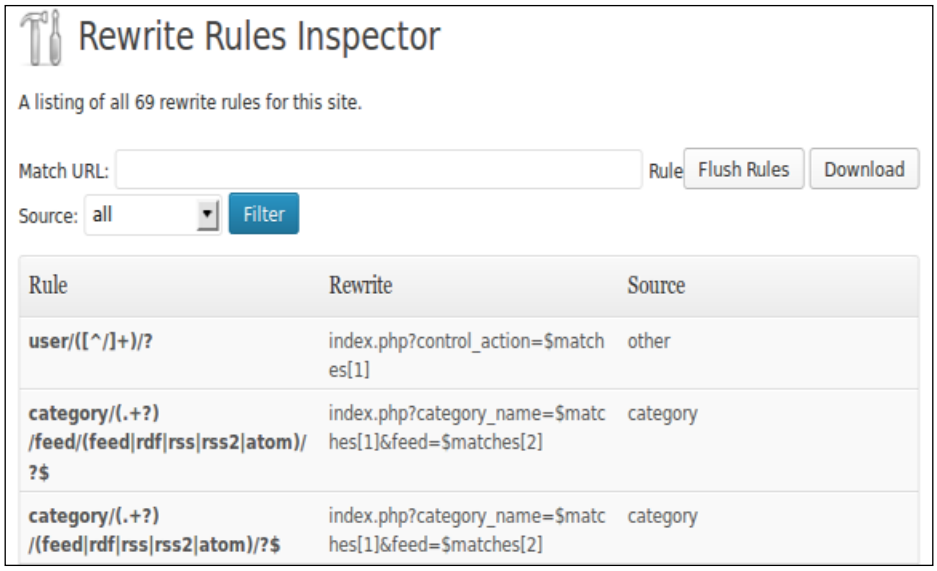
```
public function flush_application_rewrite_rules() {
  $this->manage_user_routes();
  flush_rewrite_rules();
}
```

We call the `manage_user_routes` function on plugin activation followed by the call to `flush_rewrite_rules`. So, the new rules are generated before the flushing is executed. Follow the previous process once again. Now you won't get a 404 page since all the rules have taken effect.

You can get the 404 error due to the modification in rewriting rules and not flushing them properly. In such situations, go to the permalinks section on the settings page and click on the **Save Changes** button to flush the rewrite rules manually.

Now we are ready with our routing rules for user functionalities. It's important to know the existing routing rules of your application. Even though we can have a look at the routing rules from the database, it's difficult to decode the serialized array as we encountered in previous section.

So, I recommend you use the free plugin named `Rewrite Rules Inspector`. You can grab a copy at `http://wordpress.org/plugins/rewrite-rules-inspector/`. Once installed, this plugin allows you to view all the existing routing rules as well as offers a button to flush the rules as shown in the following screenshot:



Controlling access to your functions

We have a custom router that handles the URLs of the user section of our application. Next, we need a controller to handle the requests and generate the template for the user. This works similar to the controllers in the MVC pattern. Even though we have changed the default routing, WordPress will look for an existing template to be sent back to the user. Therefore, we need to intercept this process and create our own templates. WordPress offers an action hook named `template_redirect` for intercepting requests. So, let's implement our frontend controller based on `template_redirect`. First, we need to update the constructor with the `template_redirect` action as shown in the following code:

```
add_action( 'template_redirect', array( $this, 'front_controller'
) );
```

Now let's take a look at the implementation of the `front_controller` function using the following code:

```
public function front_controller() {
  global $wp_query;
```

```
    $control_action = isset ( $wp_query->query_vars
    ['control_action'] ) ? $wp_query->query_vars
    ['control_action'] : ''; ;
    switch ( $control_action ) {
      case 'register':
        do_action( 'wpwa_register_user' );
        break;
    }
  }
```

We will be handling custom routes based on the value of the `control_action` query variable assigned in the previous section. The value of this variable can be grabbed through the global `query_vars` array of the `$wp_query` object. Then, we can use a simple switch statement to handle the controlling based on the action.

The first action to be considered will be register as we are in the registration process. Once the `control_action` query is matched with registration, we call a handler function using `do_action`. You might be confused why we use `do_action` in this scenario. So, let's consider the same implementation in a normal PHP application where we don't have the `do_action` hook:

```
switch ( $control_action ) {
  case 'register':
    $this->register_user();
    break;
}
```

This is a typical scenario where we call a function within the class or in an external class to implement the registration. In the previous code, we are calling a function within the class but with the `do_action` hook instead of the usual function call.

What are the advantages of using do_action?

WordPress action hooks define specific points in the execution process where we can develop custom functions to modify the existing behavior. In this scenario, we are calling the `wpwa_register_user` function within the class using `do_action`.

Unlike websites or blogs, web applications need to be extendable with future requirements. Think of a situation where we only allow Gmail addresses for user registration. This Gmail validation is not implemented in the original code. Therefore, we need to change the existing code to implement the necessary validations. Changing a working component is considered bad practice in application development. Let's see why it's considered as bad practice by looking at the definition of the open/closed principle by Wikipedia.

The open/closed principle states "software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification"; that is, such an entity can allow its behavior to be modified without altering its source code.

This is especially valuable in a production environment where changes to the source code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product. The code obeying the principle doesn't change when it is extended and therefore needs no such effort.

WordPress action hooks come to our rescue in this scenario. We can define an action for registration using the `add_action` function as shown in the following code:

```
add_action( 'wpwa_register_user', array( $this, 'register_user' )
);
```

Now you can implement this action multiple times using different functions. In this scenario, `register_user` will be our primary registration handler. For Gmail validation, we can define another function using the following code:

```
add_action( 'wpwa_register_user', array( $this,
'validate_gmail_registration') );
```

Inside this function, we can make the necessary validations as shown in the following code:

```
public function validate_user(){
  // Code to validate user
  // remove registration function if validation fails
  remove_action( 'wpwa_register_user', array( $this,
  'register_user' ) );
}
```

Now the `validate_user` function is executed before the primary function. So, we can remove the primary registration function if something goes wrong in validation. With this technique, we have the capability of adding new functionalities as well as changing existing functionalities without affecting the already written code.

We have now implemented a simple controller, which can be quite effective in developing web application functionalities. In the upcoming sections, we are going to continue the process of implementing registration on the frontend with custom templates.

Creating custom templates

Themes provide a default set of templates to cater to the existing behavior of WordPress. Here, we are trying to implement a custom template system to suit web applications. So, our first option is to include the template files directly inside the theme. Personally, I don't like this option due to two reasons:

- Whenever we switch a theme, we have to move the custom template files to the new theme. So, our templates become theme dependent.
- In general, all the existing templates are related to CMS functionality. Mixing custom templates with the existing ones becomes hard to manage.

As a solution to the afore mentioned concerns, we are going to implement the custom templates inside the plugin. First, create a folder inside the current plugin folder and name it `templates` to get things started.

Designing the registration form

We need to design a custom form for frontend registration containing the default header and footer. The entire content area will be used for registration and the default sidebar will be omitted for this screen. Create a PHP file named `register.php` inside the `templates` folder with the following code:

```
<?php get_header(); ?>
<div id="custom_panel">
<?php
if( count( $errors ) > 0) {
  foreach( $errors as $error ){
    echo "<p class='frm_error'>$error</p>";
  }
}
?>
HTML Code for Form
</div>
<?php get_footer(); ?>
```

We can include the default header and footer using the `get_header` and `get_footer` functions, respectively. After the header, we include a display area for the error messages generated in registration. Then, we have the HTML form as shown in the following code:

```
<form id='registration-form' method='post'
action='<?php echo get_site_url() . '/user/register'; ?>'>
  <ul>
    <li>
```

```
        <label class='frm_label' for='Username'>Username</label>
        <input class='frm_field' type='text' id='username'
        name='user' value='' />
      </li>
      <li>
        <label class='frm_label' for='Email'>E-mail</label>
        <input class='frm_field' type='text' id='email' name='email'
        value='' />
      </li>
      <li>
        <label class='frm_label' for='User Type'>User Type</label>
        <select class='frm_field' name='user_type'>
          <option value='follower'>Follower</option>
          <option value='developer'>Developer</option>
          <option value='member'>Member</option>
        </select>
      </li>
      <li>
        <label class='frm_label' for=''> </label>
        <input type='submit' value='Register' />
      </li>
    </ul>
  </form>
```

As you can see, the form action is set to a custom route named `user/register` to be handled through the front controller. Also, we have added an extra field named `user_type` for choosing the preferred user type on registration. We will get an output similar to the following screenshot once we access the `/user/register` route in the browser:

Once the form is submitted, we have to create the user based on the application requirements.

Planning the registration process

In this application, we have opted to build a complex registration process in order to understand the typical requirements of web applications. So, it's better to plan it upfront before moving on to the implementation. Let's build a list of requirements for registration:

- The user should be able to register as any of the given user roles
- An activation code needs to be generated and sent to the user
- The default notification on successful registration needs to be customized to include the activation link
- The user should activate his/her account by clicking on the link

So, let's begin the task of registering users by displaying the registration form as given in the following code:

```
public function register_user() {
  if ( !is_user_logged_in() ) {
    include dirname(__FILE__) . '/templates/register.php';
    exit;
  }
}
```

Once the user requests `/user/register`, our controller will call the `register_user` function using the `do_action` call. In the initial request, we need to check whether the user is already logged in using the `is_user_logged_in` function. If not, we can directly include the registration template located inside the `templates` folder to display the registration form.

WordPress templates can be included using the `get_template_part` function. However, it doesn't work like a typical template library as we cannot pass data to the template. In this technique, we are including the template directly inside the function. Therefore, we have access to the data inside this function.

Handling registration form submission

Once the user fills the data and clicks on the **Register** button, we have to execute quite a few tasks in order to register a user in the WordPress database. Let's figure out the main tasks for registering a user:

- Validating form data
- Registering the user details
- Creating and saving the activation code
- Sending an e-mail notification with an activation link

In the registration form, we specified the action as `/user/register`, and hence the same `register_user` function will be used to handle form submission. Validating the user data is one of the main tasks in form submission handling. So, let's take a look at the `register_user` function with the updated code:

```
public function register_user() {
  if ( $_POST ) {
    $errors = array();
    $user_login = ( isset ( $_POST['user'] ) ?
    $_POST['user'] : '' );
    $user_email = ( isset ( $_POST['email'] ) ?
    $_POST['email'] : '' );
    $user_type  = ( isset ( $_POST['user_type'] ) ?
    $_POST['user_type'] : '' );

    // Validating user data
    if ( empty( $user_login ) )
      array_push( $errors, 'Please enter a username.' );

        if ( empty( $user_email ) )
          array_push( $errors, 'Please enter e-mail.' );

        if ( empty( $user_type ) )
          array_push( $errors, 'Please enter user type.' );
  }
  // Including the template
}
```

First, we check whether the request is made as `POST`. Then, we get the form data from the `POST` array. Finally, we check the passed values for empty conditions and push the error messages to the `$errors` variable created at the beginning of this function.

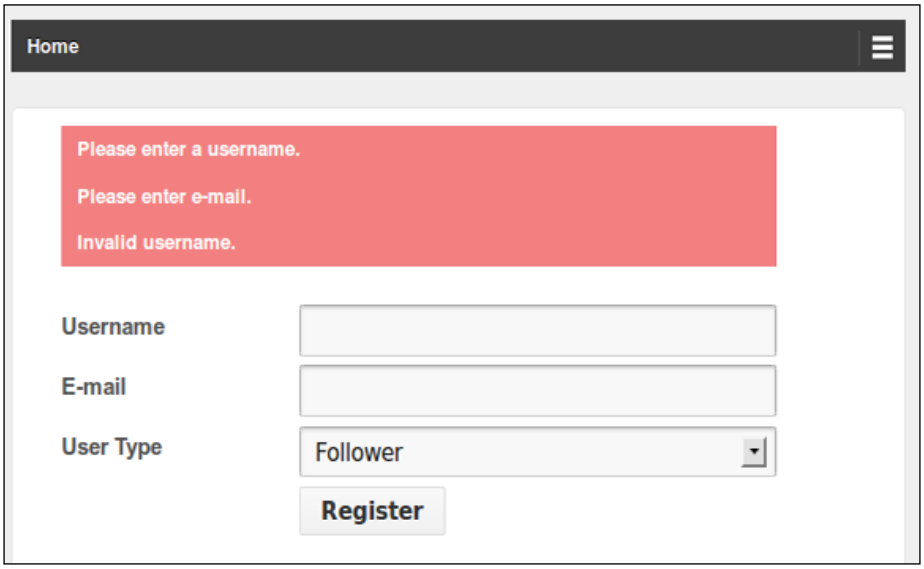
Now we can move into more advanced validations inside the `register_user` function, as shown in the following code:

```
$sanitized_user_login = sanitize_user( $user_login );

if ( !empty($user_email) && !is_email( $user_email ) )
  array_push( $errors, 'Please enter valid email.');
elseif ( email_exists( $user_email ) )
  array_push( $errors, 'User with this email already registered.'
  );

if ( empty( $sanitized_user_login ) || !validate_username(
$user_login ) )
  array_push( $errors, 'Invalid username.' );
elseif ( username_exists( $sanitized_user_login ) )
  array_push( $errors, 'Username already exists.' );
```

First, we use the existing `sanitize_user` function to remove unsafe characters from the username. Then, we make validations on e-mail to check whether it's valid and whether it exists in the system. Both the `email_exists` and `username_exists` functions check for the existence of e-mail and username from the database. Once all the validations are completed, the errors array will be either empty or filled with error messages.

In this scenario, we choose to go with the most essential validations for a registration form. You can add more advanced validation in your implementations in order to minimize the potential security threats.

In case we get validation errors in the form, we can directly print the contents of the error array on top of the form as it's visible in the registration template. Here is a preview of our registration screen with generated error messages:



Also, it's important to repopulate the form values once errors are generated. We are using the same function for loading the registration form and handling the form submission. Therefore, we can directly access the POST variables inside the template to echo the values as shown in the updated registration form:

```
<form id='registration-form' method='post'
action='<?php echo get_site_url() . '/user/register'; ?>'>
  <ul>
    <li>
```

```
        <label class='frm_label' for='Username'>Username</label>
        <input class='frm_field' type='text' id='username'
        name='user' value='<?php echo isset( $user_login ) ?
        $user_login : ''; ?>' />
      </li>
      <li>
        <label class='frm_label' for='Email'>E-mail</label>
        <input class='frm_field' type='text' id='email' name='email'
        value='<?php echo isset( $user_email ) ? $user_email : '';
        ?>' />
      </li>
      <li>
        <label class='frm_label' for='User Type'>User Type</label>
        <select class='frm_field' name='user_type'>
          <option <?php echo (isset( $user_type ) && $user_type ==
          'follower') ? 'selected' : ''; ?>
          value='follower'>Follower</option>
          <option <?php echo (isset( $user_type ) && $user_type ==
          'developer') ? 'selected' : ''; ?>
          value='developer'>Developer</option>
          <option <?php echo (isset( $user_type ) && $user_type ==
          'member') ? 'selected' : ''; ?>
          value='member'>Member</option>

        </select>
      </li>
      <li>
        <label class='frm_label' for=''> </label>
        <input type='submit' value='Register' />
      </li>
    </ul>
  </form>
```

Now let's look at the success path, where we don't have any errors, by looking at the remaining sections of the `register_user` function:

```
  if ( empty( $errors ) ) {
    $user_pass  = wp_generate_password();
    $user_id    = wp_insert_user( array('user_login' =>
    $sanitized_user_login,
    'user_email' => $user_email,
    'role' => $user_type,
    'user_pass' => $user_pass)
    );
    if ( !$user_id ) {
      array_push( $errors, 'Registration failed.' );
```

```
    } else {
      $activation_code = $this->random_string();

      update_user_meta( $user_id, 'activation_code',
      $activation_code );
      update_user_meta( $user_id, 'activation_status', 'inactive' );
      wp_new_user_notification( $user_id, $user_pass,
      $activation_code );

      $success_message = "Registration completed successfully.
      Please check your email for activation link.";
    }

    if ( !is_user_logged_in() ) {
      include dirname(__FILE__) . '/templates/login.php';
      exit;
    }
  }
```

We can generate the default password using the `wp_generate_password` function. Then we can use the `wp_insert_user` function with respective parameters generated from the form to save the user details into the database.

The `wp_insert_user` function will be used to update the current user details or add new users' details to the application. Make sure you are not logged in while executing this function. Otherwise, your admin will suddenly change into another user type after using this function.

If the system fails to save the user details, we create a registration fail message and assign it to the `$errors` variable as we did earlier. Once the registration is successful, we generate a random string as the activation code. You can use any function here to generate a random string.

Then, we update the user with the activation code and set activation status as `inactive` for the moment. Finally, we use the `wp_new_user_notification` function to send an e-mail containing the registration details. By default, this function takes the user ID and password and sends login details. In this scenario, we have a problem as we need to send an activation link with the e-mail.

This is a pluggable function, and hence we can create our own implementation of this function to override the default behavior. Since this is a built-in WordPress function, we cannot declare it inside our plugin class. So, we are going to implement it as a standalone function inside our main plugin file. The complete source code for this function will not be included here as it is quite extensive. I'll explain the modified code from the original function, and you can have a look at the source code for the complete code:

```
$activate_link = site_url().
"/user/activate/?activation_code=$activate_code";
$message = __('Hi there,') . "\r\n\r\n";
$message .= sprintf(__("Welcome to %s! Please activate your
account using the link:"), get_option('blogname')) . "\r\n\r\n";
$message .= sprintf(__('<a href="%s">%s</a>'), $activate_link,
$activate_link) . "\r\n";
```

We create a custom activation link using the third parameter passed to this function. Then, we modify the existing message to include the activation link. That's about all we need to change from the original function. Finally, we set the success message to be passed to the login screen.

Now let's move back to the `register_user` function. Once the notification is sent, the registration process is completed and the user will be redirected to the login screen. Once the user has the e-mail in the inbox, he/she can use the activation link to activate the account.

Activating system users

Once a user clicks on the activation link, he/she will be redirected to the `/user/activate` URL of the application. So, we need to modify our controller with a new case for activation as shown in the following code:

```
case 'activate':
  do_action( 'wpwa_activate_user' );
```

As usual, the definition of `add_action` goes in the constructor as shown in the following code:

```
add_action( 'wpwa_activate_user', array( $this, 'activate_user' )
);
```

Next, we can have a look at the actual implementation of the `activate_user` function:

```
public function activate_user() {

  $activation_code = isset( $_GET['activation_code'] ) ?
  $_GET['activation_code'] : '';
  $message = '';

  // Get activation record for the user
  $user_query = new WP_User_Query(
    array(
      'meta_key' => 'activation_code',
      'meta_value' => $activation_code
    )
  );

  $users = $user_query->get_results();

  // Check and update activation status
  if ( !empty($users) ) {
    $user_id = $users[0]->ID;
    update_user_meta( $user_id, 'activation_status', 'active' );
    $message = 'Account activated successfully. ';
  } else {
    $message = 'Invalid Activation Code';
  }

  include dirname(__FILE__) . '/templates/info.php';
  exit;
}
```

We get the activation code from the link and query the database to find a matching entry. If no records are found, we set the message as activation failed. Otherwise, we update the activation status of the matching user to activate the account. Upon activation, the user will be informed with a message using the `info.php` template, which consists of a very basic template like the following code:

```
<?php  get_header(); ?>
<div id='info_message'>
<?php echo $message; ?>
</div>
<?php get_footer(); ?>
```

Once a user visits the activation page on the `/user/activation` URL, information will be given to the user as shown in the following screenshot:

We have successfully created and activated a new user. The final task of this process is to authenticate and log the user into the system. Let's see how we can create the login functionality.

Creating a login on the frontend

Frontend login can be found in many WordPress websites, including small blogs. Usually, we place the login form in the sidebar of the website. In web applications, user interfaces are complex and different compared to normal websites. Hence, we are going to implement the full page login screen as we did with registration. First, we need to update our controller with another case for login as shown in the following code:

```
switch ( $control_action ) {
  // Other cases
  case 'login':
    do_action( 'wpwa_login_user' );
  break;
}
```

This action will be executed once the user enters `/user/login` in the browser URL to display the login form. The design form for login will be located in the `templates` directory as a separate template named `login.php`. Here is the implementation of the login form design with the necessary error messages:

```
<?php get_header(); ?>

<div id='custom_panel'>
  <?php
    if (count($errors) > 0) {
      foreach ($errors as $error) {
        echo "<p class='frm_error'>$error</p>";
      }
    }
    if( isset( $success_message ) && $success_message != ""){
      echo "<p class='frm_success'>$success_message</p>";
    }
  ?>
  <form method='post' action='<?php echo site_url();
  ?>/user/login' id='login_form' name='login_form'>
    <ul>
      <li>
        <label class='frm_label' for='username'>Username</label>
        <input class='frm_field' type='text'  name='username'
        value='<?php echo isset( $username ) ? $username : ''; ?>'
        />
      </li>
      <li>
        <label class='frm_label' for='password'>Password</label>
        <input class='frm_field' type='password' name='password'
        />
      </li>
      <li>
        <label class='frm_label' > </label>
        <input  type='submit'  name='submit' value='Login' />
      </li>
    </ul>
  </form>
</div>
<?php get_footer(); ?>
```

Similar to the registration template, we have the header, error messages, HTML form, and footer in this template. We have to point the action of this form to `/user/login`. The remaining code is self-explanatory, and hence I am not going to give detailed explanations. You can take a look at the preview of our login screen with the following screenshot:

Next, we need to implement the form submission handler for login functionality. Before that, we need to update our plugin constructor with the following code to define another custom action for login:

```
add_action( 'wpwa_login_user', array( $this, 'login_user' ) );
```

Once the user requests `/user/login` from the browser, the controller will execute the `do_action( 'wpwa_login_user' )` function to load the login form on the frontend.

Displaying a login form

We are going to use the same function to handle both template inclusion and form submission for login, as we did earlier with registration. So, let's look at the initial code of the `login_user` function for including the template:

```
public function login_user() {

  if ( !is_user_logged_in() ) {
    include dirname(__FILE__) . '/templates/login.php';
  } else {
    wp_redirect(home_url());
  }
  exit;
}
```

First, we need to check whether a user has already logged in to the system. Based on the result, we redirect the user to the login template or home page for the moment. Once the whole system is implemented, we will be redirecting the logged-in users to their own admin area.

Now we can take a look at the implementation of the login to finalize our process. Let's take a look at the form submission handling part of the `login_user` function:

```
if ( $_POST ) {

  $errors = array();

  $username = isset ( $_POST['username'] ) ? $_POST['username'] :
  '';
  $password = isset ( $_POST['password'] ) ? $_POST['password'] :
  '';

  if ( empty( $username ) )
    array_push( $errors, 'Please enter a username.' );

  if ( empty( $password ) )
      array_push( $errors, 'Please enter password.' );

  if(count($errors) > 0){
    include dirname(__FILE__) . '/templates/login.php';
    exit;
  }

  $credentials = array();

  $credentials['user_login']      = $username;
  $credentials['user_login']      = sanitize_user(
  $credentials['user_login'] );
  $credentials['user_password']   = $password;
  $credentials['remember']        = false;

  // Rest of the code
}
```

As usual, we need to validate the post data and generate necessary errors to be shown on the frontend. Once validations are successfully completed, we assign all the form data into an array after sanitizing the values. The username and password are contained in the `credentials` array with the `user_login` and `user_password` keys. Remember that the key defines whether to remember the password or not. Since we don't have a remember checkbox in our form, it will be set to `false`. Next, we need to execute the WordPress login function in order to log the user into the system as shown in the following code:

```
$user = wp_signon( $credentials, false );
if ( is_wp_error( $user ) )
  array_push( $errors, $user->get_error_message() );
else
  wp_redirect( home_url() );
```

WordPress handles user authentication through the `wp_signon` function. We have to pass all the credentials generated in the previous code with an additional second parameter of `true` or `false` to define whether to use a secure cookie. We can set it to `false` for this example. The `wp_signon` function will return an object of the `WP_User` or `WP_Error` class based on the result. Once a user is successfully authenticated, redirection will be made to the home page of the site. Now we should have the ability to authenticate users from the login form on the frontend.

Do you think we implemented the process properly? Take a moment to think carefully about our requirements and try to figure out what we have missed.

Actually, we didn't check the activation status on login. Therefore, any user will be able to log in to the system without activating their account. Now let's fix this issue by intercepting the authentication process with another built-in action named `authenticate`, as shown in the following code:

```
public function authenticate_user( $user, $username, $password ) {
  if ( !in_array( 'administrator', (array) $user->roles ) ) {
    $active_status = '';
    $active_status = get_user_meta( $user->data->ID,
    'activation_status', true );

    if ( 'inactive' == $active_status ) {
      $user = new WP_Error( 'denied',
      __('<strong>ERROR</strong>: Please activate your account.'
      ) );
    }
  }
  return $user;
}
```

This function will be called in the authentication action by passing the `user` object, `username`, and `password` as default parameters. All the user types of our application need to be activated except for the administrator accounts. Therefore, we check the roles of the authenticated user to figure out whether he/she is the admin. Then, we can check the activation status of other user types before authenticating. In case the authenticated user is in inactive status, we can return the `WP_Error` object and prevent authentication from being successful.

Last but not least, we have to include the authenticate action in the controller as shown in the following code to make it work:

```
add_filter( 'authenticate', array( $this, 'authenticate_user' ),
30, 3 );
```

Now, we have a simple and useful user registration and login system ready to be implemented on the frontend of web applications. Make sure to check login and registration-related plugins from the official repository to gain knowledge of complex requirements in real-world scenarios.

Time to practice

In this chapter, we implemented a simple registration and login functionality from the frontend. There are plenty of other tasks to be completed before we have a complete user creation and authentication system. So, I would recommend you to try out the following tasks in order to be comfortable with implementing such functionalities for web applications:

- Create frontend functionality for lost passwords
- Block the default WordPress login page and redirect to our custom page
- Include extra fields in the registration form

Make sure to try out these exercises and validate your answers against the implementations provided in the website for this book.

Summary

In this chapter, we explored the basics of user roles and capabilities related to web application development. We were able to choose the user roles for our application considering the various possibilities provided by WordPress.

Next, we learned how to create custom routes in order to achieve an MVC-like process using the front controller and custom templates system.

Finally, we looked at how we can customize the built-in registration and login process on the frontend to cater to the advanced requirements in web application development.

By now, you should be capable of doing the following tasks:

- Define user roles and capabilities to match your application
- Create custom routers for common modules
- Implement custom controllers with custom template systems
- Customize exiting user registration and authentication process

In the next chapter, we are going to look at how we can adapt the existing database of WordPress into web applications while planning the database for a portfolio management application. Stay tuned for another exciting chapter.

3
Planning and Customizing the Core Database

In general, a database acts as the primary location for keeping your web application data accessible from frontend interfaces or any third-party systems. Planning and designing the database should be one of the highest priority tasks in the initial stages of a project.

As developers, we have the chance to design the database from scratch in many web applications. WordPress comes with a prestructured database and hence the task of planning the table structure and adapting to existing tables becomes much more complex than everyone thinks. Throughout this chapter, we are going to focus on the basics of planning and accessing the database for web applications. This chapter is important for the rest of the book and may seem more theoretical compared to other chapters.

In this chapter, we will be covering the following topics:

- Understanding the WordPress database
- Exploring the role of existing tables
- Adapting existing tables into web applications
- Extending a database with custom tables
- Planning the portfolio application tables
- Querying the database
- Limitations and considerations

Understanding the WordPress database

Typical full stack web development frameworks don't come with a preplanned database structure. Instead, these frameworks focus on the core foundation of an application while allowing the developers to focus on application-specific features. On the other hand, WordPress provides a preplanned database structure with a fixed set of tables. WordPress is built to function as a content management system and hence it can be classified as a product rather than a pure development framework. A WordPress core database is designed to power the generic functionalities of a CMS. So, it's our responsibility to use our skills to make it work as an application development framework.

Our WordPress database is intended to work with MySQL and hence we need to have a MySQL database set up before installing WordPress. On successful installation, WordPress will create 11 database tables to cater to core functionality with the default MySQL table engine.

MyISAM was used as the default MySQL table engine prior to Version 5.5 and has been changed to InnoDB from Version 5.5 onwards.

WordPress core features will always be limited to these 11 tables, and it's quite surprising to see the flexibility of building a wide range of applications with such a limited number of tables. Both WordPress and framework developers need to have a thorough understanding of the existing tables in order to associate them in web applications.

Exploring the role of existing tables

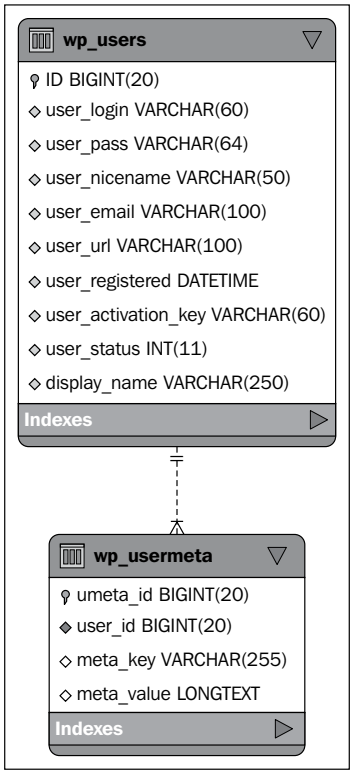
Assuming that most of you are existing WordPress developers, you will have a solid understanding of the existing database table structure. However, I suggest you to continue with this section, as web applications can have a different perspective in using these tables. Based on the functionality, we are going to categorize the existing tables into four sections, named:

- User-related tables
- Post-related tables
- Term-related tables
- Other tables

Let's look at how each table fits into these categories and their role in web applications.

User-related tables

This section consists of two tables for maintaining the user-related information of your application. Let's take a look at the relationship between user-related tables before moving on to explanations. The following diagram will give a brief idea on this:



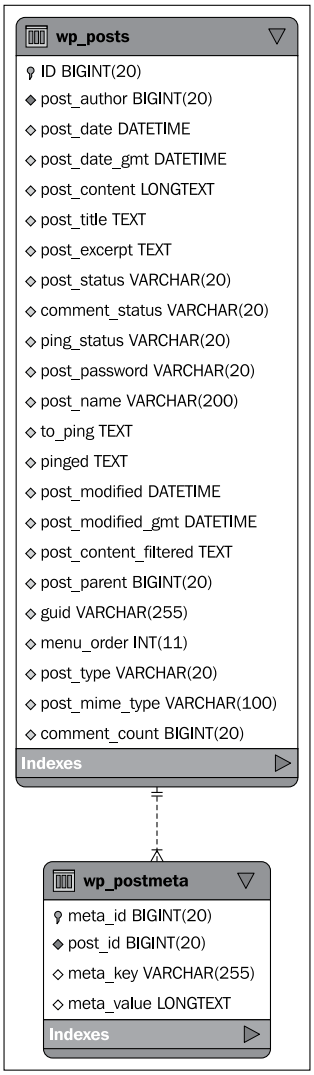
The following is the explanation of these user-related tables:

- `wp_users`: All the registered users will be stored in this table with their basic details, such as name, e-mail, username, and password.
- `wp_usermeta`: This table is used to store additional information about the users as key-value pairs. User roles and capabilities can be considered as the most important user-specific data of this table. Also, we have the freedom of adding any user-related information as new key-value pairs.

Throughout this chapter, we'll be referring to the WordPress tables with its default prefix `wp_`. You can change the prefix through the installation process or by manually changing the `config` file.

Post-related tables

This section consists of two tables for keeping the website's post- and page-related information. Let's take a look at the relationship between post-related tables before moving on to the explanations. The following diagram will give a brief idea on this:



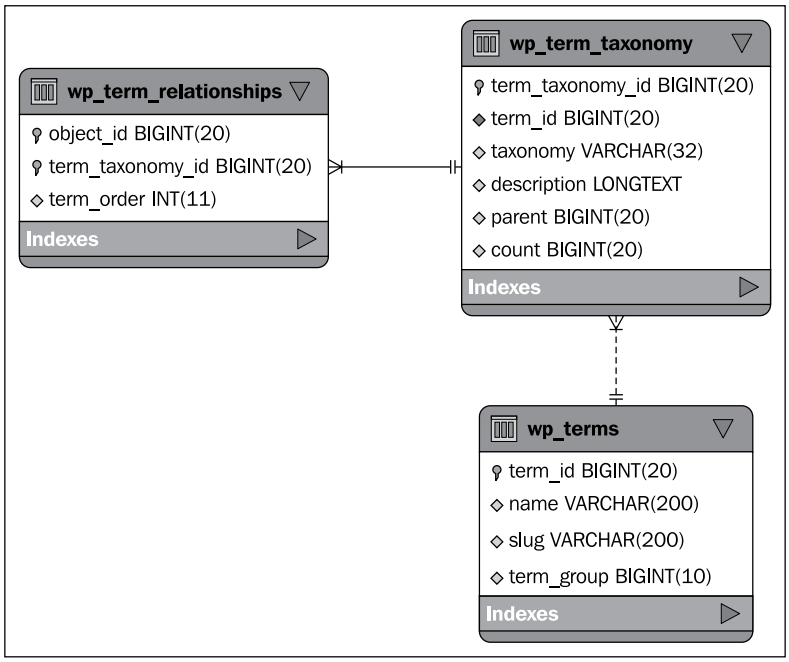
The following is the explanation of these post-related tables:

- `wp_posts`: This table is used to keep all the posts and pages of your website with their details, such as post name, author, content, status, and post type.

- `wp_postmeta`: This table is used to keep all the additional details for each post as key-value pairs. By default, it will contain the details, such as page template, attachments, and edit locks. Also, we can store any post-related information as new key-value pairs.

Term-related tables

WordPress terms can be simply described as categories and tags. This section consists of three tables for post-, category-, and tag-related information. Let's take a look at the relationship among term-related tables:



The following is the explanation of these term-related tables:

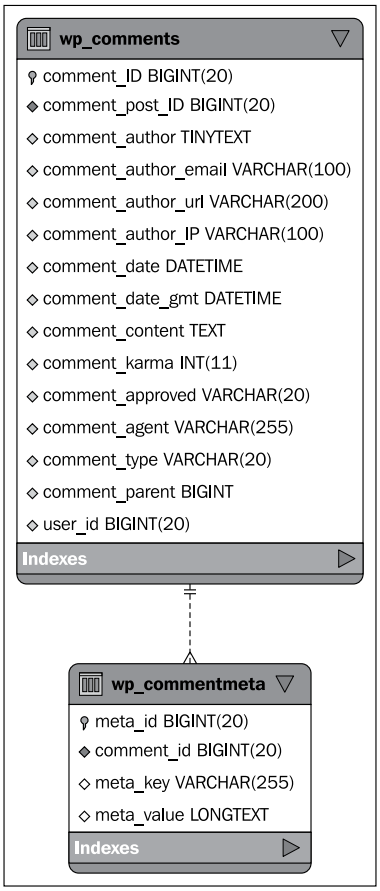
- `wp_terms`: This table contains the master data for all the new categories and tags, including custom taxonomies.
- `wp_term_taxonomy`: This table is used to define the type of terms and the number of posts or pages available for each term. Basically, all the terms will be categorized as category, post-tags, or any other custom terms created through plugins.
- `wp_term_relationships`: This table is used to associate all the terms with their respective posts.

Other tables

I have categorized all the remaining four tables in this section as they play a less important or independent role in web applications:

- `wp_comments`: This table is used to keep the user's feedback for posts and pages. Comment-specific details such as author, e-mail, content, and status are saved in this table.
- `wp_commentmeta`: This table is used to keep additional details about each comment. By default, this table will not contain much data as we are not associating advanced comment types in typical situations.

The following diagram previews the relationship between the comment-related tables:



The following is the explanation of these comment-related tables:

- `wp_links`: This table is used to keep the necessary internal or external links. This feature is rarely used in content management systems.
- `wp_options`: This table acts as the one and only independent table in the database. In general, this is used to save the application-specific settings that don't change often.

You can take a look at the complete entity-relationship diagram of WordPress at `http://codex.wordpress.org/images/9/9e/WP3.0-ERD.png`.

Now, you should have a clear idea of the role of existing tables and the reasons for their existence in the CMS perspective. Most importantly, our goal is to figure out how these tables work in advanced web applications, and the next section will completely focus on the web application perspective.

Adapting existing tables into web applications

Unlike content management systems, web applications have the possibility of scaling infinitely as they become popular and stable. Such systems can contain hundreds of database tables to cater to various aspects. Here, we are trying to build such applications using this popular CMS framework. Therefore, we need to figure out the features that we can build using the existing tables and the features that should need their own table structures.

We should try to maximize the use of existing tables in every possible scenario to get the most out of WordPress. Built-in database access functions are optimized to work directly with the existing tables, allowing us to minimize the time for implementation. On the other hand, we need to write custom queries from scratch to work with newly created tables. Let's find out the possible ways of adapting the existing tables using the four categories we discussed in the previous section.

User-related tables

The role of user tables in web applications is similar to that in normal CMS. Therefore, we don't have to worry about changing the default functionality. Any other user-related functionalities should be associated with the `wp_usermeta` table. Let's recall the user activation feature we implemented in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*. We had an additional requirement for activating users before login. We made use of the new `wp_usermeta` field named `wpwa_activate_status` to build this functionality. Now, open your database explorer, and take a look at the fields of the `wp_users` table. You will find a column named `user_activation_key` with an empty value. This field could have been easily used for the activation functionality. Table columns such as `user_activation_key` and `user_status` are used by WordPress for providing the core functionality. There is every chance that other plugin developers are using these fields with a different meaning and hence creating the possibility of loss of data and conflicts.

It's a good rule of thumb to use metatables or custom tables for advanced functionalities with your own unique keys by using a prefix, instead of relying on existing columns of core tables.

Therefore, we choose the `wp_usermeta` table to keep the activation status of all the users. Other plugin developers can also implement similar functionalities with the unique keys inside the `wp_usermeta` table. In short, the `wp_usermata` table can be used effectively to create advanced user-related functionalities in web applications as long as it doesn't involve a one-to-many relationship. It's possible to use multiple meta fields with the same name. However, most developers will prefer the use of custom tables for features that require multiple data rows to be associated with the single user, allowing additional flexibility in data filtering.

Post-related tables

Usually, the `wp_posts` and `wp_postmeta` tables will act as the main data storage tables for any web application. With the introduction of custom posts, we have the capability of matching most of our application data with these two tables. In web applications, we can go beyond normal posts by creating various custom post types. Let's take a look at a few practical scenarios for identifying the role of the `wp_posts` and `wp_postmeta` tables.

Scenario 1 – an online shopping cart

Assume that we are building an online shopping cart application to sell books. In this scenario, books can be matched to a custom post type to be saved in the `wp_posts` table. We can match the post title as book title, post content as book description, and post type as book. Then, all the other book-related information such as price, author, pages, and dimensions can be stored in the `wp_postmeta` table with the associated book from the `wp_posts` table.

Scenario 2 – hotel reservation system

In this scenario, we need to provide the ability to book hotel rooms through the system. We can use a custom post type called rooms to keep the details of various types of rooms inside the `wp_posts` table. All the additional room-specific data, such as room type, check in/out dates, and number of people, can be created using additional fields in the `wp_postmeta` table.

Scenario 3 – project management application

Let's consider a much more advanced scenario in creating relationships among post types. Assume that we have been assigned to build a project management application with WordPress. We can match projects as a custom post type. Project-specific details such as project manager, duration, and cost will be stored in the `wp_postmeta` table. It's not ideal to use the `wp_postmeta` table to store project tasks since each project contains multiple tasks, and a single project task can contain its own attributes and values. Therefore, we create another custom post type to store project tasks and all the task-related data is stored inside the `wp_postmeta` table. Finally, we can associate projects with the tasks using taxonomies or a separate custom table.

Up until now, we discussed three completely different scenarios in real-world applications, and we were able to match all the requirements with the custom post types. Now, you should be able to understand the importance of these two tables in web application development. In *Chapter 4*, *The Building Blocks of Web Applications*, we will be continuing our exploration on the custom post types and the usage of these two tables.

Term-related tables

Even though they're not as important as posts, terms will play a vital part in supporting post functionalities. Let's see how we can effectively use the terms in the previous three scenarios:

- **Scenario 1**: In the book store, we can use terms to store book categories or book types, such as e-books, Kindle editions, and printed books

- **Scenario 2**: In the hotel reservation system, we can use terms to select services and facilities required for rooms
- **Scenario 3**: In the project management system, we can associate terms for defining the complexity of the given task

It's important to keep in mind that multiple terms can be associated with a single post. So, it's not the wisest thing to use terms for a feature such as project status.

Other tables

In this section, we are going to discuss the practical usage of the `wp_comments`, `wp_comment_meta`, and `wp_options` tables. The `wp_links` table is skipped on purpose as we don't generally require it in web application development.

Link manager is hidden by default for new WordPress installations since Version 3.5, proving that links are not considered a major aspect in WordPress.

Comments might not indicate a significant value with its default usage. However, we can certainly think of many ways of incorporating comments into actual web applications. In the previous section, we talked about custom post types. So what about custom comment types? Definitely, we can implement custom comment types in web applications. The only difference is that custom post types are defined in the posts table, while custom comment types will have to be handled manually as they're not supported yet in WordPress.

Let's recall the example in *Chapter 1*, *WordPress As a Web Application Framework*, where we created the question-answer interface using posts and comments. Answers were considered as the custom comment type. Similarly, we can match things such as bids in auctions, reviews in books, and ratings for movies as custom comment types to be stored in the `wp_comment_meta` table. Since the column named `comment_type` is not available, we have to use a meta key named `wpwa_comment_type` to filter different comments from each other.

Finally, we are going to take a look at the `wp_options` table for system-wide configurations. By default, this table is populated with the settings to run the website. WordPress theme settings will also be stored in this table. In web applications, we will definitely have a considerable number of plugins. So, we can use this table to store the settings of all our plugins.

Most of the existing WordPress plugins use a single field to store all the settings as a serialized array. It's considered a good practice that increases the performance due to a limited number of table records.

Up until this point, we explored the role of the existing tables and how we can adapt them in real-world web applications. Complex web applications will always come up with requirements for pushing the boundaries of these tables. In such cases, we have no option other than going with custom tables. So, we will be looking at the importance of custom tables and their usage in the next section.

Extending a database with custom tables

Default WordPress databases can be extended by any number of custom tables to suit our project requirements. The only thing we have to consider is the creation of custom tables over existing ones. We can think of two major reasons for creating custom tables:

- **Difficulty of matching data with existing tables**: In the previous section, we considered real application requirements and matched the data with the existing tables. Unfortunately, it's not practical in all scenarios. Consider a system where the user purchases books from a shopping cart. We need to keep all the payment and order details for tracking purposes, and these records act as the transactions in the system. There is no way that we can find a compatible table for these kinds of requirements. Such requirements will be implemented using a collection of custom tables.
- **Increased data volume:** As I mentioned earlier, the posts table plays a major role in web applications. When it comes to large scale applications with a sizeable amount of data, it's not recommended to keep all the data types in the posts table. Assume that we are building a product catalog that has over a hundred different products. Creating a hundred custom post types to cater this scenario is almost impossible. On the other hand, the posts table would go out of control due to the large dataset. The same theory applies for the existing metatables as well. In such cases, it's wise to separate different datasets into their own tables to improve performance and keep things manageable.

Planning the portfolio application tables

As described in this book, the portfolio management system will make use of the existing tables in every possible scenario. However, it's hard to imagine even an average web application without using custom tables. So, here we are going to identify the possible custom tables for our system. You might need to refer back to the planning section of *Chapter 1*, *WordPress As a Web Application Framework*, in order to recollect the system requirements. We planned to create a functionality to allow subscribers to follow developers in the system. Let's discuss the requirements in detail to identify the potential tables.

> *Developers can build their portfolios with personal info, services, projects, articles, or any other necessary things to demonstrate their skills. Each user will have their own RSS feed containing all the activities within the system. Followers will be allowed to subscribe to multiple developers.*

This is a very simple and practical scenario for identifying the use of custom tables. We can easily scale this up so that it is compatible with the complex systems. Developers are stored as users of the system. Therefore, we only have the choice of the `wp_usermeta` table for additional features. It's highly impractical to keep user activities in the `wp_usermeta` table. So, we need to create our first custom table named `user_activities` to implement this feature.

Types of tables in web applications

Database tables of web applications can be roughly categorized into three sections, namely:

- **Master tables**: These tables contain predefined or configuration data for the application that rarely gets changed. The wp_options table can be considered the perfect example for this type of table in the WordPress context.
- **Application data tables**: These tables contain the highly dynamic core application data. wp_users can be considered good examples for these types of tables in the WordPress context.
- **Transaction tables**: These tables contain the highest volume of data in any application. Records in these tables rarely get changed, but new records will be added at a faster rate. It's difficult to find good examples for these types of tables in the WordPress context.

Based on the categories, we can clearly see that the `user_activities` table falls into the transaction table category. Next, we need to allow the followers to subscribe to developers. So, we need another transaction type table named `subscribed_developers`. We can assume that most of the transaction type tables will need their own custom tables. For now, we are going to stick with these two tables and additional custom tables will be added in later chapters when needed.

Creating custom tables

In typical scenarios, we create the database tables manually before moving into the implementation. With the WordPress plugin-based architecture, it's certain that we might need to create custom tables using plugins in the later stages of the projects. Creating custom tables through plugins involves certain predefined procedures recommended by WordPress. Since table creation is a one-time task, we can implement the process on plugin activation. So, let's get started by creating a new folder named `wpwa_database_manager` inside the `/wp-content/plugins` folder.

Then, create a PHP file inside the folder and save it as `wpwa-database-manager.php`. Now it's time to add the plugin definition as shown in the following code:

```
<?php
class WPWA_Database_Manager {

  public function __construct(){
    register_activation_hook( __FILE__, array
    ( $this, 'create_custom_tables' ) );
  }

  public function create_custom_tables() {
    // Creating Database Tables
  }
}
```

Once the activation hook is defined inside the plugin constructor, we can implement the `create_custom_tables` function to create the necessary tables for our application. Basically, we can execute direct SQL queries using the `$wpdb->query` function to create all the tables we need. WordPress recommends using a built-in function called `dbDelta` for creating custom tables. This function is located in a file outside the default process and hence we need to load it manually within our plugins. Let's create two tables for our application using the `dbDelta` function.

```
public function create_custom_tables() {
  global $wpdb;
```

```
    $table_name = $wpdb->prefix.user_activities;

    require_once( ABSPATH . 'wp-admin/includes/upgrade.php' );

    $sql = "CREATE TABLE $table_name (
      id mediumint(9) NOT NULL AUTO_INCREMENT,
      time datetime DEFAULT '0000-00-00 00:00:00' NOT NULL,
      user_id mediumint(9) NOT NULL,
      activity text NOT NULL,
      url VARCHAR(255) DEFAULT '' NOT NULL,
      UNIQUE KEY id (id)
    );";

    dbDelta($sql);

    // subscribed_developers will be created in a similar manner
  }
```

Initially, we have to include the `upgrade.php` file to make use of the `dbDelta` function. The next most important thing is to use the prefix for database tables. By default, WordPress creates a prefix named `wp_` for all the tables. It's important to use the existing prefix to keep consistency and avoid issues in multisite scenarios. Finally, you can define your table creation query and use the `dBDelta` function to implement it on the database.

Check out the guidelines at `http://codex.wordpress.org/Creating_Tables_with_Plugins` for creating the table creation query as the `dbDelta` function can be tricky in certain scenarios.

We created the custom tables using the `dbDelta` function inside plugin activation. WordPress recommends the `dbDelta` function over direct SQL queries for table creation since it examines the current table structure, compares it with the desired table structure, and makes the necessary modifications without breaking the existing database tables. Apart from table creation, we can execute quite a few database-related tasks on plugin activation, such as altering tables, populating initial data to custom tables, and upgrading the plugin tables.

We looked at the necessity of custom tables for web applications. Even though custom tables offer you more flexibility within WordPress, there will be a considerable amount of limitations, as listed in the following section:

- They are difficult to manage in WordPress upgrades.
- WordPress default backups will not include custom tables.

- No built-in functions for accessing databases. All the queries, filtering, and validation need to be done from scratch using the existing `$wpdb` variable.
- User interfaces for displaying the data of these tables need to be created from scratch.

Therefore, you should avoid creating custom tables in all possible scenarios unless you have a distinct advantage from the perspective of your application.

The WordPress PODS framework works very well in managing custom post types with custom tables. You can have a look at the source code at `http://wordpress.org/plugins/pods/` for learning the use of custom tables.

Detailed exploration about the PODS framework will be provided in the next chapter, *Chapter 4*, *The Building Blocks of Web Applications*.

Querying the database

As with most frameworks, WordPress provides a built-in interface for interacting with the database. Most of the database operations will be handled by the `wpdb` class located inside the `wp-includes` directory. The `wpdb` class will be available inside your plugins and themes as a global variable and provides access to all the tables inside the WordPress database including custom tables.

Using the `wpdb` class for CRUD operations is straightforward with its built-in methods. The complete guide for using the `wpdb` class can be found at `http://codex.wordpress.org/Class_Reference/wpdb`.

Querying the existing tables

WordPress provides well-optimized built-in methods for accessing the existing database tables. Therefore, accessing these tables becomes straightforward. Let's see how basic **CRUD** (**Create, Read, Update, and Delete**) operations are executed on existing tables.

Inserting records

All the existing tables contain a prebuilt insert method for creating new records. The following list illustrates a few of the built-in insert functions:

- `wp_insert_post`: This function creates a new post or page in the `wp_posts` table
- `add_option`: This function creates a new option on the `wp_options` table, if it doesn't already exist
- `wp_insert_comment`: This function creates a new comment in the `wp_comments` table

Updating records

All the existing tables contain a prebuilt update method for updating existing records. The following list illustrates a few of the built-in update functions:

- `update_post_meta`: This function creates or updates additional details about posts in the `wp_postmeta` table
- `wp_update_term`: This function updates the existing terms in the `wp_terms` table
- `update_user_meta`: This function updates the user's meta details in the `wp_usermeta` table based on the user ID

Deleting records

Similar methods are available for deleting records in each of the existing tables.

Selecting records

As usual, there is a set of built-in functions for selecting records from the existing tables. The following list contains few of the data selecting functions:

- `get_posts`: This function retrieves the posts as an array from the `wp_posts` table based on the passed arguments. Also, we can use the `WP_Query` class with the necessary arguments to get the post list from the OOP method.
- `get_option`: This function retrieves the option value of the given key from the `wp_options` table.
- `get_users`: This function retrieves the list of users as an array from the `wp_user` table.

Most of the database operations on existing tables can be executed using these built-in functions. So, use of the `$wpdb` class is not necessary in most occasions unless queries become complex and hard to handle using direct functions.

Querying the custom tables

Basically, there are no built-in methods for accessing custom tables using direct functions. So, it's a must to use the `wpdb` class for handling custom tables. Let's take a look at some of the functions provided by the `wpdb` class:

- `$wpdb->get_results( "select query" )`: This function can be used to select a set of records from any database table.
- `$wpdb->query('query')`: This function can be used to execute any custom query. This is typically used for update and delete statements instead of select statements as it only provides the affected rows' count as the result.
- `$wpdb->get_row('query')`: This function can be used to retrieve a single row from the database as an object, an associative array, or as a numerically indexed array.

The complete list of the `wpdb` class functions can be accessed at `http://codex.wordpress.org/Class_Reference/wpdb`. When executing these functions, we have to make sure that we include the necessary filtering and validations as these are not built to directly work with the existing tables. For example, consider the following query for proper usage of these functions with the necessary filtering:

```
$wpdb->query(
  $wpdb->prepare("SELECT FROM $wpdb->postmeta
  WHERE post_id = %d AND meta_key = %s",
  1, 'book_title'
  )
);
```

Here, we are filtering the user input values through the `prepare` function for illegal operations and illegal characters. Similarly, you have to use functions such as `escape` and `escape_by_ref` for securing direct SQL queries.

Data validation is an important aspect of keeping the consistency of the database. WordPress offers the `prepare` function for formatting the SQL queries from possible threats. Usually, developers use the `prepare` function with direct queries including variables instead of using placeholders and value parameters. It's a must to use placeholders and value parameters to get the intended outcome of the `prepare` function. Therefore, WordPress Version 3.5 and onwards enforce a minimum of two arguments to prevent developers from misusing the `prepare` function.

Working with posts

WordPress posts act as the main module in web application development as well as content management systems. Therefore, WordPress comes with a separate class named `WP_Query` for interacting with the posts and pages. You can look into more details about the use of `WP_Query` at `http://codex.wordpress.org/Class_Reference/WP_Query`.

Up until now, we had a look at procedural database access functions using global objects. Web application developers are much more familiar with object-oriented coding. The `WP_Query` class is a good choice for such developers in querying the database. Let's find out the default usage of `WP_Query` using the following code:

```
$args = array(
  'post_type' => 'projects',
  'meta_query' => array(
    array(
    'language' => '',
    'value' => 'PHP'
    )
  )
);

$query = new WP_Query($args);
```

First, we need to add all the filtering conditions to an array. The `WP_Query` class allows us to include conditions on multiple tables, such as categories, tags, and postmeta. This technique allows us to create highly complex queries without worrying about the SQL code. The advantage of `WP_Query` comes with its ability to create sub classes to cater to project-specific requirements. In the next section, we are going learn how to extend the `WP_Query` class to create custom database access interfaces.

Extending WP_Query for applications

The default `WP_Query` class works similarly for all types of custom post types. In web applications, we can have different custom post types with different meanings. For example, developers can create services inside our portfolio application. Each service will have a price and process associated with it. There is no point retrieving these services without those additional details. Now, let's look at the default way of retrieving services with `WP_Query` using the following code:

```
$args = array(
  'post_type' => 'services',
  'meta_query' => array(
    array(
```

```
      'key' => 'price'
      ),
      array(
        'key' => 'process'
      )
    )
);

$query = new WP_Query( $args );
```

This query works perfectly in retrieving services from the database. However, each time we have to pass the price and process keys in order to join them while retrieving services. Since this is a services-specific requirement, we can create a custom class to extend `WP_Query` and avoid repetitive argument passing as it's common to all the services-related queries. Let's implement the extended `WP_Query` class as follows:

```
class WPWA_Services_Query extends WP_Query {

  function __construct( $args = array() ) {

  $args = wp_parse_args( $args, array(
    'post_type' => 'services',
    'meta_query' => array(
      array(
        'key' => 'price'
      ),
      array(
        'key' => 'process'
      )
    )
  ) );
  parent::__construct( $args );
  }
}
```

Now, all the common conditions are abstracted inside the `WPWA_Services_Query` class. So, we don't have to pass the conditions every time we want services. The preceding example illustrates the basic form of object inheritance. Additionally, we can use post filters to combine custom tables with services. Now, we can access services using the following code without passing any arguments:

```
$query = new WPWA_Services_Query();
```

The `WP_Query` class is going to play a vital part in our portfolio application development. In the upcoming chapters, we are going to explore how it can be extended in several ways using advanced post filters provided by WordPress. Until then, you can check out the available post filters at `http://codex.wordpress.org/Plugin_API/Filter_Reference#WP_Query_Filters`.

Limitations and considerations

We have less flexibility with WordPress built-in databases compared with designing a database from scratch. Limitations and features unique to WordPress need to be understood clearly to make full use of the framework and avoid potential bottlenecks. Let's find out some of the WordPress-specific features and their usage in web applications.

Transaction support

In advanced applications, we can have multiple database queries that need to be executed inside a single process. We have to make sure that either all the queries get executed successfully or none of them gets executed to maintain the consistency of the data. This process is known as transaction management in application development. In simple website development, we rarely get such requirements for handling transactions. As mentioned earlier, MySQL Version 5.5 and onwards use InnoDB as the table engine and hence we have the possibility of implementing transaction support. However, WordPress doesn't offer a library or any functions for handling transactions and hence all the transaction handling should be implemented manually.

Post revisions

WordPress provides an important feature for keeping revisions of your posts in the `wp_posts` table. On every update, a new revision of the post will be created in the database. If you have experience working with software versioning and revision control systems, you should probably know the importance of revisions. However, it could create unnecessary performance overheads in executing queries in large databases. In web applications, you should disable this feature or limit the revisions to a certain number, unless it provides potential benefits within your system.

How to know whether to enable or disable revisions

Ideally, you should disable this feature in all forms of web application development. Later, you can consider enabling this feature based on your application requirements.

It's important to keep in mind that we don't get revisions of the post meta fields. Therefore, importance of post revisions are restricted to the fields such as post title, content, author, and excerpts.

Let's consider a practical scenario for identifying the importance of post revisions. Assume that we have an event management system with a custom post type called events. Each event is going to be spanned across multiple days. So, you can create an event and use the post content to include the activities of the first day. Then from next day onwards, you can completely replace the content with the activity of each day and update the event. Finally, we can get all the post revisions with a link to each day for filtering the activities conducted each day. So, the decision of keeping post revisions purely depends on your requirements.

Consider disabling post revisions by placing the following code inside the `wp-config.php` file.

```
define('WP_POST_REVISIONS', false );
```

Autosaving

This is another feature that combines with post revisions. Autosaving will create a different type of post revision in predefined time intervals. In most occasions, this feature will expand the size of your database rather than providing something useful. Unfortunately, we cannot switch off autosaving without editing the core files. Therefore, we need to extend the interval of autosaving by defining a large value for the `AUTOSAVE_INTERVAL` constant inside the `wp-config.php` file.

```
define('AUTOSAVE_INTERVAL', 600 );
```

The value of the `AUTOSAVE_INTERVAL` constant needs to be configured in seconds. Here we have used 600 seconds (10 minutes) as the autosave interval.

Using metatables

WordPress table structure gives higher priority to metatables for keeping additional data as key-value pairs. Although metatables work well in most scenarios, it can become a considerable factor in situations where you need to implement complex select queries and search functionality. Searching for `n` number of fields means that you create `n` number of SQL table joins on the metatable. As the number of joins increases, your queries will get slower and slower. In such situations, it's ideal to go with custom tables instead of relying on existing tables.

We had a brief introduction to the WordPress database and the possible ways of using it in web applications. Covering all possible database design and access techniques was beyond the scope of this chapter. So, I recommend you to follow the resource section for this chapter on the official book website at `http://www.innovativephp.com/wordpress-web-applications` for more resources and tutorial updates.

Summary

Understanding the WordPress database is the key to building successful web applications. Throughout this chapter, we looked at the role of existing tables and the need for custom database tables through practical scenarios.

Database querying techniques and limitations were introduced with the necessary examples. By now, you should have a clear understanding of choosing the right type of tables for your next project. We had to go in a theoretical approach with practical scenarios to learn the basics of database design and implementation inside WordPress.

The real excitement begins in the next chapter where we start the development of our main modules in web applications using the building blocks of WordPress, so stay tuned.

4
The Building Blocks of Web Applications

The majority of WordPress powered systems are either simple websites or blogs. Adapting WordPress for building complex web applications can be a complex task for developers as beginners, as they are used to working with simple websites everyday. Understanding the process of handling web application specific functions becomes vital in such scenarios.

Managing data is one of the most important tasks in web applications. WordPress offers a concept called custom post types for modeling application data and backend interfaces. I believe this is the foundation of most web applications, hence, named this chapter, *The Building Blocks of Web Applications*.

While exploring advanced use cases of custom post type implementations, we are going to get used to popular web development techniques such as modularizing, template management, data validations, and rapid application development in a practical process.

In this chapter we will cover the following topics:

- Introduction to custom content types
- Planning custom post types for applications
- Implementing the custom post type settings
- Validating a post creation
- Integrating a template engine to WordPress
- Introduction to custom post type relationships
- The Pods framework for custom content types
- Tasks for practicing custom post types

Let's get started.

Introduction to custom content types

In WordPress terms, custom content types are referred to as custom post types. The term "custom post types" misleads some people into thinking of them as different types of normal posts. In reality, these post types can model almost anything in real web applications. These post types are stored in the normal posts table and it could well be the reason behind its conflicting naming convention.

Prior to the introduction of custom post types, we only had the ability to use normal posts with custom fields to cater to advanced requirements. The process of handling multiple post types was a complex task with those limited features. With the introduction of custom post types, we now have the ability to separate each different type to act as a model and cater to complex requirements. The demand for using these custom post types to build complex applications is increasing everyday. The features provided out of the box to cater to common tasks might be one of the reasons behind its popularity in website development.

The role of custom post types in web applications

Even the simplest of web applications will contain a considerable amount of models compared to normal websites. Therefore, organizing model data becomes one of the critical tasks in application development. Unless you want complete control over your data processing, it's preferable to make use of custom post types without developing everything from scratch.

Once a custom post type is registered, you will automatically get the ability to execute **CRUD** (**Create**, **Read**, **Update**, **Delete**) operations. Default fields enabled in the post creation and custom category types will be saved automatically upon hitting the publish button. Generally, this is all we need to build simple applications. For web applications, we may need to handle a large amount of data with various types of fields. This is where web applications differ from simple websites with the use of custom fields within meta boxes. This chapter will mainly focus on handling this custom data in different ways to suit the development of complex applications.

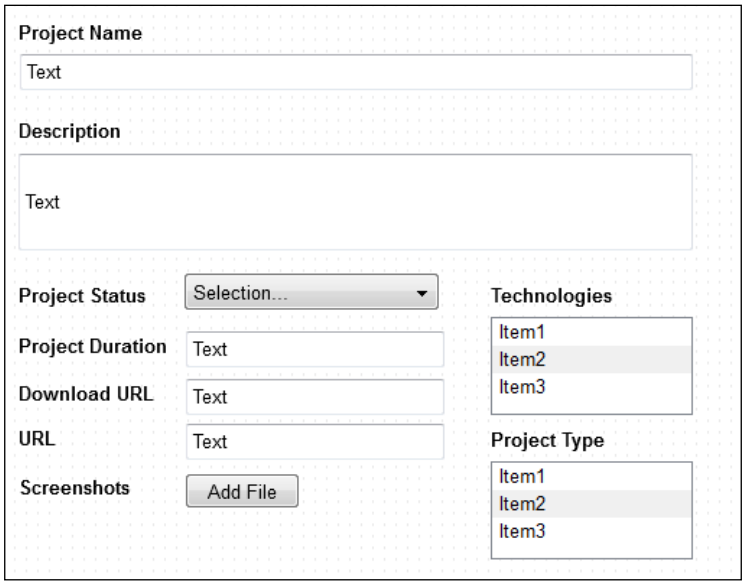
Planning custom post types for the application

After that brief introduction to custom post types and their role in web applications, we are now going to find the necessary custom post types for our portfolio application. The majority of our applications and data will be based on these custom post types. So let's look at the detailed requirements of a portfolio application.

The main objective of this application is to let developers promote their work to enhance their reputation. So, we are going to target a few important components such as the projects developed, services offered, and articles and books written. Now try to visualize the subsections of each of these components. It's obvious that we can match these components into four custom post types. The following sections illustrate the detailed subcomponents of each of these models.

Projects

Developers or designers could have a list of projects to build their portfolio. Project information can vary based on the type of project. We are going to limit the implementation of projects to some common fields to cover the different areas of custom post types. The following screenshot illustrates the fields for the project creation screen:



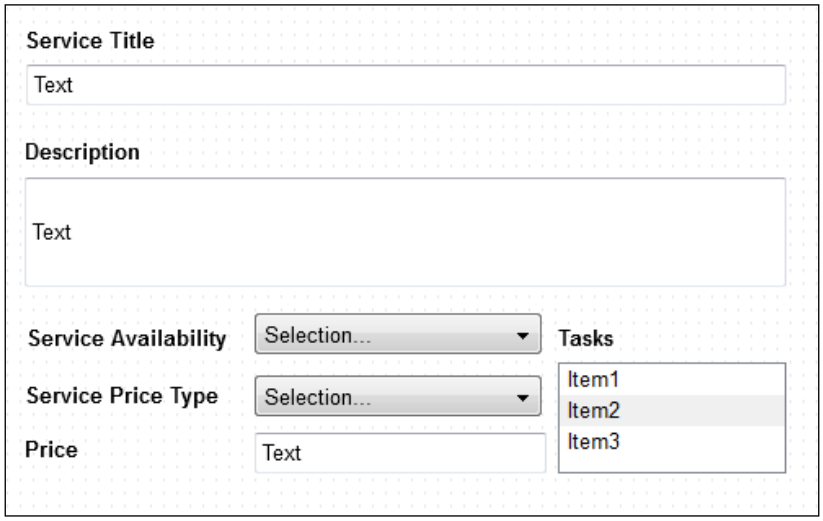
Let's have a look at the usage of the fields, shown in the previous screenshot:

- **Project Name**: This can be created as the title field of **custom post type** (**CPT**)
- **Description**: This can be created as the editor field of custom post type
- **Technologies**: This can be created as custom taxonomies
- **Project Type**: This can be created as another custom taxonomy
- **URL**: This can be created as a custom text field

- **Screenshots**: This can be created as custom fields (this will be implemented in a later chapter)
- **Project Duration**: This can be created as a custom text field
- **Download URL**: This can be created as a custom text field
- **Project Status**: This can be created as a custom drop-down field

Services

Generally, people who work for someone or a company don't offer services, but freelancers actually do have various ways of making money. Most professional freelancers have a specific page on their website to market their services. For this application, we are going to model the services using custom post types. Now, let's look at the following screenshot for the necessary data requirements and their WordPress specific matches:



Let's have a look at the usage of the fields, shown in the previous screenshot:

- **Service Title**: This can be matched as the title field of CPT
- **Description**: This can be matched as the editor field of CPT
- **Tasks**: This can be matched as custom taxonomies
- **Service Price type**: This can be matched to a custom drop-down field
- **Price**: This can be matched to a custom text field or drop-down field
- **Service Availability**: This can be matched to a custom drop-down field

Articles

This section contains people's articles, tutorials, news written for their own websites, as well as guest post submissions for other websites. An article is something that we can match exactly to the WordPress normal post type, but every website might need a blog at some point in its life cycle. So, we are going to skip normal posts and create a separate custom post type for articles. We are not going to discuss articles in detail as they contain similar fields such as `Title`, `Summary`, `URL`, `Categories`, and `Screens`.

Books

Compared to other sections, books will have less impact and less data as most developers and designers are not authors. However, writing books on your preferred technology is a great way of enhancing your reputation and building a name online. Similar to articles, books will have a generic set of fields such as `Title`, `Summary`, `URL`, `Download`, `Categories`, and `Screens`.

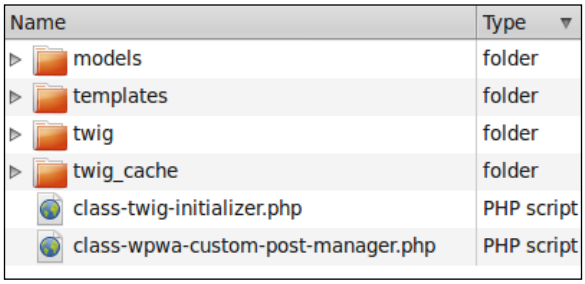
I hope you have a clear understanding about the things we are going to implement in this chapter. We are going to start by implementing these custom post types using a plugin. While implementing these models, we are also going to take a look at some advanced techniques for template management, validations, and post relationships.

Implementing custom post types for a portfolio application

As usual, we are going to create a separate plugin for the custom post type related functionalities of this chapter. First, we have to create the plugin folder and main file. So, create a folder named `wpwa-custom-post-manager` inside the `/wp-content/plugins` folder. Afterwards, you can create a main file named `class-wpwa-custom-post-manager.php` to include the plugin definitions, as shown in the following code:

```
<?php

/*
  Plugin Name: WPWA Custom Posts Manager
  Plugin URI:
  Description: Core data management using Custom Post Types for
the portfolio management application.
  Author: Rakhitha Nimesh
  Version: 1.0
  Author URI: http://www.innovativephp.com/
 */
?>
```

Once the file is created with the preceding definition code, you can refresh the plugin list and activate it to see if it's working properly. Most web applications will be larger in scale compared to the normal websites or blogs. Implementing all custom post functionalities in one file is not the most ideal or practical task to do. So, our plan here is to keep the initialization and generic configurations in the main file while separating each of the custom post types into their own class files. Before we go any further, I would like you to have a look at the folder structure of the plugin in the following screenshot:



Now let's go through each of the files and folders to identify their role:

- `class-wpwa-custom-post-manager.php`: This includes the main plugin initialization and configuration code
- `class-twig_initializer.php`: This includes the template engine initialization and configuration code
- `models`: This folder contains all of the custom post type specific classes
- `templates`: This folder contains all of the HTML templates required for the plugin
- `twig`: This folder contains the library files for the Twig template engine
- `twig_cache`: This folder is for the created cached versions of templates

All of the custom post type specific classes located inside the `models` folder need to be included into the main plugin file prior to their usage. WordPress itself handles most of the files in a procedural way. Hence, some WordPress developers prefer the inclusion of files through a set of `require` or `include` statements. As applications grow larger, including each and every file in a manual process can be a tedious task. Most experienced web developers will look for an autoloading concept for such projects. So, we are going to implement an autoloader for our post type classes inside the `models` folder.

Let me explain how the PHP autoload works before moving onto the real implementation. Consider the following code snippet:

```
spl_autoload_register('wpwa_autoloader');
function wpwa_autoloader($class_name){
  include_once $class_name;
}
```

PHP provides a function called `spl_autoload_register` to register a function to implement the autoloading process. Whenever a class is instantiated, the autoloader function will be called by passing the class name as the parameter. We can use the class name to include the files with the same name as the class name. But there can be two major problems with the default technique:

- We cannot have classes inside subfolders
- This function is going to load each and every class inside the application, even the classes we don't want to include

Basically, this means we have to look for an alternate autoloader implementation to suit our plugins. We are going to use a predefined format for our model classes to solve this issue. All of the model classes will be named `WPWA_Model_{post_type_name}`. For example, the projects class will be named `WPWA_Model_Project` while the filename will be `class-wpwa-model-project.php`, so that we can look for the project file inside the `models` folder. Having the solution in mind, we are going to implement the autoloader function illustrated in the following code:

```
spl_autoload_register('wpwa_autoloader');
function wpwa_autoloader($class_name) {
  $class_components = explode("_", $class_name);
  if (isset($class_components[0]) && $class_components[0] ==
  "WPWA" &&
    isset($class_components[1])) {

      $class_directory = $class_components[1];
      unset($class_components[0], $class_components[1]);

      $file_name = implode("_", $class_components);
      $base_path = plugin_dir_path(__FILE__);
      switch ($class_directory) {
        case 'Model':

        $file_path = $base_path . "models/class-wpwa-model-
".lcfirst( $file_name ) . '.php';
        if (file_exists($file_path) && is_readable($file_path)) {
          include $file_path;
```

```
            }
          break;
        }
      }
    }
```

Since we have opted to go with predefined class names, the initial task is to split the class name into subcomponents using the explode function. Once the class name is exploded into parts, we will have `WPWA` as the first component and `Model` as the second component. The second component is assigned to the `$class_directory` variable to be used as the folder name. Then, we need to rebuild the filename of the class. So, we unset the first two components and rejoin the remaining components using the `implode` function.

Here, we used a manual process for filtering the class names and file paths. We can simplify the process for advanced applications using a regular expression check with the `preg_match` function.

Then, we define the base path to the WordPress plugin directory in order to get the path to the class file. Next, we come to the most important part where we switch `$class_directory` to find the right folder. Here, we have used `Model` in the switch statement since our folder is named `models`. For each and every new folder, you can add a new case to the switch statement.

Afterwards, we define the path to the class file by combining the plugin base path with the Models directory and filename. Finally, we include the file into the plugin by checking the existence of the file and the necessary permissions. With this technique, we can autoload all the classes inside the `models` folder without manually loading them using `require` or `include`. Now, we can move on to the implementation of the custom post type manager.

Implementing the custom post type settings

As planned, we need to implement configurations and general functions in the main class of our plugin. Let's create a class called `WPWA_Custom_Post_Manager` in `class-wpwa-custom-post-manager.php` and initialize the object as usual, using the following code:

```
class WPWA_Custom_Post_Manager {

  private $base_path;
  private $template_parser;
```

```
  private $projects;

  public function __construct() {
    // Initialization
  }
}

$custom_post_manager = new WPWA_Custom_Post_Manager();
```

You should be familiar with this plugin initialization technique as we used it in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*. Apart from the basic initialization, we have used few instance variables to keep the data across all the functions of the class. The following section explains the role of each of these variables:

- `base_path`: This holds the path to the plugin folder from your document root
- `template_parser`: We are going to handle templates using a template engine, so this variable will hold an instance of the template system
- `projects`: This keeps an object of the Projects custom post type to be used across many functions

Also, you will have to create instance variables for all of your custom post types. Once the main class is instantiated, we have to define our custom post types inside the constructor. Remember that we planned to separate each custom post type according to its own class. Therefore, all of the post type specific implementations will be inside those classes, meaning that the responsibility of the constructor will be limited to the instantiation of those classes. Let's see how the main file constructer looks:

```
class WPWA_Custom_Post_Manager {
  // Instance variables
  public function __construct() {
    $this->base_path = plugin_dir_path(__FILE__);
    $this->projects = new WPWA_Model_Project();
  }
}
```

We start the implementation by assigning the plugin directory path to our instance variable using the WordPress `plugin_dir_path` function. Next, we need to initialize all the custom post type classes. In this chapter, we are going to look at the detailed implementation of the `Project` class. Other custom post types are similar, hence, you can find them inside the source code folder. We have initialized the `Project` class and assigned it to the instance variable. Now that we have set up everything required for custom post implementation, we can move on to the implementation of those classes.

Creating a projects class

We are going to choose the `Project` class as it's the most complex of the four custom post types. So, create a file called `class-wpwa-model-project.php` inside the `models` folder and define a blank class called `WPWA_Model_Project`, as shown in the following code:

```
class WPWA_Model_Project {
  private $post_type;
  private $template_parser;

  public function __construct() {
    // Initialization code goes here
  }
}
```

Here also, we have two instance variables for keeping the custom post type name and template engine object, which will be discussed later. This class constructor is responsible for handling all the project related function initializations and definitions. The first task is to register a custom post type for projects. Let's modify the constructor to add the necessary actions for the post type creation:

```
class WPWA_Model_Project {
  // Instance variables
  public function __construct() {
    $this->post_type = "wpwa_project";
    add_action('init', array($this, 'create_projects_post_type'));
  }
}
```

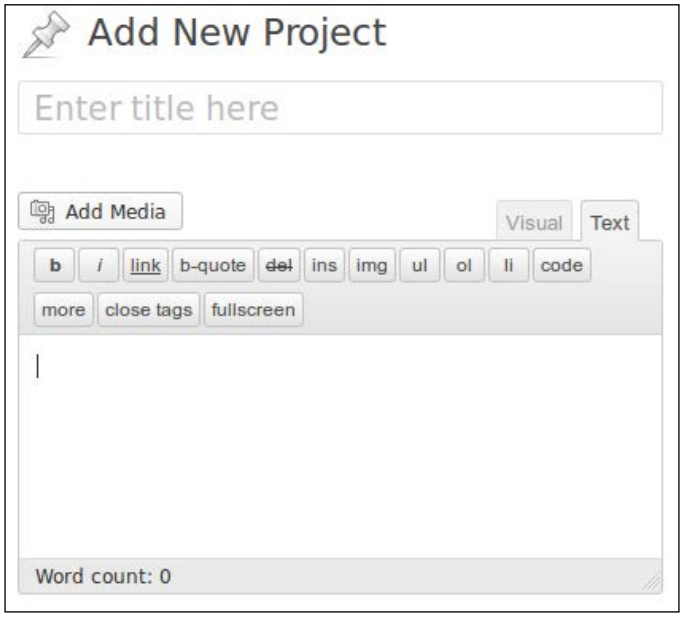
First, we assign the name of the post type to the instance variable to use it for registering the post type. In many existing plugins, you will find hardcoded names for the `register_post_type` function.

It's a good practice to use an instance variable or a global variable to store the custom post type name and use the variables across all the occurrences of the custom post type name. This will enable you to change the custom post type name anytime with minimum effort without breaking the code.

As usual, we use the WordPress `init` action to call the `create_projects_post_type` function within the same class for registering custom post type for projects. Here is the implementation of the `create_projects_post_type` function with our familiar yet complicated custom post creation code. I am not planning to provide any detailed explanations as you are already familiar with this code:

```
public function create_projects_post_type() {

  $labels = array(
    'name' => __( 'Projects', 'wpwa' ),
    'singular_name' => __( 'Project', 'wpwa' ),
    'add_new' => __( 'Add New', 'wpwa' ),
    'add_new_item' => __( 'Add New Project', 'wpwa' ),
    'edit_item' => __( 'Edit Project', 'wpwa' ),
    'new_item' => __( 'New Project', 'wpwa' ),
    'all_items' => __( 'All Projects', 'wpwa' ),
    'view_item' => __( 'View Project', 'wpwa' ),
    'search_items' => __( 'Search Projects', 'wpwa' ),
    'not_found' => __( 'No projects found', 'wpwa' ),
    'not_found_in_trash' => __( 'No projects found in the Trash',
'wpwa' ),
    'parent_item_colon' => '',
    'menu_name' => __( 'Projects', 'wpwa' )
  );

  $args = array(
    'labels'                => $labels,
    'hierarchical'          => true,
    'description'           => 'Projects',
    'supports'              => array('title', 'editor'),
    'public'                => true,
    'show_ui'               => true,
    'show_in_menu'          => true,
    'show_in_nav_menus'     => true,
    'publicly_queryable'    => true,
    'exclude_from_search'   => false,
    'has_archive'           => true,
    'query_var'             => true,
    'can_export'            => true,
    'rewrite'               => true,
    'capability_type'       => 'post',

  );
  register_post_type($this->post_type, $args);
}
```

We have to define the common settings and all the labels for each and every custom post type. Hopefully, they will be improved in the future releases of WordPress. There is nothing substantial to explain in the preceding code other than the use of the instance variable for the post type name. Now, go to the **Permalinks** menu under the **Settings** section in the admin dashboard and save it again to refresh the rewrite rules. You should see the project's creation screen, as shown in the following screenshot:



The custom post type for a project is created to have the capability of posting. Users need to have post-specific permissions to use the projects section. Since we haven't provided permissions to developers, you can only view this screen as an admin who has the post capabilities by default.

Assigning permissions to projects

In general, the developer user role should be able to handle all of the post types created in this application. Hence, we need to provide post specific capabilities to the developer role. In *Chapter 2, Implementing Membership Roles, Permissions, and Features*, we implemented the plugin for user management and permissions. Now it's time to update the plugin to add the necessary permissions. Open the `wpwa-user-manager.php` file in the `Packt WPWA User Manager` plugin. Navigate to the `add_application_user_capabilities` function and change the existing code as follows:

```
public function add_application_user_capabilities() {
```

```
    $role = get_role('follower');
    $role->add_cap('follow_developer_activities');
    $developer = get_role("developer");
    $custom_developer_capabilities = array(
      "edit_posts",
      "edit_private_posts",
      "edit_published_posts",
      "publish_posts",
      "read",
      "delete_posts",
    );

    foreach ($custom_developer_capabilities as $capability) {
      $developer->add_cap($capability);
    }
  }
```

Here, we have added most of the post specific capabilities to the developer user role, apart from the `edit_others_posts` capability. Now, you have to deactivate the `Packt WPWA User Manager` plugin and activate it again to update the capabilities for users. Once completed, you will be able to manage projects as developers.

Now we have the necessary permissions and basic fields ready for creating the project title and description. The most important part of web applications comes with the power of custom fields and custom taxonomies. In the requirements gathering section, we planned to create custom taxonomies for project technologies and the project type. So let's get started with the implementation.

Creating custom taxonomies for technologies and types

Generally, we use taxonomies to group things that don't get changed often. Here, we are in need of two taxonomies for both technologies and types. Let's open the class `wpwa-model-project.php` file and update the `Project` class constructor to implement the actions for the taxonomy creation:

```
class WPWA_Model_Project {
  // Other Instance variables
  private $technology_taxonomy;
  private $project_type_taxonomy;

  public function __construct() {
    $this->post_type = "wpwa_project";
    $this->technology_taxonomy = "wpwa_technology";
```

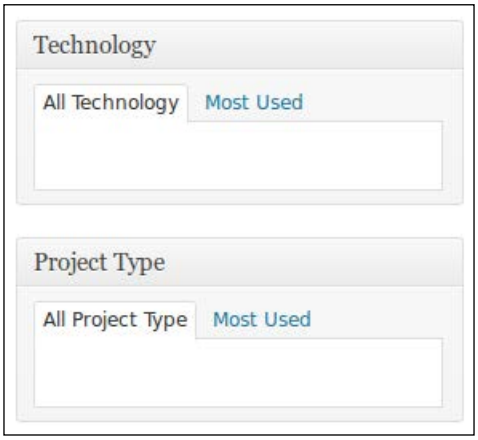
```
        $this->project_type_taxonomy = "wpwa_project_type";

        add_action('init', array($this,'create_projects_post_type'));
        add_action('init',
        array($this,'create_projects_custom_taxonomies'));
      }
    }
```

First, we need two other instance variables to hold the names of custom taxonomies to be re-used across all the functions. Initialization of these variables is handled through the constructor. Next, we define custom taxonomies on the `init` action as we did with custom post types. WordPress offers a function called `register_taxonomy` for creating taxonomies. Similar to custom post types, we have to define each and every label and the default options. Here is the implementation of the `create_projects_custom_taxonomies` function:

```
  public function create_projects_custom_taxonomies() {

    register_taxonomy(
      $this->technology_taxonomy,
      $this->post_type,
      array(
        'labels' => array(
          'name' => __( 'Technology', 'wpwa' ),
          'singular_name' => __( 'Technology', 'wpwa' ),
          'search_items'=> __( 'Search Technology', 'wpwa' ),
          'all_items'  => __( 'All Technology', 'wpwa' ),
          'parent_item'=> __( 'Parent Technology', 'wpwa' ),
          'parent_item_colon' => __( 'Parent Technology:', 'wpwa' ),
          'edit_item' => __( 'Edit Technology', 'wpwa' ),
          'update_item'=> __( 'Update Technology', 'wpwa' ),
          'add_new_item' => __( 'Add New Technology', 'wpwa' ),
          'new_item_name' => __( 'New Technology Name', 'wpwa' ),
          'menu_name' => __( 'Technology', 'wpwa' ),
        ),
        'hierarchical' => true
      )
    );

    register_taxonomy(
      $this->project_type_taxonomy,
      $this->post_type,
      array(
        'labels' => array(
          'name' => __( 'Project Type', 'wpwa' ),
```

```
            'singular_name' => __( 'Project Type', 'wpwa' ),
            'search_items' => __( 'Search Project Type', 'wpwa' ),
            'all_items' => __( 'All Project Type', 'wpwa' ),
            'parent_item' => __( 'Parent Project Type', 'wpwa' ),
            'parent_item_colon' => __( 'Parent Project Type:', 'wpwa'
  ),
            'edit_item' => __( 'Edit Project Type', 'wpwa' ),
            'update_item' => __( 'Update Project Type', 'wpwa' ),
            'add_new_item'=> __( 'Add New Project Type', 'wpwa' ),
            'new_item_name' => __( 'New Project Type Name', 'wpwa' ),
            'menu_name'  => __( 'Project Type', 'wpwa' ),
          ),
          'hierarchical' => true,
          'capabilities' => array(
            'manage_terms' => 'manage_project_type',
            'edit_terms' => 'edit_project_type',
            'delete_terms'=> 'delete_project_type',
            'assign_terms'  => 'assign_project_type'
          ),
        )
      );
  }
```

Before moving on to the code explanation, I would like you to refresh the project creation area to see the two blocks added to the right of your screen to define technologies and types for projects, as shown in the following screenshot:



The preceding code illustrates the default structure of the custom taxonomy creation function with all the necessary options. There is nothing new in technology taxonomy other than the use of instance variables for the taxonomy name and custom post type. Astute readers might notice the difference in the `project type` implementation. We have added a section called capabilities to the project type. In today's world, Web technologies change rapidly. So, we need to provide the ability for developers to define any new technology into our application. On the other hand, project types are fixed and won't change regularly. Hence, we need to block the project type creation for user roles other than admin.

By default, WordPress uses the `manage_categories` permission for all of the taxonomies, including the default categories and tags. Since we didn't define specific capabilities for technologies, it will use the default `manage_categories` permission. So, anyone who has the permission to manage categories will have the ability to create new technologies. Now let's consider the capabilities of `project type`:

```
'capabilities' => array(
  'manage_terms' => 'manage_project_type',
  'edit_terms' => 'edit_project_type',
  'delete_terms' => 'delete_project_type',
  'assign_terms' => 'assign_project_type'
)
```

These four permissions called `manage_terms`, `edit_terms`, `delete_terms`, and `assign_terms` are used to handle default permissions. Here we need to handle the permissions of project type separately and hence we have assigned four custom permission types to the respective keys. Now take a look at the project creation menu in the admin area. You will notice that the **Technology** menu is displayed and the **Project Type** menu is not visible. Since we have defined custom capabilities, even the administrator does not have permissions until we assign them.

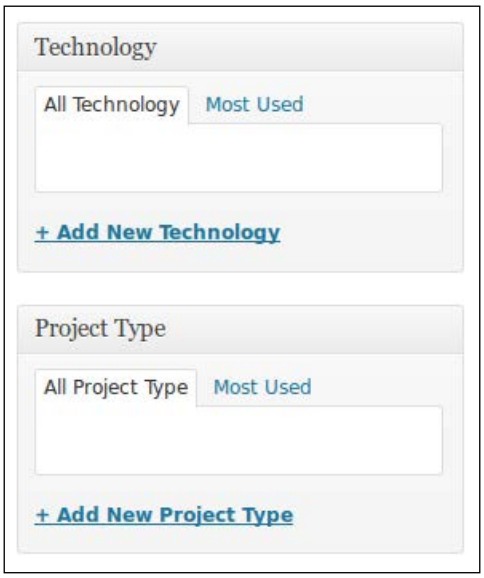
Assigning permissions to project type

We added custom capabilities in the project type creation process, but WordPress will have no idea about those capabilities until we assign them to a specific user role. In *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*, we installed the members' plugin to manage user roles. So, you can navigate to **Users | Roles | Edit Administrator** to see all of the available capabilities. You won't see the new capabilities on this screen.

Now open the `wpwa-user-manager.php` file again in the `Packt WPWA User Manager` plugin with the updated code used in the preceding section on project permissions. Navigate to the `add_application_user_capabilities` function and change the existing code as follows:

```
public function add_application_user_capabilities() {
  $role = get_role('follower');
  $role->add_cap('follow_developer_activities');

  $developer = get_role("developer");
  $custom_developer_capabilities = array(
    "edit_posts",
    "edit_private_posts",
    "edit_published_posts",
    "publish_posts",
    "read",
    "delete_posts",
    "manage_project_type",
    "edit_project_type",
    "delete_project_type",
    "assign_project_type",
  );

  foreach ($custom_developer_capabilities as $capability) {
    $developer->add_cap($capability);
  }

  $role = get_role('administrator');
  $custom_admin_capabilities = array("manage_project_type",
    "edit_project_type",
    "delete_project_type",
    "assign_project_type",
  );

  foreach ($custom_admin_capabilities as $capability) {
    $role->add_cap($capability);
  }
}
```

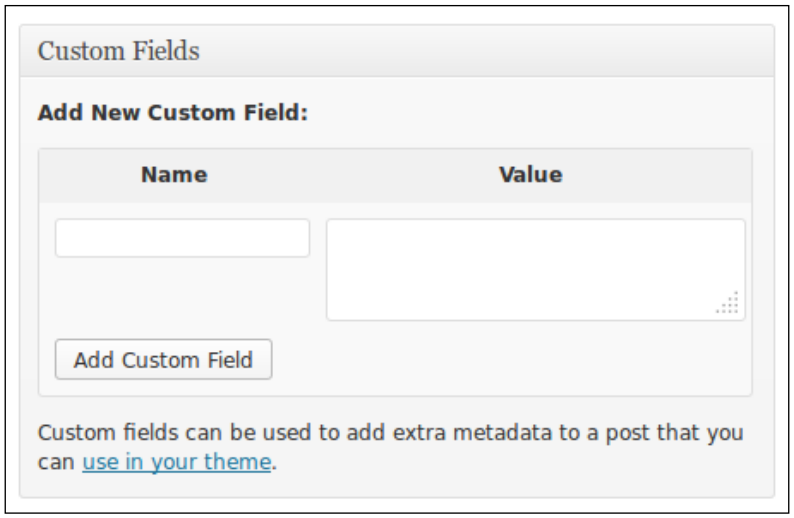
By now, you should know that capabilities cannot be defined without a user role. So, we get the administrator role as an object. Throughout this application, we are going to specify all of the custom capabilities into an array called `custom_admin_capabilities`. Finally, we add each of the custom capabilities inside the loop using the `add_cap` function. Once everything is saved, go to the **Plugins** section and deactivate the `Packt WPWA User Manager` plugin. Then, reactivate the plugin and navigate to the **Users** | **Roles** | **Edit Administrator** section to view the capabilities. Now you should be able to see the custom capabilities assigned to the admin role. Also, you will have access to the **Add New Project Type** link on the project creation screen, as shown in the following screenshot:



So far, we have created the defaults fields and taxonomies of the project creation screen. Now we come to the most important part of creating custom fields for custom post types. Let's get started.

Introduction to custom fields with meta boxes

Being a WordPress user, you should be familiar with custom fields as it's provided with default posts as well. We can enable custom fields on posts by clicking on the **Screen Option** menu on the top of the post creation screen and ticking the custom fields' checkbox. Once enabled, you will get a screen similar to the following screenshot:



The default custom fields section allows us to specify any key-value pair with the post. So, the user has complete control over the data created through these fields. In web applications, we need more control over the user input for restricting and validating data, hence, a default custom fields screen is not ideal for web applications.

Instead, we can use the same custom fields with a different approach using meta boxes. As developers, we have the control to decide the necessary fields on meta boxes rather than allowing users to decide their own key-value pairs. Let's modify the `Project` class constructor to add the necessary actions for the meta box creation of the `Projects` post type:

```
add_action('add_meta_boxes', array($this,
'add_projects_meta_boxes'));
```

We can use the `add_meta_boxes` action to define meta box creation functions for WordPress. Now let's implement the meta boxes inside the `add_projects_meta_boxes` function:

```
public function add_projects_meta_boxes() {
  add_meta_box("wpwa-projects-meta", "Project Details",
array($this, 'display_projects_meta_boxes'), $this->post_type);
}
```

It's not possible to directly implement the meta box fields without using the `add_meta_box` function. This function will decide the information and locations for creating the meta fields. The first parameter defines a unique key to the meta box (HTML ID attribute of the screen), while the second and third parameters define the meta box title and function respectively. The fourth parameter defines the associated post type; in this case it will be Projects. In case you want more advanced configurations, look at the documentation at `http://codex.wordpress.org/Function_Reference/add_meta_box`. Finally, we need to implement the `display_projects_meta_boxes` function defined in the preceding code to display custom fields. Now let's look at the most common implementation of such a function using the following code:

```
public function display_projects_meta_boxes() {
  global $post;

  $html = '<table class="form-table">';
  $html .= '<tr>';
  $html .= '<th><label for="Project URL">Project
URL</label></th>';
  $html .= '<td>';
  $html .= '<input class="widefat" name="txt_url" id="txt_url"
type="text" value=""  /></td>';
  $html .= '</tr>';
  $html .= '<tr>';
  $html .= '<th><label for="Project Duration">Project
Duration</label></th>';
  $html .= '<td><input class="widefat" name="txt_duration"
id="txt_duration" type="text" value="" /></td>';
  $html .= '</tr>';
  $html .= '</table>';

  echo $html;
}
```

Once you save the code and refresh the project creation screen, you will get a good meta box with two text fields, as defined in the preceding code. It works perfectly and most WordPress developers are comfortable with this technique of including HTML through PHP variables. Most experienced web developers will consider this technique as a bad practice. There are certain issues in including HTML in variables as listed here:

- There is difficulty in maintaining proper quotes in the right places. One invalid quote can break everything
- Template and logic codes are scattered in the same function
- There is difficulty in debugging codes

In web applications, we need to separate layers based on their functionalities. Therefore, we have to keep logic away from the templates so that designers know exactly what they are dealing with. So, it's preferable to go with a template engine for complex applications.

What is a template engine?

A template engine is a library or framework that separates logic from template files. These libraries provide their own syntaxes for passing necessary values to the template from the controllers or models. Once successfully implemented, we should only have simple `if-else` statements and loops inside the template files and there shouldn't be any complex code.

There are plenty of open source template engines available for PHP. Smarty, Mustache, and Twig are some of the popular ones amongst them. Throughout this book, we will be using the Twig templates engine created by **SensioLabs**. If you are not familiar with template engines, I suggest you look at the Twig documentation at `http://twig.sensiolabs.org/documentation`.

Now let's get started by integrating Twig templates into WordPress. You can grab a copy of the Twig library from the GitHub account at `https://github.com/fabpot/Twig`. Once downloaded, extract the ZIP file and you will get a folder called `Twig-master`. Now copy the whole folder into the `wpwa-custom-post-manager` plugin folder and rename it as `twig`.

For those who are not familiar with template systems, integration and implementation might look a little bit complex. I recommend you read the next section and try to get used to template systems.

Some of these open source libraries are built with a large number of classes. Hence, they usually do offer a built-in autoloader specific to a library. Navigate to the `twig/lib/Twig` folder inside the plugin folder and open the `Autoloader.php` file. The following code illustrates an extracted part from this `Twig_Autoloader` class:

```
public static function autoload($class){

  if (0 !== strpos($class, 'Twig')) {
    return;
  }
  if (is_file($file =
```

```
dirname(__FILE__).'/../'.str_replace(array('_', "\0"), array('/', ''),
$class).'.php')) {
    require $file;
  }
}
```

We used a prefix called `WPWA_Model` for our custom post types considering the autoloading process. Similarly, the Twig library uses the `Twig_` prefix for autoloading purposes and consistency. As you can see, the Twig autoloader works differently from the autoloader we created in the previous section. Open the `class-wpwa-custom-post-manager.php` file and include the following code to initialize the Twig autoloader:

```
$base_path = plugin_dir_path(__FILE__);
require_once $base_path.'/twig/lib/Twig/Autoloader.php';
Twig_Autoloader::register();
```

We keep the existing code to handle the auto-loading of the plugin classes. Then we place the Twig auto-loading code to load Twig-specific classes. Both the autoloaders will act independently due to the class name validation filters.

Configuring Twig templates

In order to work with templates, we need initial configurations for the Twig library. Create two folders inside the `plugins` folder and name them `templates` and `twig_cache` respectively. The `templates` folder will hold the HTML templates while the `twig_cache` folder will hold the cached versions of templates. Having completed the folder creation, we can move into the configurations by creating a new class called `Twig_Initializer` inside the `plugins` folder, as shown in the following code:

```
class Twig_Initializer {
  public static function initialize_templates() {
    $base_path = plugin_dir_path(__FILE__);
    $loader = new Twig_Loader_Filesystem( $base_path.'/templates'
);
    $twig = new Twig_Environment( $loader, array(
      'cache' => $base_path.'/twig_cache',
    );
    return $twig;
  }
}
```

The implementation of `initialize_templates` changes rarely, hence, it has been made into a static function. First, we have to define how templates are loaded into the Twig environment. There are several options available such as array, chain, filesystem, and string for loading templates. For this project, we will choose to go with loading templates as files. So, we have to initialize the `Twig_Loader_Filesystem` class with the path to the `templates` folder, as shown in the preceding code. Then, we set up the cache directory on the `Twig_Environment` class initialization. Finally, we return the `Twig_Environment` object to be re-used for template loading.

Now, we have to call this static function within our main plugin class for loading the `Twig_Environment` object. Consider the modified implementation `WPWA_Custom_Post_Manager` constructor, as shown in the following code:

```
class WPWA_Custom_Post_Manager {
  // Other instance variables
  private $template_parser;
  public function __construct() {
    $this->base_path = plugin_dir_path(__FILE__);
    require_once $this->base_path . 'twig_initializer.php';
    $this->template_parser =
Twig_Initializer::initialize_templates();
    $this->projects = new WPWA_Model_Project($this-
>template_parser);
  }
}
```

Since the `Twig_Initializer` class resides outside the `models` folder, we can't use the existing autoloader to load the class. Therefore, we need to include the file manually using the `require` or `include` statement. Then, we can call the `initialize_templates` function and assign the template object to the instance variable called `template_parser`. Finally, we have to pass the template object into each and every model using the class constructor. Also, we have to update the constructor of the `WPWA_Model_Project` class to retrieve the passed template object and initialize the `template_parser` instance variable. Now, the Twig environment is ready to function as a template engine.

Remember that we had issues in implementing custom fields without a proper template system. As a solution, we discussed template engines and chose Twig as the library for this particular application. Now, we can revert back to the meta box implementation to learn the proper ways of using templates in WordPress.

Creating your first Twig template

With the use of the Twig engine, all the templates have become pure HTML files instead of PHP. Create an HTML file called `project_meta.html` inside the `templates` folder and use the following code to define the form fields. Here, we have the complete template code for the Project meta box:

```
<input type="hidden" name="project_meta_nonce" value="{{ project_meta_
nonce }}" />
<table class="form-table">
  <tr>
    <th style=''><label for='Project Status'>Project Status
*</label></th>
  <td>
    <select class='widefat' name="sel_project_status"
id="sel_project_status">
      <option {% if project_status == 0 %}selected{% endif %}
value="0">Select</option>
      <option {% if project_status == 'planned' %}selected{% endif
%} value="planned">Planned</option>
      <option {% if project_status == 'pending' %}selected{% endif
%} value="pending">Pending</option>
      <option {% if project_status == 'failed' %}selected{% endif
%} value="failed">Failed</option>
      <option {% if project_status == 'completed' %}selected{%
endif %} value="completed">Completed</option>
    </select>
  </td>
  </tr>
  <tr>
    <th style=''><label for='Project Duration'>Project Duration
*</label></th>
    <td><input class='widefat' name='txt_duration'
id='txt_duration' type='text' value='{{ project_duration }}'
/></td>
  </tr>

  <tr>
    <th style=''><label for='Project URL'>Project URL</label></th>
  <td>

  <input class='widefat' name='txt_url' id='txt_url' type='text'
value='{{ project_url }}'  /></td>
```

```
  </tr>

  <tr>
    <th style=''><label for='Download URL'>Download
URL</label></th>
    <td><input class='widefat' name='txt_download_url'
id='txt_download_url' type='text'  value='{{ project_download_url
}}' /></td>
  </tr>
</table>
```

There are no PHP variables within the template. We start the template by defining the nonce value inside a hidden field for securing form submission. The value contained within the hidden field might be something new to you. Basically, the Twig engines use `{{ variable_name }}` for defining variables and the values need to be assigned from the `Model` class. Then, we have list of fields for the project URL, download URL, status, and duration. Each of these fields uses the `{{ }}` syntax for generating a value. Apart from normal variables, we have simple `if` conditions using the following syntax:

```
{% if  condition %} execute code {% endif %}
```

Now, all of the PHP code is replaced by Twig-specific syntaxes. So, there is a minor chance of breaking the code by mistake, compared to the direct usage of PHP variables.

In modern website design, the HTML table is not such a popular component for creating layouts. We prefer the `<div>` element-based structure for more flexibility. In the context of WordPress, it's ideal to use tables for designs to keep the consistency across all the admin screens. Also, you can use the CSS class called `'widefat'` on form fields for a better look and feel.

The next task will be to assign the created template to the project screen through meta boxes. So, we have to restructure the `display_projects_meta_boxes` function to use templates instead of hardcoded HTML elements. Here is the implementation with the use of a Twig template:

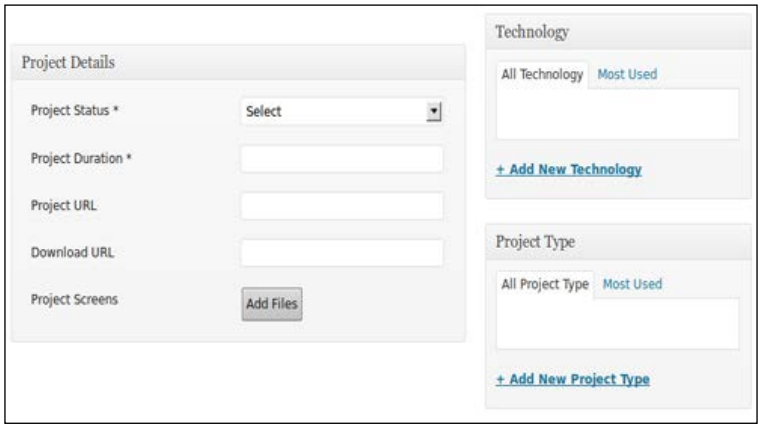
```
public function display_projects_meta_boxes() {
  global $post;

  $data = array();
  $data['project_meta_nonce'] =
wp_create_nonce(wp_create_nonce("project-meta"));

  echo $this->template_parser->render('project_meta.html', $data);
}
```

The effects of templates are noticeable even at first glance. The complexity of generating custom form fields has been reduced dramatically. We already initialized an object of the Twig loader within the constructor. So, we can use the `render` function on the same object to retrieve the template data from the `templates` folder. The first parameter to the `render` function defines the name of the template file, while the second parameter defines the necessary data as an array. In most occasions, dynamic templates will be populated with dynamic data, hence, we have to pass all the data as an array to the template.

WordPress uses the nonce value generation for securing and validating form submissions. So, we have assigned the nonce value to the data array with a key called `project_meta_nonce`. Within the template, you can use `{{ project_meta_nonce }}` to access the nonce value.

Nonce is used for security purposes to protect against unexpected or duplicate requests that could cause undesired, permanent, or irreversible changes to the website and particularly to its database. Specifically, a nonce is an one-time token generated by a website to identify future requests to that website. When a request is submitted, the website verifies if a previously generated nonce expected for this particular kind of request was also sent and decides whether the request can be safely processed or a notice of failure should be returned. This could prevent unwanted repeated, expired, or malicious requests from being processed.

Other than that, we also have to pass all of the existing values of form fields to the data array. Implementation of the existing value generation is skipped until we save the data for the first time. By now, you will have the complete project creation screen with the default fields, taxonomies, and custom fields, as illustrated in the following screenshot:



Persisting custom field data

You already know that default fields and taxonomies are automatically saved to the database on post publish. In order to complete the project creation process, we need to save the custom field data to the meta tables. As usual, we have to update the constructor to add necessary actions to match the following code:

```
public function __construct($template_parser) {
  // Instance variable initializations
  // Other actions
  add_action('save_post', array($this, 'save_project_meta_data'));
}
```

WordPress doesn't offer an out of the box solution for form validation as most websites don't have complex forms. This becomes a considerable limitation in web applications. So, let's explore the possible workarounds to reduce these limitations. The `save_post` action inside the constructor will only get called once the post is saved to the database with the default field data. We can do the necessary validations and processing for custom fields inside the function defined for the `save_post` action. Unfortunately, we cannot prevent the post from saving when the form is not validated properly. First, let's figure out the data saving process for custom fields using the following implementation:

```
public function save_project_meta_data() {
  global $post;
  if (!wp_verify_nonce($_POST['project_meta_nonce'], "project-
meta")) {
    return $post->ID;
  }
  if (defined('DOING_AUTOSAVE') && DOING_AUTOSAVE) {
    return $post->ID;
  }
  if ($this->post_type == $_POST['post_type'] &&
current_user_can('edit_post', $post->ID)) {
    // Implement the validations and data saving
  } else {
    return $post->ID;
  }
}
```

We begin the custom fields saving process by verifying the nonce against the value we generated in the form using the `wp_verify_nonce` function. Upon unsuccessful verification, we return the post ID to discontinue the process. Then, we have to execute a similar validation for the auto-saving process. Finally, we have to verify the post type and whether the current user has the permission to edit posts of this type.

These validations are common to custom field saving processes of any post type. Therefore, it's ideal to separate these checks into a common function to be re-used across multiple locations.

Once all the validations are successfully completed, we implement the data saving process as shown in the following code:

```
public function save_project_meta_data() {
  global $post;

  // Common validations
  if ($this->post_type == $_POST['post_type'] &&
current_user_can('edit_post', $post->ID)) {

    // Section 1

    $project_url = (isset( $_POST['txt_url'] ) ? (string) esc_url(
trim($_POST['txt_url']) ) : '');
    $project_duration = (isset( $_POST['txt_duration'] ) ? (float)
esc_attr( trim($_POST['txt_duration'] )) : '');
    $project_download_url   = (isset( $_POST['txt_download_url'] )
? (string) esc_attr( trim($_POST['txt_download_url'] )) : '');
    $project_status = (isset( $_POST['sel_project_status'] ) ?
(string) esc_attr( trim($_POST['sel_project_status'] )) : '');


    // Section 2
    if ( empty( $post->post_title ) ) {
      $this->error_message .= __('Project name cannot be empty.
<br/>', 'wpwa' );
    }
    if ( '0' == $project_status ) {
      $this->error_message .= __('Project status cannot be empty.
<br/>', 'wpwa' );
    }
    if ( empty( $project_duration ) ) {
      $this->error_message .= __('Project duration cannot be
empty. <br/>', 'wpwa' );
    }

    // Section 3
    if (!empty($this->error_message)) {
      remove_action('save_post', array($this,
```

```
'save_project_meta_data'));

      $post->post_status = "draft";
      wp_update_post($post);

      add_action('save_post', array($this,
'save_project_meta_data'));

      $this->error_message = __('Project creation failed.<br/>') .
$this->error_message;

      set_transient("project_error_message_$post->ID", $this-
>error_message, 60 * 10);

    } else {
      update_post_meta($post->ID, "_wpwa_project_url",
$project_url);
      update_post_meta($post->ID, "_wpwa_project_duration",
$project_duration);
      update_post_meta($post->ID, "_wpwa_project_download_url",
$project_download_url);
      update_post_meta($post->ID, "_wpwa_project_status", $project_
status);
    }
  } else {
    return $post->ID;
  }
}
```

The data saving process looks quite extensive and complex compared to the code we discussed up until now. So, I have broken the code into three sections to simplify the explanation process:

- **Section 1**: The initial code includes the retrieval of the form values to be stored in variables through the `$_POST` array. Here, we have to validate and filter the post data to avoid harmful data. Therefore, we used `trim` and the WordPress escape functions to filter the data. Finally, we cast the data into the proper data type in order to prevent invalid data submissions.
- **Section 2**: In this section, we implement the form validations for each and every form field. Once the validation fails, we can assign the error to the `error_message` instance variable to be used across the other function of this class. We can implement any type of complex validations in this section.

In case you have a large number of form fields with complex validations, integrating a third party library for validations might become a better solution than manual time consuming validations.

- **Section 3**: Here, we come to the tricky part of the validation process. Even though we execute validations and generate errors in section 2, it's not possible to prevent the project creation. So, we have chosen an alternative and poor way of handling the process.

  First, we remove the `save_post` action by using the `remove_action` function. This action should have the same syntax as the `add_action` function used in the constructor for `save_post`. In web applications, it's preferable to work with published data unless you are implementing custom application specific status. Hence, we set the post to `draft` status upon validation failure. Then we update the post to the database and add the `save_post` action back. Even though the post is still saved, we want to see it at the application frontend as we are only focusing on published data. Once the user submits the form without errors, it will revert to the publish status.

Once the project is successfully updated, WordPress will display the error as `Post draft updated`. We definitely need many more user-friendly errors to suit our applications. Therefore, we have to change the existing error messages generated by WordPress. Prior to that, we have to set the error to the `error_message` variable and save it in the database using the `set_transient` function.

Transient is a WordPress-specific technique for storing cached data in the database for temporary usage. Since WordPress uses a hooks- and actions-based procedure, it's not possible to get an error message after submission. So, we temporarily save the error message on the database to enable access from the post message handling function, which will be explained in a moment.

When the form is successfully validated without errors, we use the `update_post_meta` function on each field for saving or updating the data to the database. Having understood the code for the custom post saving function, we can revert to the error message handling process.

Customizing custom post type messages

By default, WordPress uses messages of normal posts for the custom post types. We need to provide our own custom messages to improve the user experience. Customization of messages can be done with the existing filter called `post_updated_messages`. First, we have to update the plugin constructor with following filter hook:

```
add_filter('post_updated_messages',
array($this,'generate_project_messages'));
```

This filter enables us to add new messages to the existing messages' array as well as alternate the existing messages to suit our requirements. Implementation of `generate_project_messages` differs from the commonly used code since we are handling the form validations manually to improve the process. Let's look at the implementation of the `generate_project_messages` function:

```
public function generate_project_messages($messages) {
  global $post, $post_ID;
  $this->error_message =
get_transient("project_error_message_$post->ID");
  $message_no = isset($_GET['message']) ? $_GET['message'] : '0';

  delete_transient("project_error_message_$post->ID");
  if (!empty($this->error_message)) {
    $messages[$this->post_type] = array("$message_no" => $this-
>error_message);
  } else {
    $messages[$this->post_type] = array(
      0 => '', // Unused. Messages start at index 1.
      1 => sprintf(__('Project updated. <a href="%s">View
Project</a>'), esc_url(get_permalink($post_ID))),
      2 => __('Custom field updated.'),
      3 => __('Custom field deleted.'),
      4 => __('Project updated.'),
      5 => isset($_GET['revision']) ? sprintf(__('Project restored
to revision from %s'), wp_post_revision_title((int)
$_GET['revision'], false)) : false,
      6 => sprintf(__('Project published. <a href="%s">View
Project</a>'), esc_url(get_permalink($post_ID))),
      7 => __('Project saved.'),
      8 => sprintf(__('Project submitted. <a target="_blank"
href="%s">Preview Project</a>'), esc_url(add_query_arg('preview',
'true', get_permalink($post_ID)))),
```

```
        9 => sprintf(__('Project scheduled for: <strong>%1$s</strong>.
<a target="_blank" href="%2$s">Preview Project</a>'),
        date_i18n(__('M j, Y @ G:i'), strtotime($post->post_date)),
esc_url(get_permalink($post_ID))),
        10 => sprintf(__('Project draft updated. <a target="_blank"
href="%s">Preview Project</a>'), esc_url(add_query_arg('preview',
'true', get_permalink($post_ID)))),
      );
    }
    return $messages;
}
```

WordPress uses the messages' array with ten keys to cater all the messages generated on the custom post screens. Once the **Publish** button is clicked, we validate the form and save the error messages as transients. However, WordPress will execute the whole process to generate the common error or message without considering our validations. You can find a parameter in the URL with a message as the key and a specific number as the value.

In order to show the validation errors, we need to intercept the WordPress generated message and change it according to our preference. First, we get the error message using the `get_transient` function and the default message number using `$_GET` array. After assigning the value to a variable, we remove the transient using the `delete_transient` function to prevent unnecessary database load.

Next, we check whether a specific error message exists in the database using the `get_transient` function. In case errors are generated, we update the existing messages array and set our own message to replace the WordPress-generated message number. In situations where we don't have form errors, we can use the complete array shown in the `else` part of the preceding function to include all the custom messages we need for specific custom post types.

Passing data to Twig templates

We discussed how to create a Twig template for the project screen in an earlier section of this chapter called *Creating your first Twig template*. We skipped the data passing process until the project data is saved to the database. Having completed the project saving process, we can move back into the data passing process of templates. Consider the following code for updated contents of the `display_projects_meta_boxes` function:

```
public function display_projects_meta_boxes() {
  global $post;
  $data = array();
  // Get the existing values from database
```

```
  $data['project_meta_nonce'] = wp_create_nonce("wpwa-project-
meta");
  $data['project_url'] = esc_url(get_post_meta( $post->ID,
"_wpwa_project_url", true ));
  $data['project_duration'] = esc_attr(get_post_meta( $post->ID,
"_wpwa_project_duration", true ));
  $data['project_download_url']   = esc_attr(get_post_meta( $post-
>ID, "_wpwa_project_download_url", true ));
  $data['project_status'] = esc_attr(get_post_meta( $post->ID,
"_wpwa_project_status", true ));

  // Render the twig template by passing the form data
  echo $this->template_parser->render( 'project_meta.html', $data
);
}
```

As explained earlier, the `render` function on the Twig object takes two parameters for the template name and necessary data. First, we have to decide the data required for the project template. Then, we have to create an array containing the project data in a separate key. Next, we retrieve the existing project data from the database and assign it to the specified `$data` array. It's important to filter all of the data with the necessary filter functions before passing them to templates. Inside the template, we have to define the Twig variables for all of the passed data.

In the preceding code, we used four parameters for project-related data. So, we have to define four variables called `{{ project_status }}`, `{{ project_duration }}`, `{{ project_url}}`, and `{{ project_download_url}}`. These are direct template variables. We will have to use different syntaxes for complex data such as loops, `if` conditions, and Twig functions.

Once the `$data` array is passed to the template function, the Twig engine compiles the data and the template to generate the output with the provided data. The dynamically generated output template will be stored inside the `twig_cache` folder.

Twig uses cached versions of templates from the `twig_cache` folder, so it's important to clear the cache folder before executing a modified template.

Now we have completed the basic foundation for implementing our Projects post type. Other post type creations are similar in nature, hence, I am not going to make things boring by explaining the implementation of each post type. You can use the source code of this chapter to play with the implementations of other post types and I highly recommend you change the code to understand the various aspects of custom post types.

Introduction to custom post type relationships

In general, we use relational databases in developing applications, where each model will be matched to a separate database table. Each model will be related to one or more than one module, but in WordPress, we have all of the custom post types stored in the posts table. Therefore, it's not possible to create relationships between different post types with the existing functionality.

WordPress developers around the world have been conducting discussions to get the post relationship capability built into the core. Even though the response from the WordPress core development seems positive, we still don't have it on the core version as of 3.6. You can have a look at an interesting discussion at: `http://make.wordpress.org/core/2013/07/28/potential-roadmap-for-taxonomy-meta-and-post-relationships/`.

Since this is one of the most important aspects of web application development, we have no choice other than looking for a custom solution built by external developers. There are a few competitive plugins amongst the open source community for providing the post relationship functionality. The **Posts 2 Posts** plugin developed by *Cristian Burca* and *Alex Ciobica* seems to be the plugin of choice of many developers.

In our portfolio application, we need to associate projects to services and vice versa, so let's see how we can use this cool plugin to implement the necessary features. First, grab a copy of the plugin from `http://wordpress.org/plugins/posts-to-posts/` and get it activated in your WordPress installation. This plugin doesn't offer a GUI for defining post relationship, hence, we need to implement some source code.

Let's get started by updating our `WPWA_Model_Project` class constructor with the following code:

```
add_action( 'p2p_init', array($this,
'join_projects_to_services'));
```

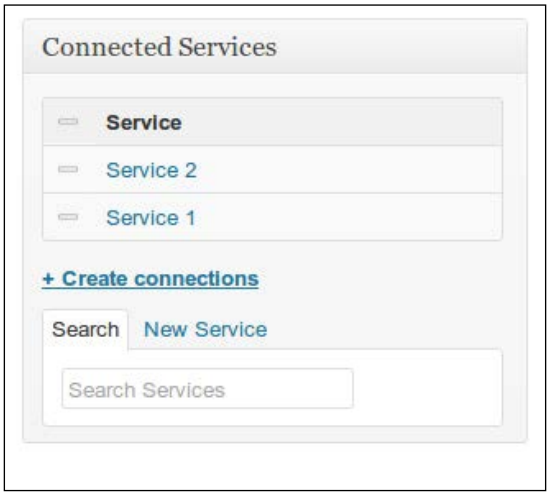
Here we have the `p2p_init` action, which is built-in with the Posts 2 Posts plugin. All the post relationship definitions and configurations go inside the `join_projects_to_services` function as illustrated in the following code:

```
public function join_projects_to_services(){

  p2p_register_connection_type( array(
    'name' => 'projects_to_services',
    'from' => $this->post_type,
```

```
        'to' => 'wpwa_service'
    ) );
}
```

The Posts 2 Posts plugin provides a function called `p2p_register_connection_type` for defining post type relationships. The first parameter called `name` defines a unique identifier for the relationship. Then, we can define two post types for the connection using the `from` and `to` parameters. This will enable the services selection section for projects and the project selection section for services. The following screenshot illustrates the project screen with the **Connected Services** box:

First, you have to create services from the **Services** item on the left-hand menu. Then, you can add any number of services to projects by clicking on the services' name. Once saved, you will have a screen similar to following screenshot with the associated services of the given project:



This is the most basic implementation of post relationships with the Posts 2 Posts plugin. If you are curious to learn more advanced usages, you can have a look at the plugin documentation at `https://github.com/scribu/wp-posts-to-posts/wiki`. In later chapters of this book, we are going learn how to retrieve and work with the related posts.

We have set up the foundation of our application backend throughout the chapter. Now, users with the role of a developer can log in to the system and create portfolio data for projects, services, books, and articles. Through the previous sections, we had to work with complicated WordPress functions related to custom post types and taxonomies. In complex applications, we need better quality code to avoid duplicates of code scattering around hundreds of files.

As a solution, we can develop our own custom post type specific library to simplify the implementations. Ideally, such a library should abstract the custom post type functions into a common interface and let users choose to define their configurations without forcing them to do so. Implementing such a library is a time consuming and complex task, that is beyond the scope of this book. So, I suggest you look for existing open source libraries or take time to plan your own library. An alternate solution will be to take advantage of popular existing frameworks such as Pods, which will be introduced shortly.

Pods framework for custom content types

Up until this point, we explored the advanced usage of WordPress custom post types by considering the perspective of web applications. Although custom post types provide immense power and flexibility for web applications, manual implementation is a time consuming task with an awful amount of duplicate and complex code. One of the major reasons for using WordPress for web development is its ability to build things in a rapid process. Therefore, we need to look for quicker and more flexible solutions beyond manual custom post implementations.

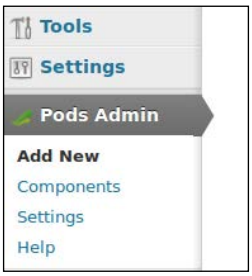
Pods is a custom content type management framework, which is becoming popular among developers. Most of the tedious functionalities are baked into the framework while providing us with a simpler interface for managing custom content types. Apart from simplifying the process, Pods provides an extensive list of functionalities that are not available with default WordPress administrative screens.

Let's consider some of the key features of the Pods framework, compared to manual implementation of custom post types:

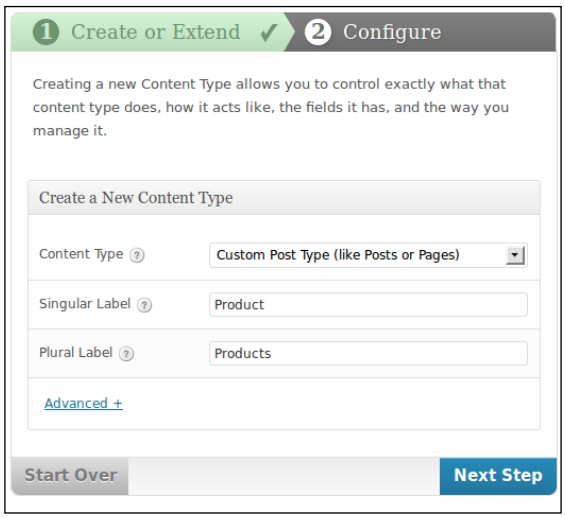
- Creating, extending, and managing custom types and fields, including custom post types, taxonomies, comments, and users
- Using default tables or custom tables for content types
- Listing of built-in form fields and components with necessary validations
- Managing form field level permissions
- Creating page settings for plugins and themes

Basically, this framework provides out of the box functionalities to cater to common tasks in WordPress application development without much hassle. For example, think how much effort you need to put in to add a custom field for your comment form. With the Pods framework, you will be able to implement such tasks with a few clicks in ten minutes. Pods does have an active community and can be recommended for rapid application development with WordPress.

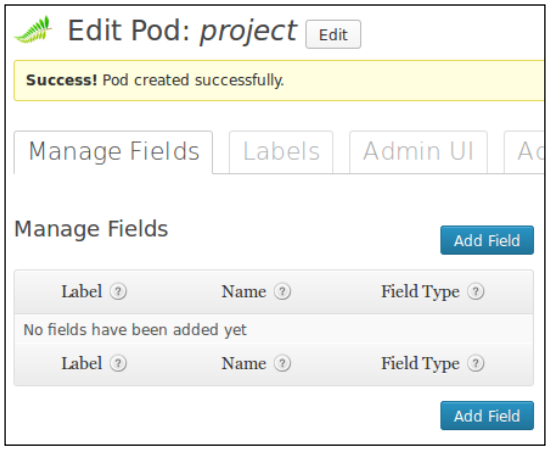
Let's get our hands dirty by implementing something practical with this awesome framework. First, you have to grab a copy of this plugin from `http://pods.io` and get it installed in your WordPress folder. Once activated, you should see the **Pods Admin** menu as shown in the following screenshot:



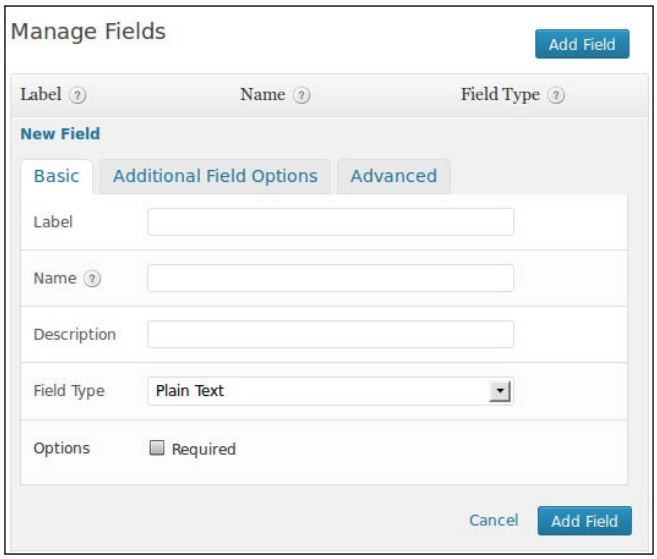
Earlier, we implemented projects post type by manually implementing the custom post type code. Here, we are going to see how the Pods framework simplifies the same process. Keep in mind that we are going to create a basic implementation to show the power of Pods instead of implementing the whole task we have already completed. Click on **Create New Section** to add a new content type and you will get a screen similar to the following screenshot:



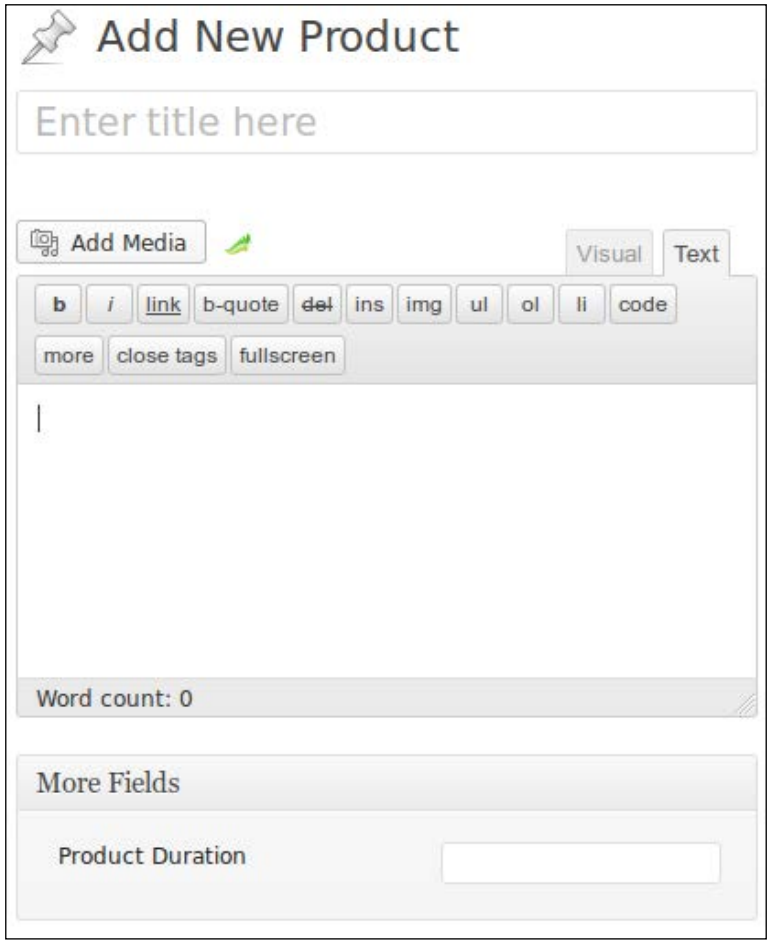
In the content type creation screen, you can select the custom post type and add the labels as `Product` since it's not wise to redefine projects. Then, move to the next step and you should see what has been shown in the following screenshot:



Surprisingly, we have the custom post ready in three clicks. The next task is to add the necessary custom fields and associate them with the `Products` post type. Remember the tasks we did in our manual custom field creation process; we had to create HTML for fields, use template engines, implement validations, and so on. The Pods framework abstracts all these tasks behind the framework allowing us to focus on application logic. So, click on the **Add Field** button to create the first custom field, as shown in the following screenshot:



All of the necessary configurations are divided into three tabs on the top of the screen. You can create the field by defining the necessary labels, validations, and access restrictions. Pods provides over 15 built-in field types including data pickers, color pickers, file fields, and text editors. Once all the fields are created, click on the **Save Pod** button. Now, navigate to the `Product` creation screen and you will find all of the custom fields created with Pods, as shown in the following screenshot:



We were able to create a complete custom post type with the necessary fields within ten minutes. This is one of the most basic usages of the Pods framework. You can find more advanced and different usages from the official site: `http://podsframework.org/`.

Should you choose Pods for web development?

I would definitely go with the Pods framework for applications that require a rapid development process with a low budget. It's maturing into an excellent framework with the support of the open source community. The Pods framework offers more advantages and fewer limitations compared to similar competitive plugins and frameworks.

The ability to choose between existing tables versus custom tables can be quite handy when it comes to complex web applications. We have to enable the **Advanced Content Type** component in order to make use of custom tables. It's recommended to use the existing tables in every possible scenario, but there can be scenarios where custom tables are a better option to default tables:

- **Extensive data load**: In applications where you have a large number of records with a number of custom post types, it's wise to separate post types into custom tables for better performance.
- **More control**: Using existing tables will force us to stick with the features provided in WordPress by default. In situations where you need complete control over your data, it's better to choose custom tables instead of the existing ones.

In short, you should be using this framework or a similar one for reducing repetitive work and focus on application-specific tasks. If you choose not to go with such a framework, you should definitely have your own common library to interact with custom content type related functions for maintaining the quality of code and building maintainable systems.

There is a popular phrase that states "with great power comes great responsibility." We have used some very powerful plugins throughout this chapter and will be continuing to do so in the remaining chapters. As developers, we have the huge responsibility of working with these plugins. It can be dangerous to rely on third-party plugins to build the core of your application. The following are some of the risks of using third party plugins:

- Plugins can break due to WordPress upgrades
- Developers might discontinue the development of plugins
- Plugins might not be updated regularly

Having said that, all the plugins used throughout this book are highly popular and stable. So, it's important to know how these plugins work in order to customize them at later stages if needed. Also, it's better not to rely heavily on third-party plugins and keep alternatives whenever possible.

Time to practice

In this chapter, we discussed the advanced usages of custom post types to suit web application functionalities. Now I would recommend you to try out the following tasks in order to evaluate what you learned in this chapter:

- Assume that we have to provide access to Projects and Services for developers while blocking the access to Articles and Books. How can we change the existing implementation to provide the preceding features?
- In the post relationships section, we enabled relationships using the `join_projects_to_services` function. With the use of this function, we generate a new problem considering the possibility of extending. First, find the issue and try to solve it by yourself.
- We enabled relationships between custom post types. Research the possibilities of including metadata for relationships.

Summary

In this chapter, we tackled the custom post types to learn their advanced usages within web applications. Generally, custom posts act as the core building blocks of any type of complex web application. We went through the basic code of custom post related functionality while developing the initial Projects post type of the portfolio management application.

We were able to explore advanced techniques such as autoloaders, modularizing custom post types, and template engines to cater to common web application requirements. Also, we had to go through the process of validating form data, which was tough due to the limited support offered by WordPress.

We learned the importance of relating custom post types using the Posts 2 Posts plugin. Finally, we explored the possibilities of improving the custom post type management process with the use of an amazing framework called Pods.

In the next chapter, we are going to learn how to use the WordPress plugin beyond its conventional usage by implementing pluggable and extendable plugins. Until then, get your hands dirty by playing with custom post type plugins.

5
Developing Pluggable Modules

Plugins are the heart of WordPress, which make web applications possible. WordPress plugins are used to extend the core features as independent modules. As a developer, it's important to understand the architecture of WordPress plugins and design patterns in order to be successful in developing large scale applications.

Everyone that has basic programming knowledge can create plugins to meet the application-specific requirements, but it takes considerable effort to develop plugins that are reusable across a wide range of projects. In this chapter, we are going to build a few plugins to demonstrate the importance of reusability and extensibility. WordPress developers who don't have a lot of experience in web application development shouldn't skip this chapter as it's the most important part of web application development.

I assume that you have sound knowledge about basic plugin development using the exciting WordPress features in order to be comfortable in understanding the concepts discussed in this chapter.

In this chapter, we will cover the following topics:

- A brief introduction to WordPress plugins
- WordPress plugins for web development
- Understanding different types of plugins for web development
- Creating a reusable AJAX library with plugins
- Creating an extensible file uploading plugin
- Integrating a media uploader into custom fields
- Exploring the use of pluggable functions

Let's get started.

A brief introduction to WordPress plugins

WordPress offers one of the most flexible plugin architectures compared to other similar frameworks such as Joomla! and Drupal. The existence of over 20,000 plugins in the WordPress plugin directory proves the vital role of plugins. In typical websites, we create simple plugins to tweak theme functionalities or application-specific tasks. The complexity of web applications forces us to modularize the functionalities to enhance the maintainability. Most of the existing application developers will be familiar with the concept called **Open Close Principle**.

Open Close Principle states that the design and writing of code should be done in a way so that a new functionality is added with minimum changes in the existing code. The design should be done in a way that allows the adding of the new functionality as new classes, keeping the existing code unchanged. Refer to `http://www.oodesign.com/open-close-principle.html` for more information.

We can easily achieve Open Close Principle with WordPress plugins. Plugins can be developed to be open for new features through actions, filters, and pluggable functions, while remaining closed for modifications. Additional features will be implemented using separate plugins, which can be activated or deactivated any time without breaking the existing code.

Understanding the WordPress plugin architecture

WordPress is not the most well documented framework from the perspective of architectural diagrams and processes, so you won't find detailed explanations on how plugins actually work behind the scenes. Plugins need to have a main file that includes the block comment in a predefined format in order for WordPress to identify it as a plugin. Activation and deactivation of plugins can be done any time by using the plugins section of the admin area. WordPress uses meta field in the `wp_options` table called `active_plugins` to keep a list of the existing active plugins. The following screenshot previews the contents of the `active_plugins` field using the **phpMyAdmin** database browser.

Field	Type	Value
option_id	bigint(20) unsigned	35
option_name	varchar(64)	active_plugins
option_value	longtext	a:4:{i:0;s:40:"Packt WPWA User Manager/user_manager.php";i:1;s:19:"members /members.php";i:2;s:13:"pods /init.php";i:3;s:48:"wpwa_custom_post_manager /custom_post_manager.php";}
autoload	varchar(20)	yes
		Go

In the initial execution process, WordPress loads each and every active plugin through its main file. From there onwards, action hooks and filters will be used to initialize the plugin functions. We can use built-in hooks as well as custom hooks inside the plugins. As a WordPress plugin developer, it's important to understand how the action hooks run during a typical request to avoid conflicts and improve the performance and quality of the plugins. You can visit `http://codex.wordpress.org/Plugin_API/Action_Reference` to understand the action execution procedure and see how plugins fit into the execution process.

WordPress plugins for web development

By default, WordPress offers blogging and CMS functionalities to cater to simple applications. In real web applications, we need to develop most of the applications using the existing features provided by WordPress. In short, all those web application related features will be implemented using plugins. Throughout this book, we created a few plugins intended for providing implementations for the portfolio management application. All those plugins were focused on a specific task in our application and hence were not reusable in different projects. As developers who are willing to take a long journey in web application development with WordPress, we need more and more reusable plugins and libraries. In this section, we are going to discuss various such plugins:

- Reusable libraries
- Extensible plugins
- Pluggable plugins

Keep in mind that plugins are categorized into the preceding types conceptually for the sake of understanding the various features of plugin development. Don't look for the details using the preceding categories as they are not defined anywhere.

Create reusable libraries with plugins

These days web developers will rarely go without frameworks and libraries in application development. The main purpose of choosing such frameworks is to reduce the amount of time required for common tasks in application development. In WordPress, we do need similar libraries to abstract the common functionalities and keep our focus on the core business logic of the application.

AJAX has become one of the most popular technologies in recent years to provide a smooth user interaction in modern applications. WordPress itself uses AJAX for its core features like auto saving, image uploading, menu management, and more. In modern web applications, we might need hundreds of different functionalities with AJAX requests. So, we need to have a proper method for reusing the common tasks of handling AJAX requests. Here, we are going to build a simple AJAX plugin to abstract the common functionalities in order to make it reusable as well as identify the importance of creating reusable WordPress plugins.

How to use AJAX in WordPress

As with most well-established frameworks, WordPress offers a well-defined unique method for using AJAX within WordPress plugins. In general, we use separate PHP files for interacting with AJAX requests. Even though we are allowed to use the same method in WordPress, it's recommended to use the built-in technique provided through the `admin-ajax.php` file located in the `wp-admin` folder for handling AJAX requests. So let's make an AJAX request using the recommended method.

Creating an AJAX request using jQuery

jQuery is built into WordPress and most developers seem to prefer it over other client-side libraries in plugin development, so we are going to use jQuery throughout this process of creating AJAX plugins. It's important to know that you can use similar libraries like Prototype, Dojo, or plain JavaScript code for making AJAX requests. Let's identify the basic code required for creating an AJAX request:

```
<script type="text/javascript" >
jQuery(document).ready(function($) {
  var data = {
  action: 'wpwa_create_posts'
```

```
  };
  $.post(http://www.example.com/wp-admin/admin-ajax.php, data,
function(response) {
    alert('Response from the server: ' + response);
  });
});
</script>
```

Here, we have an AJAX post request that executes on a page load. In the preceding code, we have hardcoded the full path to the `admin-ajax.php` file. In the actual development process, we need to use `admin_url( 'admin-ajax.php' )` from the PHP file and pass it to the JavaScript file. Also, you might notice the use of an action parameter in a data variable. In WordPress, it's essential to pass the action parameter so that the `admin-ajax.php` file can identify the server-side handler function for the request. Having identified the client-side code, we can now move on to server-side definitions and handlers.

Defining AJAX requests

In normal circumstances, we identify the server-side handler function through URL parameters or hidden variables. The WordPress way of handling the requests is quite unique compared to other frameworks. First, we have to define the AJAX handlers using predefined syntaxes as shown in the following code:

```
add_action('wp_ajax_wpwa_create_posts', 'wpwa_create_posts');
add_action('wp_ajax_nopriv_wpwa_create_posts',
'wpwa_create_posts');
```

We need to define two actions for handling the request, where the `wp_ajax_` prefix is for logged-in users and `wp_ajax_nopriv_` is for users who are not logged in. In case we need the request for logged-in users, the `wp_ajax_nopriv_` action can be omitted. After the prefix, we have the action sent from the JavaScript code. The second parameter will be the function name of the handler. Here we have used the same action name for the handler function as well. You can use a different function name within the same file or inside any other class. Once these actions are defined, WordPress knows how to match the request against the handlers. Then, we can implement the server-side function as follows:

```
function wpwa_create_posts(){
  // Get data from post
  // Check AJAX referrers and necessary validations
  // Execute the logic
  // Echo the data to the front end script
}
```

This is the most basic way of handling AJAX requests within WordPress. Now let's identify the drawbacks of this technique in web applications.

Drawbacks in a normal process

It's obvious that we need to define client-side request actions and handler functions for each and every request. This technique works well for small scale applications and websites that don't require heavy usage of AJAX. For applications that depend highly on AJAX, we need to organize the code to avoid these recurring tasks and only focus on what varies across different requests. As a solution, we are going to develop a basic AJAX plugin to cater to these common tasks.

Planning the AJAX plugin

The main goal of creating this plugin is to reduce the common code required for making AJAX requests. Hence we need to identify the common code discussed in the default process. Basically, the post or get request created from the client side is common across all the requests with the exception of parameter values. So, we need a common function to execute the AJAX request by passing the dynamic data. The common function should be able to handle the dynamic parameters such as the URL, data, success, and error handlers.

On the server side, we need a method to automatically define the necessary actions instead of defining them manually for each and every request. Also, the handler function should be defined dynamically when creating the actions.

Finally, we have the AJAX handler functions on the server side. Generally, these functions contain less common code, so we are not going tackle the implementations of the handler function within our plugin. In case you want to remove the common code from the handler functions, you can take the following path within your plugins:

1. Define a main handler function name for each and every action definition with a dynamic parameter.
2. Execute the validations and security checks.
3. Call a dynamic function based on the parameters passed to the main handler function, to execute the logic and return the response.
4. Retrieve the response from the dynamic function and send it back to the client side.

Now we are going to start the development of our plugin to handle the reusable code in client-side requests and action definitions.

Creating the plugin

As usual, the initial tasks of creating the plugin includes the creation of the plugin folder and files, and adding the plugin definition. Create a folder named `wpwa-ajax` inside the `wp-content/plugins` folder and create the main plugin as `class-wpwa-ajax.php`. Next, we can include the plugin definition and the main plugin class as defined in the following code:

```
<?php
/*
  Plugin Name: WPWA AJAX
  Plugin URI:
  Description: Common library for making ajax requests
  Version: 1.0
  Author: Rakhitha Nimesh
  Author URI: http://www.innovativephp.com/
  License: GPLv2 or later
 */
class WPWA_AJAX {

  public function __construct() {
    // Initializations goes here
  }

}

$ajx = new WPWA_AJAX();
?>
```

We have used the same plugin creation technique by defining the main class as `WPWA_AJAX` and creating an object to initialize the plugin through the constructor. Now we can start the implementation of the plugin.

Remember that we used actions in the default AJAX request creation process. A unique action name was repeated in both the client-side and server-side code. In case we want to change the action name, we have to modify two places. In order to avoid such repetitive tasks and the possibility of making mistakes while defining actions, we can use a common actions' array to be re-used in both client and server sides. Let's update the plugin to include the custom actions within our applications, as shown in the following code:

```
class WPWA_AJAX {
  private $ajax_actions;

  public function __construct() {
```

```
      $this->configure_actions();
    }
    public function configure_actions() {
      $this->ajax_actions = array(
        "sample_key" => array("action"=>"sample_action","function"
=> "sample_function_name" , "logged" => TRUE),
        "sample_key1" => array("action"=>"sample_action1","function"
=> "sample_function_name1"),
      );

    }
}
```

In this modified implementation, we have used an instance variable called `ajax_actions` to hold the custom actions in an array. The configuration of actions is done within the `configure_actions` function initialized inside the constructor. Inside the function, you can see the configurations array in a predefined format. Each action has a subarray within the main array with a unique key.

The `action` key defines the name of the action while the `function` key defines the name of the function. Now, you might wonder why we didn't use the action name as the main key instead of a separate unique key. With the action name as the key, we again come to the repetitive code in both the client and server sides. Once we define a unique key in both the client and server code, we can easily change the action name at a later stage by modifying the configuration array. Finally, we have a key called `logged` to filter the action across logged-in users and users not logged in. Now let's see how we can automate the action defining process by using the following code:

```
public function configure_actions() {
  $this->ajax_actions = array(
    "sample_key" => array("action"=>"sample_action","function" =>
"sample_function_name" , "logged" => TRUE),
    "sample_key1" => array("action"=>"sample_action1","function"
=> "sample_function_name1"),

  );

  foreach ($this->ajax_actions as $custom_key => $custom_action) {
    if (isset($custom_action["logged"]) &&
$custom_action["logged"]) {
      add_action("wp_ajax_".$custom_action['action'], array($this,
```

```
$custom_action["function"]));

    } else if (isset($custom_action["logged"]) &&
!$custom_action["logged"]) {

      add_action("wp_ajax_nopriv_".$custom_action['action'],
array($this, $custom_action["function"]));

    } else {

      add_action("wp_ajax_nopriv_".$custom_action['action'],
array($this, $custom_action["function"]));
      add_action("wp_ajax_".$custom_action['action'], array($this,
$custom_action["function"]));

    }
  }
}
```

Here, we have looped the elements of the custom array to define the necessary AJAX actions for WordPress. Filtering of actions is done by matching the logged key value from the configuration array. The `logged` value can be either true or false, where `TRUE` is used for logged-in users and `FALSE` is used for users who are not logged in. Omitting the `logged` key in the configuration array makes the action available for users who are logged in as well as not logged in.

Based on the `logged` parameter, we define the necessary actions and functions using the values from the configuration array. Now you can see that we have limited the action definition code for the entire request into a few lines of code. Whenever you want a new AJAX request, you can add a new entry to the configurations array.

Including plugin scripts for AJAX

In the previous section, we completed the task of automating the action definitions in WordPress for the AJAX requests. We need to configure the necessary scripts to handle the AJAX functions prior to the implementation of the common AJAX request functions. As usual, we need to update the plugin constructor with the following code to define the actions for the inclusion of scripts:

```
add_action('wp_enqueue_scripts', array($this, 'include_scripts'));
```

Now, we can move into the implementation of the `include_scripts` function as illustrated in the following code:

```
public function include_scripts() {
  global $post;

  wp_enqueue_script('jquery');

  wp_register_script('wpwa_ajax', plugins_url('js/wpwa-ajax.js',
__FILE__), array("jquery"));
  wp_enqueue_script('wpwa_ajax');

  $nonce = wp_create_nonce("unique_key");

  $config_array = array(
    'ajaxURL' => admin_url('admin-ajax.php'),
    'ajaxActions' => $this->ajax_actions,
    'ajaxNonce' => $nonce,
  );

  wp_localize_script('wpwa_ajax', 'wpwa_conf', $config_array);
}
```

Since we are depending on jQuery for AJAX requests, it's essential to load the WordPress built-in jQuery version using the `wp_enqueue_script` function.

WordPress provides the latest stable jQuery version with every new release. It's recommended to use the existing file to avoid conflicts of duplicate jQuery versions with other plugins. If required, you can always load the custom jQuery version using the `wp_register_script` function, but make sure to be careful when adding jQuery manually.

Next, we register a custom script file for including the common AJAX request function and the necessary requests. It's important to validate the AJAX requests before executing and hence we create a nonce value using a unique key on the `wp_create_nonce` function.

Create a new folder named `js` inside the `wpwa-ajax` plugin folder and put an empty JavaScript file called `wpwa-ajax.js` in it. We are using a custom JavaScript file for the plugin and hence we cannot use PHP statements inside the script file to get the necessary configuration data. WordPress provides a cool and simple way of passing server-side data to script files using the `wp_localize_script` function. First, we need to create an array with all the option values required for script files. In the preceding code, we have used the AJAX URL, actions, and nonce value as the options. The `wp_localize_script` function takes three parameters, where the first one defines the unique key of the JavaScript file followed by the JavaScript object name and configuration data.

The `wp_localize_script()` function *must* be called after the script it's being attached to has been queued or registered. It doesn't put the localized script in a queue for later scripts.

We used the `wpwa_ajax` key for loading the `wpwa-ajax.js` file and hence the configuration data will be accessible inside the script using `wpwa_conf` as the variable. Now let's look at the following screenshot firebug console to identify how `wp_localize_script` works:

```
<script type="text/javascript">
  1
  2 /* &lt;![CDATA[ */
  3 var conf = {"ajaxURL":"http:\/\/localhost\/wp36\/wp-admin\/ad
  4 ,"ajaxActions":{
  5    "sample_key":{"action":"sample_action","function":"sample_
  6    "sample_key1":"action":"sample_action1","function":"sample
  7 /* ]]&gt; */
</script>
<script src="http://localhost/wp36/wp-content/plugins/wpwa-ajax/js
/wpwa-ajax.js?ver=3.6" type="text/javascript">
<script src="http://localhost/wp36/wp-includes
/js/underscore.min.js?ver=1.4.4" type="text/javascript">
```

As shown in the screenshot, the configuration data is included as an inline script before the loading of the `wpwa-ajax.js` file. Hence, it will be available for all the js files following the `wpwa-ajax.js` file. Now the scripts are loaded, we can begin the implementation of the common AJAX request inside the `wpwa-ajax.js` file.

Creating reusable AJAX requests

Earlier, we created an AJAX request to illustrate the default usage inside WordPress. Now, we have to make the same request into a common function to be reusable across multiple requests without duplicating the common values. Consider the following code inside the `wpwa-ajax.js` file for making dynamic AJAX requests:

```
var ajaxInitializer = function(options){

  var defaults = {
    type: 'POST',
    url: wpwa_conf.ajaxURL,
    data: {},
    beforeSend:"",
    success:"",
    error:""
  };

  var settings = $jq.extend({}, defaults, options);

  $jq.ajax(settings);
}
```

We have named the function `ajaxInitializer` and abstracted all the common code inside the function. First, we can define the default values for the necessary parameters.

Keep in mind that we have used limited options for this defaults array. You can find the complete parameter list at `http://api.jquery.com/jQuery.ajax/`.

Here, we have used the `wpwa_conf` variable passed from the `wp_localize_script` function to access the AJAX URL. Then, we use the jQuery extend function to merge the default options with the dynamic options passed as the parameters of the `ajaxInitializer` function. The default options will be overridden by the values inside the options array. Finally, we can pass the merged options' array as settings to the jQuery `ajax` function to complete the AJAX request.

You might have noticed the use of `$jq` for jQuery selectors instead of the default `$` sign. In WordPress, there are a number of built-in libraries that use the `$` syntax, creating the possibility of conflicts. Hence, we have assigned `$jq` in a no-conflict mode to avoid potential issues, using the following code:

```
$jq =jQuery.noConflict();
```

Now, we have the ability to re-use this function by passing dynamic data based on different requests. Consider the following code for a sample request using the `ajaxInitializer` function:

```
$jq(document).ready(function() {

  ajaxInitializer({
    "success":"sample_ajax_sucess",
    "data":{
      "name":"John Doe",
      "age":27,
      "action" : wpwa_conf.ajaxActions.sample_key.action,
      "nonce" : wpwa_conf.ajaxNonce
    }
  });

});
```

Here, we have passed the `data` and `success` handler functions name to the `ajaxInitializer` function on page load. This means we are using the default values for all the other options. The nonce value and action is retrieved from the `wpwa_conf` array passed from the `wp_localize_script` function. Note that we are using the key to access the respective actions from the `ajaxActions` array. So, the changes to the action name won't affect the existing code. Now let's implement the `success` handler to print the output from the server to the console.

```
var sample_ajax_sucess = function(data, textStatus, jqXHR){
  console.log(data);
}
```

Next, we can complete the process by implementing the server-side AJAX handler function for `sample_action`, as shown in the following code:

```
function sample_function_name() {

  $nonce = $_POST['nonce'];

  if (!wp_verify_nonce($nonce, 'unique_key'))
  die('Unauthorized request!');

  echo json_encode($_POST);
  exit;
}
```

Here, we are just verifying the nonce value and sending the post data back to the client side. Now, activate the plugin from the admin dashboard and reload the home page of the site while keeping the browser inspections section open. You will see the AJAX request and response as shown in the following screenshot:

Through this process, we were able to build a simple and reusable plugin for handling the AJAX requests without code duplication. This plugin can be customized to improve the functionality as well as suit your coding style. In the next few chapters, we will be using this plugin to handle the AJAX request of our portfolio management application.

Extensible plugins

In the previous section, we created a reusable plugin for AJAX requests, but the plugin doesn't allow us to extend the core features other than providing dynamic parameter passing. Here, we will be exploring the possibility of creating plugins that other developers can extend using their own plugins to change the existing behavior or add new behavior. WordPress uses its actions and filters technique to extend plugins. So, we are going to create a reusable and extensible plugin for automating the file upload process for custom meta fields. Let's get started.

Planning the file uploader for portfolio application

WordPress offers a built-in media uploader for handling all the file uploading tasks within applications. The simplicity and adaptability of a media uploader is one of the keys to its success in CMS development. Web applications require heavy usage of custom meta fields and there can be a number of file fields within a single screen. Integrating the media uploader to each and every field can become a tedious and unnecessary task, so we need a method to automatically integrate the file fields with the media uploader. In *Chapter 4*, *The Building Blocks of Web Applications*, we created all the custom post types and fields for the portfolio management application, but we skipped the screen uploading process for projects. Here, we are going to complete the implementation while building an extensible plugin. So let's begin with the planning:

- All the meta file fields should be automatically converted to buttons, which would open the media uploader on a click event

- A dynamic container needs to be created to gather multiple images within a single field
- Plugin developers should be able to extend the plugin by customizing the media uploader interface for limiting the allowed file types

Before we begin the implementation, it's necessary to modify the `Custom Posts Manager` plugin created in the previous chapter, to include the file field for uploading project screenshots. Open the `wpwa-custom-post-manager` plugin and navigate to the templates folder. Include the following code after the `Download URL` field code in the `project_meta.html` file:

```
<tr>
  <th ><label for='Project Screens'>Project Screens</label></th>
  <td>
  <input class='widefat wpwa_multi_file' type="file"
id="project_screens"  />
  </td>
</tr>
```

Here, we have added a file field for uploading project screens using the default HTML tags used throughout the template files. The CSS class called `wpwa_multi_file` is used as the identifier for the file field conversion. Once the file uploading plugin is implemented, this file field will be converted into a button and a container for keeping the uploaded images.

Creating the extensible file uploader plugin

As usual, we begin the implementation by creating a new plugin. This time we are going to name it `WPWA File Uploader`. Create a folder named `wpwa-file-uploader` inside the `wp-content/plugins` folder and implement the main plugin file as `class-wpwa-file-uploader.php`:

```
<?php
/*
  Plugin Name: WPWA File Uploader
  Plugin URI:
  Description: Automatically convert file fields into multi file
uploaders.
  Version: 1.0
  Author: Rakhitha Nimesh
  Author URI: http://www.innovativephp.com/
```

```
    License: GPLv2 or later
   */

  class WPWA_File_Uploader {

    public function __construct() {

    }
  }
  $file_uploader = new WPWA_File_Uploader();
```

According to the plan, the initial task is to convert the file fields into a button and a container that works with the media uploader. Conversions need to be done from the client side through jQuery or plain JavaScript. Therefore, we have to include the necessary scripts in the file uploader plugin as illustrated in the following code:

```
  class WPWA_File_Uploader {

    public function __construct() {
      add_action('admin_enqueue_scripts', array($this,
  'include_scripts'));
    }

    public function include_scripts() {
      wp_enqueue_script('jquery');

      if (function_exists('wp_enqueue_media')) {
        wp_enqueue_media();
      } else {
        wp_enqueue_style('thickbox');
        wp_enqueue_script('media-upload');
        wp_enqueue_script('thickbox');
      }

      wp_register_script('wpwa_file_upload', plugins_url('js/wpwa-
  file-uploader.js', __FILE__), array("jquery"));
      wp_enqueue_script('wpwa_file_upload');
    }
  }
```

The `admin_enqueue_scripts` action is used for the script inclusion since we only need the plugin to work in the admin side. Based on your requirements, the `wp_enque_scripts` action can also be used to enable the conversion at the frontend.

Create a new folder named `js` inside the `wpwa-file-uploader` folder and put an empty JavaScript file as `wpwa-file-uploader.js`. First, we include jQuery into the plugin since the media uploader and the `wpwa-file-uploader.js` file depend on jQuery. WordPress' latest version uses a modified media uploader, which is simple and interactive compared to the iFrame-based uploader provided in the earlier releases. So, now we have to check the availability of the `wp_enqueue_media` function. Then we can load the necessary scripts and styles based on the available WordPress version. Finally, we register the `file_uploader.js` file as `wpwa_file_upload` for defining the custom code required for the plugin.

Converting file fields with jQuery

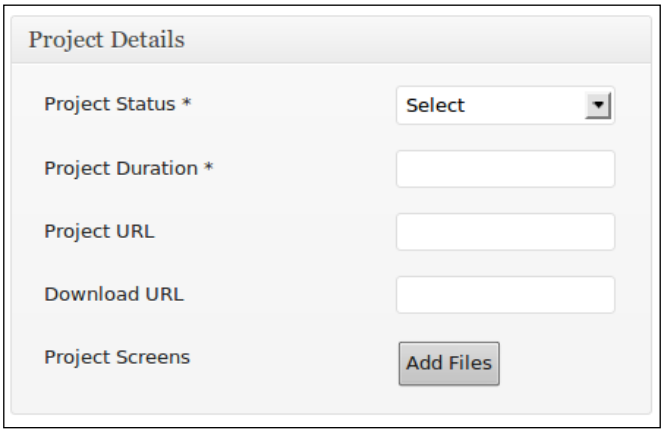
Now, we can begin the conversion of file fields into the media uploader-integrated buttons and image containers. While creating the file field for project screens, we assigned a special CSS class called `wpwa_multi_file`. This class is used to identify the file fields that need to be converted. Let's get started with the implementation inside the `wpwa-file-uploader.js` file:

```
$jq =jQuery.noConflict();

$jq(document).ready(function(){
  $jq(".wpwa_multi_file").each(function(){
    var fieldId = $jq(this).attr("id");

    $jq(this).after("<div id='wpwa_upload_panel_"+ fieldId +"'
></div>");
    $jq("#wpwa_upload_panel_"+ fieldId).html("<input type='button'
value='Add Files' class='wpwa_upload_btn' id='"+ fieldId +"' />");

    $jq("#wpwa_upload_panel_"+ fieldId).append("<div
class='wpwa_preview_box' id='"+ fieldId +"_panel' ></div>");

    $jq(this).remove();
  });
});
```

We begin the implementation by introducing the jQuery `noConflict` variable. Each loop of jQuery is used to traverse through all the file fields with the CSS class of `wpwa_multi_file`. Then, we assign the ID of the file field into the `fieldId` variable. Next, we insert a DIV container after the file field to keep the button and image container. Next, we assign the button to the main container with a class called `wpwa_upload_btn`. Then, we can append the image container with a class called `wpwa_preview_box`. All the containers are given dynamic IDs with a static prefix to be used for the media uploader handling. Finally, we remove the file field using the jQuery remove method.

Make sure you define the `wpwa_multi_file` class on file fields to avoid potential conflicts. Otherwise, you need to check the type of field for each element with the `wpwa_multi_file` class.

Now, all the file fields with the CSS class `wpwa_multi_file` will be converted into a dynamic button and image container. The image container will not be visible until the images are uploaded. Hence your `Project Screens` field will look something like the following screenshot:



Make sure you clear the `twig_cache` folder's contents to see the updated screen with the **Add Files** button.

Having completed the field conversion, we can now move to the media uploader integration process.

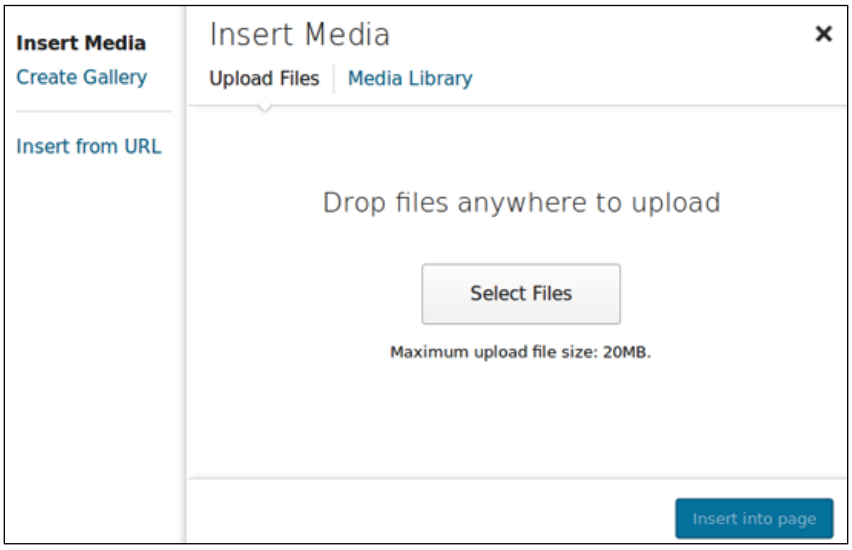
Integrating the media uploader to buttons

WordPress provides a quick and flexible way of integrating the media uploader to any type of field. Implementation can vary based on the WordPress version. We are using version 3.6 throughout the book, and hence we can use the following code snippet for integration:

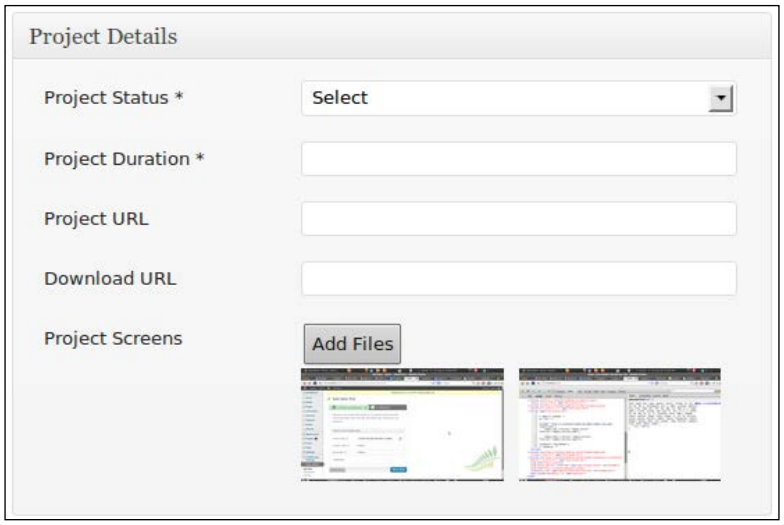
```
$jq(".wpwa_upload_btn").click(function(){

  var uploadObject = $jq(this);
  var sendAttachmentMeta = wp.media.editor.send.attachment;

  wp.media.editor.send.attachment = function(props, attachment) {
```

```
    $jq(uploadObject).parent().find(".wpwa_preview_box").append("<img
class='wpwa_img_prev' style='with:75px;height:75px' src='"+
attachment.url +"' />");

    $jq(uploadObject).parent().find(".wpwa_preview_box").append("<inpu
t class='wpwa_img_prev_hidden' type='hidden' name='h_"+
$jq(uploadObject).attr("id")
    +"[]' value='"+ attachment.url +"' />");


    wp.media.editor.send.attachment = sendAttachmentMeta;
  }
  wp.media.editor.open();
  return false;
});
```

Earlier in the process, we used a class called `wpwa_upload_btn` for every button. Here, we are using the click event of those buttons to load the media uploader. We start the process by assigning the `wp.media.editor.send.attachment` function into a variable. This function takes two parameters called `props` and `attachment` by default. The path of the uploaded file can be retrieved using the `attachment.url` property. Then, we assign the image preview using the URL and assign the URL to a hidden field to be used in the saving process. Finally, we call the `wp.media.editor.open` function to load the media uploader on the click of a button. Once completed, click on the **Project Screens** button and you will get the modern media uploader as illustrated in the following screenshot:



Upload an image for the `Project Screens` field and click on the **Insert into page** button at the bottom to assign a preview of the image inside the dynamic image container, under the upload button of the `Project Screens` field. You will end up with something similar to the following screenshot:



The custom file uploader section created in the preceding sections is fully functional at this stage. We can add any number of images one by one using the **Upload** button. Flexibility adds more value to any type of plugin. Even though we can insert multiple images at the moment, we don't have a method to remove them. Let's create some simple jQuery code to remove the assigned images on a double-click, as shown in the following code:

```
$jq("body").on("dblclick", ".wpwa_img_prev" , function() {
  $jq(this).parent().find("wpwa_img_prev_hidden").remove();
  $jq(this).remove();
});
```

In jQuery, the dynamically created elements can't be assigned directly to events. We need to use the `on` function to attach events to the dynamically created elements. Here, we have specified the `on` function on the `body` tag. You can choose any related element according to your preference. We have assigned the `dblclick` event to the `wpwa_img_prev` class specified inside the dynamically created images. Then, we remove the `img` tag and the related hidden fields from the preview box. Try uploading a few images and double-click on the image in the preview area to see the effect in action.

Extending the file uploader plugin

Remember that we created this plugin to illustrate the extending capabilities of the WordPress plugins for web development. So far, we have completed the core functionality of this plugin to upload images through the custom meta fields. Now, we have to think about the extensible features and hook points within the plugin. Let's think about the possible features required for such plugins:

- Customizing the allowed types of images
- Customizing the media uploader features
- Validating image sizes and dimensions

These are a few of the most important enhancements and there can be more such enhancements depending on your project. In this section, we are going to look at the first requirement for creating the extending capabilities.

Customizing the allowed types of images

Usually, we do allow `jpg`, `jpeg`, `png`, and `gif` types in image uploads, but there can be occasions where we need more control over the allowed file types. So, let's see how we can change the allowed file types within WordPress. Add the following line of code to the constructor of the file uploader plugin:

```
add_filter('upload_mimes', array($this, 'filter_mime_types'));
```

Now, consider the implementation of the `filter_mime_types` function for the image restricting process as shown in the following code:

```
function filter_mime_types($mimes) {
  $mimes = array(
    'jpg|jpeg|jpe' => 'image/jpeg',
  );
  return $mimes;
}
```

WordPress passes the existing mime types as the parameter to this function. Here, we have modified the mimes array to restrict the image types to `jpg`. This means that only `jpg` files will be allowed for each and every post type within WordPress. Ideally, this filtering should be extensible to allow for different file types based on application requirements. Usually, WordPress developers tend to redefine the `upload_mimes` filter with another function to cater to such requirements. It's not the best practice to redefine the same filter or action in multiple locations, making it almost impossible to identify the order of the execution, unless specific priority values are given.

Those who are not familiar with the `priority` parameter for actions and filters can take a look at the official documentation at `http://codex.wordpress.org/Function_Reference/add_filter`.

A better solution is to define the filter at specific places and allow developers to extend through custom actions. Let's consider the modified implementation of the preceding code with the usage of actions:

```
function filter_mime_types($mimes) {
  $mimes = array(
    'jpg|jpeg|jpe' => 'image/jpeg',
  );
  do_action_ref_array('wpwa_custom_mimes', array(&$mimes));
  return $mimes;
}
```

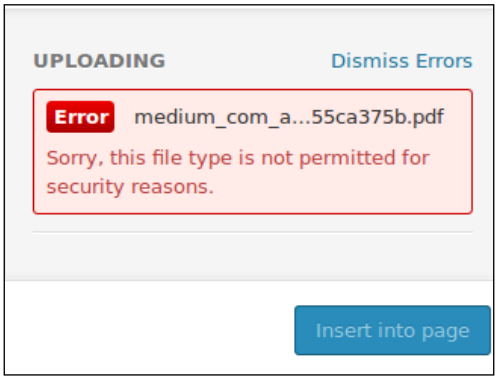
In the modified version, we have a WordPress action called `wpwa_custom_mimes`. With the use of an action, any developer can extend the function to include their own requirements. In this code, the mimes array is passed as a reference variable to the action. Therefore, the original mimes array can be modified through the extended versions. WordPress uses global variables for most of the functionalities. Experienced web developers don't prefer the use of global variables. Hence, I have used reference passing instead of using global variables.

The functionalities of WordPress, `do_action` and `do_action_ref_array`, are similar. Usually, most developers use the `do_action` function. Here, we have used `do_action_ref_array` since reference variable passing is not supported by `do_action`.

Now, let's extend the functionality using the custom function on the specified action hook, as shown in the following code:

```
function wpwa_custom_mimes(&$mimes) {
  $mimes['png'] = 'image/png';
}
add_action("wpwa_custom_mimes", "wpwa_custom_mimes");
```

This implementation can be defined inside the theme or any other plugin file. First, we take the mimes array as a reference variable. Then we can add the required mime types back to the mimes array. Since we are using reference passing, we don't need to return the mimes array. Now upload the files to the projects section and you will notice that only the `png` format will be allowed. For other formats, you will get an error as shown in the following screenshot:



Now, we have successfully extended the plugin without touching the code of the core function. In complex application development, make sure to include actions and filters in proper places to allow for extension at later stages. Now, we have built an extensible file uploader plugin for the portfolio application. Finally, we need to take the necessary steps to save the uploaded images to projects.

Saving and loading project screens

Once again, we have to modify the `Custom Posts Manager` plugin created in the previous chapter to handle the saving and loading process of project screens. In this chapter, we updated the `project_meta.html` template to include the project screens upload field. Now, we have to save the uploaded plugins to the database. Consider the following code included after the bunch of update meta statements in the `save_project_meta_data` function in the `project.php` file:

```
$project_screens = isset ($_POST['h_project_screens']) ? $_POST['h_
project_screens'] : "";
$project_screens = json_encode($project_screens);
update_post_meta($post->ID, "_wpwa_project_screens",
$project_screens);
```

We can retrieve the list of uploaded images using the hidden field inline with every image. Then, we save all the project screens in a JSON string using a meta table key called `_wpwa_project_screens`. Next, we have to retrieve the list of project screens to be displayed on the project load. Here is the updated version of the function for loading the existing images:

```
public function display_projects_meta_boxes() {
  global $post;
  $data = array();
  // Get the exisiting values from database
  $data['project_meta_nonce'] = wp_create_nonce("wpwa-project-
meta");
  $data['project_url'] = esc_url(get_post_meta( $post->ID,
"_wpwa_project_url", true ));
  $data['project_duration'] = esc_attr(get_post_meta( $post->ID,
"_wpwa_project_duration", true ));
  $data['project_download_url'] = esc_attr(get_post_meta( $post-
>ID, "_wpwa_project_download_url", true ));
  $data['project_status'] = esc_attr(get_post_meta( $post->ID,
"_wpwa_project_status", true ));

  $data['project_screens'] = json_decode(get_post_meta($post->ID,
"_wpwa_project_screens", true));

  // Render the twig template by passing the form data
  echo $this->template_parser->render( 'project_meta.html', $data
);
}
```

The display function is updated to include the retrieval of project screens from the database using the `_wpwa_project_screens` key. Then, we assign the screens' data to the template file as we did with other meta fields. Finally, we have to modify the template file to display the existing image previews as shown in the following code of the `project_meta.html` file:

```
<tr>
  <th style=''><label for='Download URL'>Project
Screens</label></th>
  <td><input class='widefat wpwa_multi_file' type="file"
id="project_screens"  />

    <div class='wpwa_preview_box' id='project_screens_panel' >
    {% for item in project_screens %}
    <img class='wpwa_img_prev' style='with:75px;height:75px'
src='{{item}}' />
```

```
    <input class='wpwa_img_prev_hidden' type='hidden'
name='h_project_screens[]' value='{{item}}' />

    {% endfor %}
    </div>
  </td>
</tr>
```

After the file field, we include the image preview, the image hidden field, and the preview box container in the same format we used in the file uploader `js` file. Here, you can notice the usage of Twig loops inside the template files. We include the existing images from the database for the project. You can delete any existing image and add new images to update the project screens at any time.

Now, we have completed the project creation by including the project screen upload section. You can test the plugin by adding and removing project screens.

Pluggable plugins

WordPress provides the ability to use pluggable functions through its pluggable architecture. Even though pluggable functions are no longer added to the core, there are many plugins and themes that take advantage of this technique. These plugins can be considered as a different version of extensible plugins. We used actions and filters to create extensible plugins. Here, we use functions that are pluggable through custom implementations. In web application terms, we can think of it as a very basic version of inheritance. Instead of inheritance, WordPress prefers extending through functions. Let's build a simple test plugin to understand the use of pluggable plugins, using functions.

As usual, we start with the plugin folder creation and definition. Create a folder named `wpwa-pluggable-plugin` and create the main file named `wpwa-pluggable-plugin.php`, as shown in the following code:

```
<?php
/*
  Plugin Name: WPWA Pluggable Plugin
  Plugin URI:
  Description: Explain the use of pluggable plugins by sending
mails on post saving
  Version: 1.0
  Author: Rakhitha Nimesh
  Author URI: http://www.innovativephp.com/
  License: GPLv2 or later
 */
?>
```

Assume that we need a plugin to send newsletters to user's emails. The following code is a basic implementation of such a requirement using a pluggable function:

```
if (!function_exists('wpwa_send_newletter')) {
  function wpwa_send_newletter($heading, $content) {

    $message = "<p><b>$heading</b><br/></p>";
    $message .= "<p>$content<br/></p>";

    wp_mail("example@gmail.com", "Pluggable Plugins", $message);
  }
}
```

We have created a function called `wpwa_send_newletter` to hold the e-mail heading and content, and have sent an e-mail message to the specified address. It's important to consider the use of the `function_exists` check. First, it allows us to check whether a function with the same name is already defined. This function will be executed when other functions with the same name are not available. So, plugin developers can redefine the function to extend the capabilities of the core function.

In the extensible plugins, we extended part of the functionality using actions and filters. With pluggable functions, we need to recreate the complete implementation instead of a part.

Now, we can move to the plugged version of this function. You can define the modified function inside any other plugin. Here, I have kept both the functions inside the same plugin for simplicity, as shown in the following code:

```
function wpwa_send_newletter($heading, $content , $template_name = "")
{
  $message = "";
  if(empty($template_name)){
    $message = "<p><b>$heading</b><br/></p>";
    $message .= "<p>$content<br/></p>";
  }else{
    $template = wpwa_get_template($template_name);
    $message .= str_replace("%title%",$heading,$template);
    $message   = str_replace("%content%",$content,$message);
  }
  wp_mail("example@gmail.com", "Pluggable Plugins", $message);
}
function wpwa_get_template($template_name) {
  $template = "";
  switch($template_name){
```

```
      case 'projects':
      $template .= "<h2>%title%</h2><br/><p><i>%content%</i></p>";
      break;
    }
    return $template;
  }
```

In the plugged version, we have an additional parameter to pass the template dynamically. Earlier we used a fixed template inside the function. The template is made optional to prevent issues with the existing code. The plugged function has two sections for handling the fixed template and the dynamic template. So, all the existing function calls to `wpwa_send_newletter` will work without any issues using the fixed template. All the new function calls will work by passing a dynamic template name. Here, we have used another function called `wpwa_get_template` to get the respective template. Now let's look at the execution of a newsletter sending function on the post's save and update process:

```
add_action('save_post', 'wpwa_create_newsletter');
function wpwa_create_newsletter($post_id) {
  if ( !wp_is_post_revision( $post_id ) ) {
    $post_title = get_the_title($post_id);
    $post_url = get_permalink($post_id);
    wpwa_send_newletter($post_title, $post_url,"projects");
  }
}
```

The WordPress `save_post` action allows us to call the custom function on post save or update. Here, we are calling the `wpwa_send_newletter` function with the post title as the heading and the post URL as the content. Also, we have used a template called `projects`.

With pluggable functions, we can turn the new functionalities on or off any time without affecting the existing code. Since WordPress uses procedural function calling, pluggable plugins through functions make sense. If you prefer OOP-based plugins, you can choose inheritance over pluggable functions to build pluggable plugins. Once the preceding code is completed and the plugin is activated, you can enter your e-mail and create some posts to see the usage of pluggable functions.

So far, we have discussed various types of reusable plugins suitable for web applications. Pluggable plugins with procedural functions is not the most popular method amongst developers. Instead, it's recommended that you extend plugins with WordPress actions and filters or use inheritance with object-oriented plugins.

Time to practice

Developing high-quality plugins is the key to success in web development using WordPress. In this chapter, we introduced various techniques for creating extensible plugins. Now it's time for you to take one step further by exploring the various other ways of using plugins. Take some time and try out the following tasks to get the best out of this chapter.

- In this chapter, we integrated the media uploader to custom fields and restricted the file types using actions. But the restrictions will be global across all types of posts. Try to make the restrictions based on custom post types and custom fields. (You should be able to customize the media uploader for each field).
- Use the `wp_handle_upload` function to implement manual file uploading to cater to complex scenarios, which cannot be developed using the existing media uploader.
- Create extensible plugins using global variables instead of actions and filters.
- Create pluggable plugins using inheritance without considering pluggable functions.

Summary

We began this chapter by exploring the importance and architecture of WordPress plugins. In previous chapters, we developed standalone plugins to cater to application-specific requirements. Here, we identified the importance of creating reusable plugins by categorizing such plugins into three types called reusable libraries, extensible plugins, and pluggable plugins.

While building these plugins, we learned the use of actions, filters, and pluggable functions within WordPress. Integration of the media uploader was very important for web applications, which works with file-related functionalities.

In *Chapter 6*, *Customizing the Dashboard for Powerful Backends*, we are going to master the use of the WordPress admin section to build highly customizable backends using the existing features. Stay tuned, as this will be important for developers who are planning to use WordPress as a backend system without using its theme.

6
Customizing the Dashboard for Powerful Backends

Usually, developers build an application's backend from scratch as full stack PHP frameworks don't provide built-in admin sections. WordPress is mainly built on an existing database and hence, makes it possible to provide a pre-built admin section. Most of the admin functionality is developed to cater to the existing content management functionality. As developers, you won't be able to develop complex applications without having the knowledge of extending and customizing the capabilities of existing features.

The structure and content of this chapter is built in a way that enables tackling of the extendable and customizable components of admin screens and features. We will be looking at the various aspects of an admin interface using popular frameworks and libraries while building the portfolio management application.

In this chapter, we will cover the following topics:

- Understanding the admin dashboard
- Customizing the admin toolbar
- Customizing the main navigation menu
- Adding features with custom pages
- Building options pages with **Slightly Modded Options Framework** (**SMOF**)
- Using feature-packed admin list tables
- Awesome visual presentations with admin themes
- The responsive nature of the admin dashboard

Understanding the admin dashboard

WordPress offers one of the most convenient admin sections among similar frameworks such as Drupal and Joomla! for building any kind of application. In the previous chapters, we looked at the administration screens related to various areas such as user management, custom post types, and posts. Here, we are going to look at some of the remaining components in the perspective of web application development. Let's identify the list of sections we are going to consider:

- The admin toolbar
- The main navigation menu
- Option and menu pages
- Admin list tables
- Responsive design capabilities

Customizing the admin toolbar

The admin toolbar is located at the top of the admin screen to allow direct access to the most used parts of your website. Once you log in, the admin toolbar will be displayed on the admin dashboard as well as at the frontend. Typical web applications contain separate access menus for the frontend and backend. Hence, web developers might find it difficult to understand the availability of the admin toolbar at the frontend from the perspective of the functionality as well as the look and feel. In web applications, it's your choice to remove the admin toolbar from the frontend or customize it to provide a useful functionality. In this section, we are going to look at both the methods to simplify your decision on the admin toolbar. First, let's preview the admin toolbar at the frontend with its default settings as shown in the following screenshot:

As usual, we are going to provide these functionalities through a custom plugin. Let's begin by creating the main plugin file called `class-wpwa-dashboard.php` inside a new folder called `wpwa-dashboard`, as shown in the following code:

```
<?php
/*
  Plugin Name: WPWA Admin Dashboard
  Plugin URI:
  Description: Customize admin dashboard to suit web applications.
  Version: 1.0
  Author: Rakhitha Nimesh
  Author URI: http://www.innovativephp.com/
  License: GPLv2 or later
 */
class WPWA_Dashboard {
  public function __construct() { }
}
$admin_dashboard = new WPWA_Dashboard();
?>
```

Activate the plugin from the plugins section to get things started.

Removing the admin toolbar

WordPress allows us to configure the visibility settings of the admin toolbar at the frontend. Unfortunately, it doesn't provide a way to remove the toolbar from the backend. Let's consider the following implementation for removing the admin toolbar from the frontend:

```
class WPWA_Dashboard {
  public function __construct() { }
  public function set_frontend_toolbar($status) {
    show_admin_bar($status);
  }
}
$admin_dashboard = new WPWA_Dashboard();
$admin_dashboard->set_frontend_toolbar(FALSE);
```

Here, we use a function called `set_frontend_toolbar` to dynamically set the visibility of the admin toolbar at the frontend. WordPress uses the `show_admin_bar` function with a Boolean condition to implement this functionality. You might have noticed the difference in implementation compared to the plugins developed in the previous chapters. Earlier, we used to initialize all the functions through the plugin constructor using actions and filters. Setting the admin toolbar can be implemented as a standalone function without actions or filters. Hence, we call the `set_frontend_toolbar` function on the `admin_dashboard` object. Here, we have used the `FALSE` value to hide the admin toolbar at the frontend.

Managing the admin toolbar items

Default items in the admin toolbar are designed to suit generic blogs or websites, and hence it's a must to customize the toolbar items to suit web applications. The **Profile** section in the top-right corner is suitable for any kind of application as it contains common functionalities such as **Edit Profile**, **Log out**, and **Profile Picture**. Hence, our focus should be on the menu items on the left side of the toolbar. First, we have to identify how menu items are generated in order to make the customizations. So, let's look at the following code for retrieving the available toolbar menu items list:

```
 add_action( 'wp_before_admin_bar_render', array( $this,
'wpwa_customize_admin_toolbar' ) );
```

As usual, we start by adding the necessary actions to the constructor of the dashboard plugin as shown in the following code:

```
public function wpwa_customize_admin_toolbar() {
  global $wp_admin_bar;
  $nodes = $wp_admin_bar->get_nodes();
  echo "<pre>";
  var_dump($nodes);
  exit;
}
```

We have access to the `wp_admin_bar` global object inside the `wpwa_customize_admin_toolbar` function. All the toolbar items of the current page will be returned by the `get_nodes` function. Then, we can use `print_r` on the returned result to identify the nodes. The following is a part of the returned nodes list, and you can see the main item IDs called `user-actions` and `user-info`:

```
Array
(
  [user-actions] => stdClass Object
  (
```

```
      [id] => user-actions
      [title] =>
      [parent] => my-account
      [href] =>
      [group] => 1
      [meta] => Array()
    )
    [user-info] => stdClass Object
    (
      [id] => user-info
      [title] =>developer developerdeveloper
      [parent] => user-actions
      [href] => http://localhost/packt/wordpress-web-develop-
test/wp-admin/profile.php

    )
)
```

We need to use those unique IDs to add or remove menu items. Now we are going to remove all the items other than the first item and create menu items specific to the portfolio application. So, remove the preceding code and modify the `wpwa_customize_admin_toolbar` function as follows:

```
public function wpwa_customize_admin_toolbar() {
  global $wp_admin_bar;
  $wp_admin_bar->remove_menu('updates');
  $wp_admin_bar->remove_menu('comments');
  $wp_admin_bar->remove_menu('new-content');
}
```

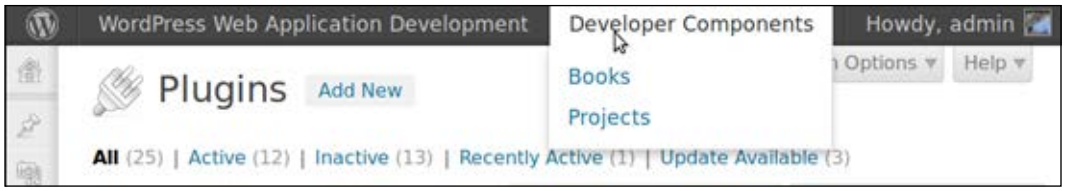
By default, the admin toolbar contains three items for site updates, comments and new posts, pages, and so on. Explore the result from `print_r` and you will find the respective keys for the preceding items such as `updates`, `comments`, and `new-content`. Then, use the `remove_menu` function on the `wp_admin_bar` object to remove the menu items from the toolbar. Now the toolbar should look like the following screenshot:

Next, we need to add application-specific items to the toolbar. Since we are mainly focusing on developers, we can have a menu called **Developers** to contain links to projects, books, articles, and services, as shown in the following updated code of the `wpwa_customize_admin_toolbar` function:

```
public function wpwa_customize_admin_toolbar() {
  global $wp_admin_bar;
  // Remove menus
  if (current_user_can('edit_posts')) {
    $wp_admin_bar->add_menu( array(
      'id'    => 'wpwa-developers',
      'title' => 'Developer Components',
      'href'  => admin_url()
    ));

    $wp_admin_bar->add_menu( array(
      'id'    => 'wpwa-new-books',
      'title' => 'Books',
      'href'  => admin_url()."post-new.php?post_type=wpwa_book",
      'parent'=>'wpwa-developers'
    ));
    $wp_admin_bar->add_menu( array(
      'id'    => 'wpwa-new-projects',
      'title' => 'Projects',
      'href'  => admin_url()."post-
new.php?post_type=wpwa_project",
      'parent'=>'wpwa-developers'
    ));
  }
}
```

The WordPress `wp_admin_bar` global object provides a method called `add_menu` to add new top menus as well as submenus. The preceding code contains the top menu item called `developers`, containing two submenu items for books and projects. Other menu items can be implemented similarly and omitted here for simplicity. When defining submenus, we have to use the ID of the top menu for the parent attribute. It's important to make the menu item IDs unique to avoid conflicts. Finally, we define the URL to be invoked on the menu item click using the `href` attribute. We can use any internal or external URL for the `href` attribute.

We have validated the permission called `edit_posts`, as only developers are allowed to create books and projects. Make sure you check the necessary permission levels while building custom admin toolbars. The following screenshot previews the admin toolbar with custom menu items:



Now we have the ability to extend the admin toolbar to suit various applications. Make sure you add or remove the menu items for the portfolio application to understand the process of the admin toolbar.

Customizing the main navigation menu

In WordPress, the main navigation menu is located on the left of the screen where we have access to all the sections of the application. Similar to the admin toolbar, we have the ability to extend the main navigation menu with customized versions. Let's start by adding the admin-menu-invoking action to the constructor:

```
add_action( 'admin_menu', array(
  $this,'wpwa_customize_main_navigation' ) );
```

Now, consider the initial implementation of the `wpwa_customize_main_navigation` function:

```
public function wpwa_customize_main_navigation(){
  global $menu,$submenu;
  echo "<pre>";print_r($menu);exit;
}
```

The preceding code uses the global variable `menu` for accessing the available main navigation menu items. Before we begin the customizations, it's important to get used to the structure of the menu array using a `print_r` statement. A part of the output generated from the `print_r` statement is shown in the following section:

```
Array
(
  [2] => Array
  (
```

```
        [0] => Dashboard
        [1] => read
        [2] => index.php
        [3] =>
        [4] => menu-top menu-top-first menu-icon-dashboard
        [5] => menu-dashboard
        [6] => none
    )

    [4] => Array
    (
        [0] =>
        [1] => read
        [2] => separator1
        [3] =>
        [4] => wp-menu-separator
    )
)
```

The structure of the menu array seems to be different compared to the admin toolbar items array. Here, we have array indexes instead of unique keys, and hence any altering of the menu will be done using index values.

Up until this point, we have used the existing WordPress features for the functionality of the portfolio management application, and hence the main navigation menu is constructed based on user roles and permissions. Therefore, we don't need to alter the menu at this point. However, we are going to see how menu items can be added and removed to cater for advanced requirements in the future. Let's get started by removing the dashboard menu using the following code:

```
public function wpwa_customize_main_navigation() {
  global $menu, $submenu;
  unset($menu[2]);
}
```

We can use the `unset` function to remove items from the menu array. Now your Dashboard menu item will be removed from the menu. Similarly, we can use the global submenu variable to remove the submenus when needed.

As of WordPress 3.1, we can use the `remove_menu_page` and `remove_submenu_page` functions to remove the existing menu items. I suggest you try the preceding method to get an understanding of the menu slugs and links before moving on to these functions.

The following code contains the functionality for removing the Dashboard menu with the latest technique:

```
remove_menu_page('index.php');
```

Creating new menu items

The latest versions of WordPress use `add_menu_page` or `add_sub_menu_page` to create custom menu pages. In the preceding section, we removed items from the existing menu. Adding new menu items is not as simple as removing. We have to provide a functionality and a display code for the menu page while adding them to the menu. The implementation of `add_menu_page` will be discussed in the next section on the settings page.

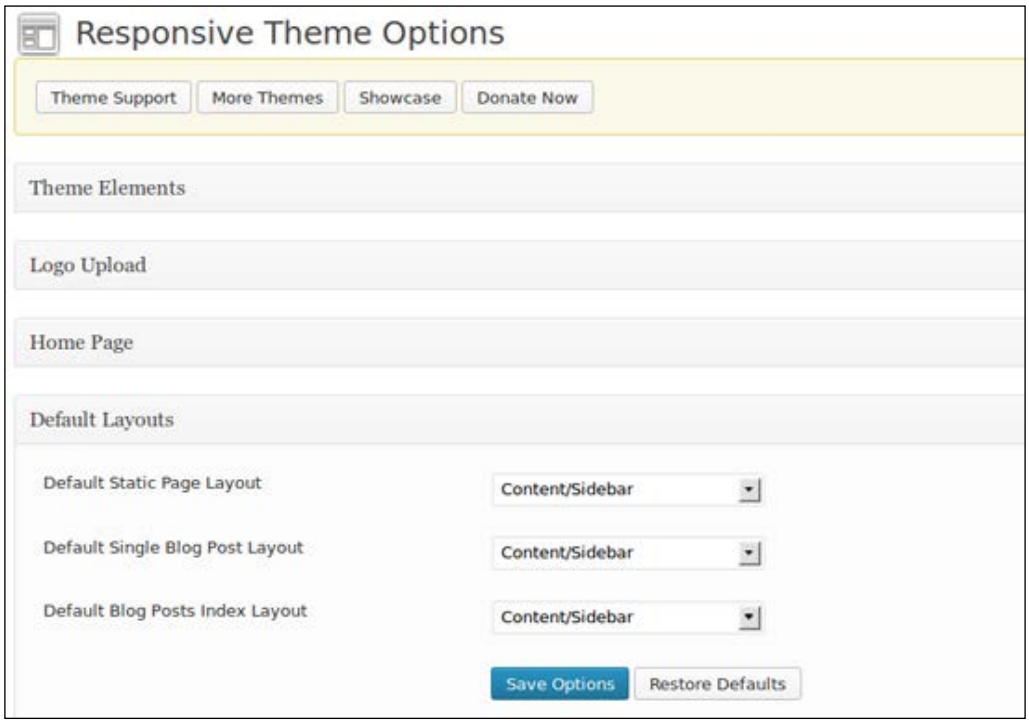
Adding features with custom pages

WordPress was originally created as a blogging platform and evolved into a content management system. Hence, most of the core functionalities are implemented on the concept of posts and pages. In web applications, we need to go way beyond these basic posts and pages to build quality applications. Custom menu pages play a vital role in implementing custom functionalities within the WordPress admin dashboard. Let's consider the various types of custom pages in the default context:

- **Options pages**: These are used to manage the options of the application. Even though the option pages are generally used for theme options, we can manage any type of application-specific settings with these pages.
- **Custom menu pages**: Generally, these pages are blank by default. We need to implement the interface as well as the implementation for catering to custom requirements that can't take advantage of WordPress' core features.

Building options pages

Options pages are implemented in each and every WordPress theme by default. The design and available options may vary based on the quality and features of the theme. We selected a theme called **Responsive** for the purpose of this book. So, let's take a look at the default theme options panel of **Responsive Theme** using the following screenshot:



The responsive theme uses its own layout structure for the options page. In addition to the theme settings, these pages are used as settings pages for plugins. Compared to generic websites, the applications focus mainly on its functionality and give lesser priority to its design. Therefore, we are going to create an application settings page to use the WordPress core options.

By default, WordPress offers functions called `add_menu_page` and `add_sub_menu_page` to manually create such pages. But there are a number of frameworks for automating the process of creating options pages, and hence we are going to take advantage of one of these existing frameworks.

It's a good practice to automate common application tasks through frameworks and libraries to focus more on application-specific logic. But you have to keep in mind that not all WordPress plugins are properly tested and maintained by experts in the field.

SMOF (**Slightly Modded Options Framework**) is one of the standout plugins in building WordPress options pages, and hence we are going to use it for the settings page of the portfolio management application.

Automating option pages with SMOF

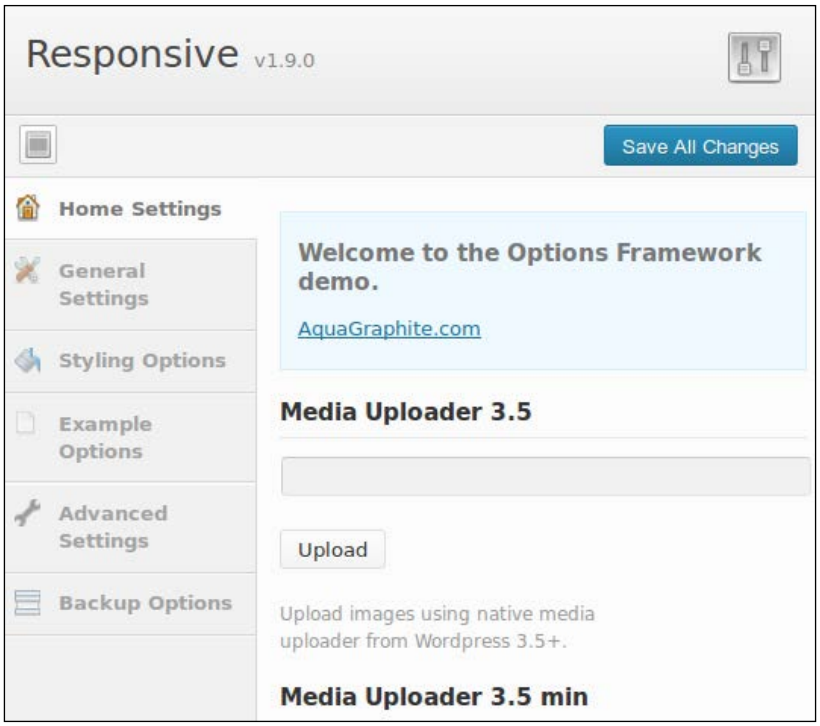
First, I would like to suggest that you take a look at professional WordPress themes in popular marketplaces. SMOF seems to be one of the favorites among professional theme developers. This framework is designed to build the theme options pages. Here, we are going to see how we can use the theme settings by creating the custom settings page for the application. Let's get started by downloading a copy of the plugin from GitHub at `https://github.com/syamilmj/Options-Framework`.

Once the downloaded ZIP file is extracted, you will find two folders called `admin` and `images`. Copy those two folders to the root of the Responsive theme located inside the `/wp-content/themes/Responsive` folder. If you already have an `images` folder, copy the contents of the SMOF image folder to your original `images` folder.

Then, open the `functions.php` file of the Responsive theme and add the following line to initialize the SMOF framework:

```
require_once ('admin/index.php');
```

Now, navigate to the **Appearance** menu of the dashboard and you will be able to see another tab called **Theme Options** right under the original **Theme Options** tab of the Responsive theme. Click on the second **Theme Options** tab and you will get the default SMOF configuration screen as follows:



This is the default behavior of any options framework for configuring theme settings. Since we already have a theme options panel, we can use the SMOF framework to create a site settings panel, which would be accessed directly from the main menu instead of from inside the **Appearance** section.

By default, the SMOF framework generates various types of sample fields in its default view. As developers, we need to create our own fields and tabs by using the sample fields and tabs.

Customizing the options page to use as a generic settings page

First, we have to move the menu item from the **Appearance** section into the main navigation menu. The SMOF framework uses the `functions.interface.php` file located in the `admin/functions` folder to define the theme page. Open the file in your text editor and you will see a function called `optionsframework_add_admin` with the following source code:

```
function optionsframework_add_admin() {
  $of_page = add_theme_page( THEMENAME, 'Theme Options',
'edit_theme_options', 'optionsframework',
'optionsframework_options_page');
  // Add framework functionaily to the head individually
  add_action("admin_print_scripts-$of_page", 'of_load_only');
  add_action("admin_print_styles-$of_page",'of_style_only');
}
```

The preceding code uses the `add_theme_page` function to add the page to the menu. Since we want direct access to the menu, this should be converted into an admin menu page instead of a theme page. So, let's consider the following implementation of the modified function:

```
function optionsframework_add_admin() {
  $of_page = add_menu_page( 'Protfilo App Settings', 'Protfilo App
Settings', 'edit_theme_options', 'optionsframework',
'optionsframework_options_page');
  // Add framework functionaily to the head individually
  add_action("admin_print_scripts-$of_page", 'of_load_only');
  add_action("admin_print_styles-$of_page",'of_style_only');
}
```

Now you will be able to see the menu item on the left menu instead of the **Appearance** section. Also, notice that we have used **Portfolio App Settings** as the title of the menu and the settings page to make things clear. Having completed the default configurations, we can now move to creating application-specific options.

Building the application options panel

As I mentioned earlier, the default options panel contains dozens of sample configurations and fields. We have to get rid of the existing ones before defining application-specific tabs and fields.

The SMOF framework uses the `functions.options.php` file located in the `admin/functions` folder to configure the structure and fields of the options panel. Open the file and you will be able to find a bunch of arrays with configurations under the comment `The Options Array`. Now, we have to remove or comment all the elements of the `$of_options` array to have the following code:

```
/*-----------------------------------------------------------------------
---------------*/
/* The Options Array */
/*-----------------------------------------------------------------------
---------------*/
// Set the Options Array
global $of_options;
$of_options = array();
```

Once completed, we can start defining the options required for our application. Here, we are going to create two main sections called `Subscription Settings` and `Frontend Widget Settings`, containing one option field each. So, add the following code after the preceding code to define the main tabs for the options panel:

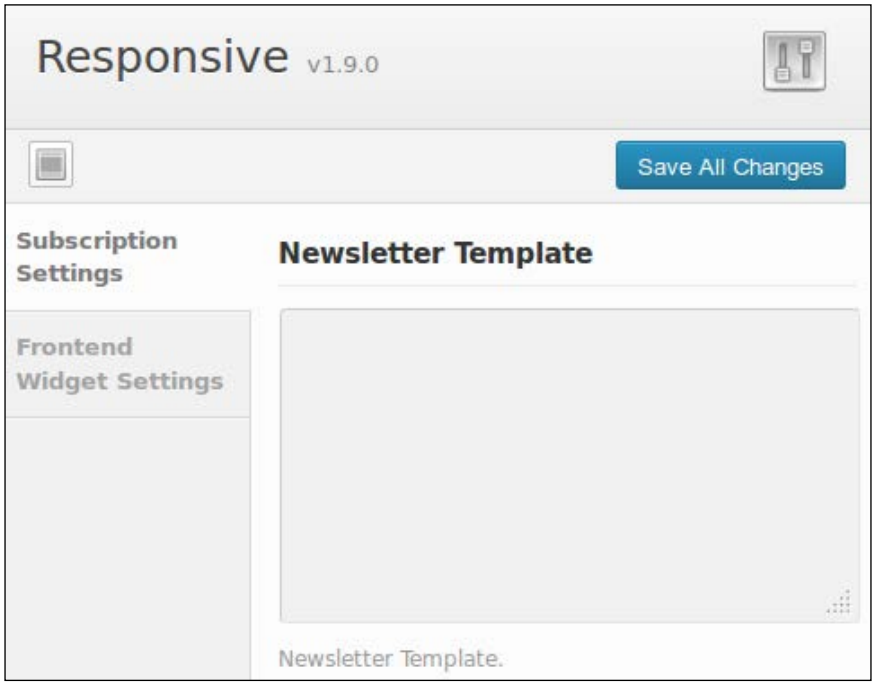
```
$of_options[] = array( "name" => "Subscription Settings",
"type" => "heading"
);
$of_options[] = array( "name" => "Frontend Widget Settings",
"type" => "heading"
);
```

Now you will be able to see two blank main tabs. We need to use `type` as the heading to define the main tabs for the panel and `name` as the key to define the display text. The next task is to add the configuration fields into these tabs. For this example, we are going to configure the newsletter template for `Subscription Settings` and the maximum number of items for `Frontend Widgets`. Consider the modified implementation of the preceding code:

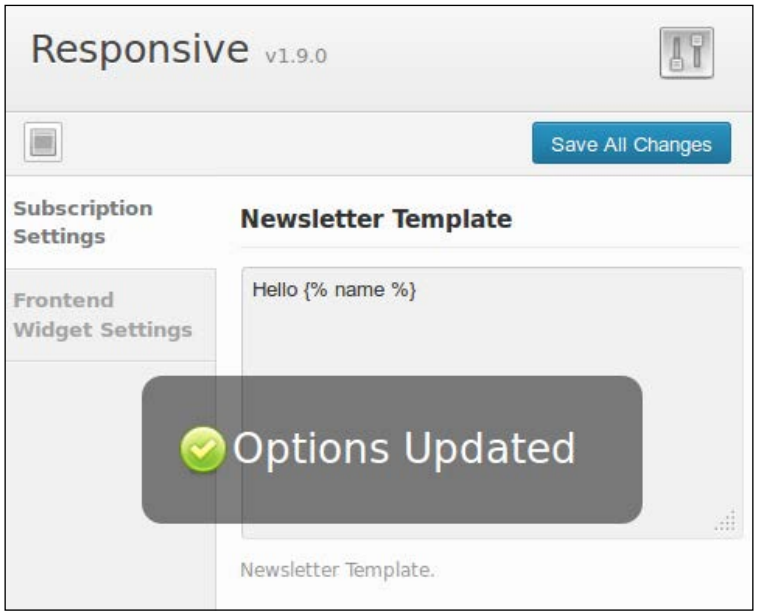
```
$of_options[] = array( "name" => "Subscription Settings",
  "type" => "heading"
);
$of_options[] = array(
  "name" => "Newsletter Template",
  "desc" => "Newsletter Template.",
```

```
    "id"    => "newsletter_temp",
    "std"   => "",
    "type" => "textarea"
  );
  $of_options[] = array(
    "name"=> "Frontend Widget Settings",
    "type" => "heading"
  );
  $of_options[] = array(
    "name" => "Number of records in lists",
    "desc" => "Number of records in lists",
    "id"    => "max_list_no",
    "std"   => "5",
    "type" => "text"
  );
```

Now we have two main tabs and two fields for the panel. Each of the fields related to a specific tab should be defined after the definition of the tab. Hence, all the fields should be defined after the respective heading element. Once the preceding code is used, we can see the screen with options fields as given in the following screenshot:



We can edit the option values and click on the **Save All Changes** button to save the data into the `wp_options` tables with unique keys, as shown in the following screenshot:



Using the SMOF framework to create options panels saves a considerable amount of time and hence should be used whenever possible. In web applications, we need custom forms rather than site-specific settings. But unfortunately, we can't use this framework to save the data to custom tables without altering its implementation. Here, we have a very basic application options panel which we will be updating with additional fields when necessary throughout the remaining chapters.

Using the WordPress Options API

These options frameworks save time and provide useful out of the box features. But it's important to know how to use the WordPress Options API in situations where you don't want to rely on third-party plugins or themes. We have to use the `wp_options` table for storing custom options for our plugins and themes. WordPress provides a set of built-in functions to work with the `wp_options` tables. Let's look at the most commonly used functions of the WordPress Options API:

- `add_option`: This is used to save new option/value pairs to the database. It doesn't do anything if the option already exists in the database.
- `delete_option`: This is used to remove the existing option/value pairs from the database. It returns `TRUE` when the option is deleted successfully.

- `get_option`: This is used to retrieve the option/value pairs from the database. It returns `FALSE` if the option doesn't exist in the database.
- `update_option`: This is used to update the option/value pairs in the database. First, it checks for the existence of the option and then updates it accordingly. In case the option doesn't exist, it creates a new option/value pair in the database.

Be sure to use these functions whenever you need to work with the `wp_options` table instead of writing your own queries. These functions come with built-in filters and validations and are hence considered the safest ways of working with the `wp_options` table. You can look at the complete WordPress Options API at `http://codex.wordpress.org/Options_API`.

Now, assume that we want to build an admin options panel instead of using a third-party plugin such as SMOF. In such cases, we first have to define an admin settings page or menu page as shown in the following code:

```
add_action('admin_menu', 'wpwa_options_menu');
function wpwa_options_menu() {
  add_menu_page('WPWA Options', 'WPWA Options', 'administrator',
__FILE__, 'wpwa_options_page');
  add_action( 'admin_init', 'wpwa_register_settings' );
}
```

Then, we have to define the options of our page using the `register_setting` function provided by WordPress. Let's consider the implementation of the `wpwa_register_settings` function:

```
function wpwa_register_settings() {
  register_setting( 'wpwa-settings-group', 'option1' );
  register_setting( 'wpwa-settings-group', 'option2' );
}
```

Here, we have two fields in the options panel called `option1` and `option2`. We can define them inside a single group with the `register_setting` function. Next, we can move into the HTML implementation of the form using the following code:

```
<?php
function wpwa_options_page() {
?>
  <div class="wrap">
    <form method="post" action="options.php">
    <?php settings_fields( 'wpwa-settings-group' ); ?>
    <table class="form-table">
      <tr valign="top">
```

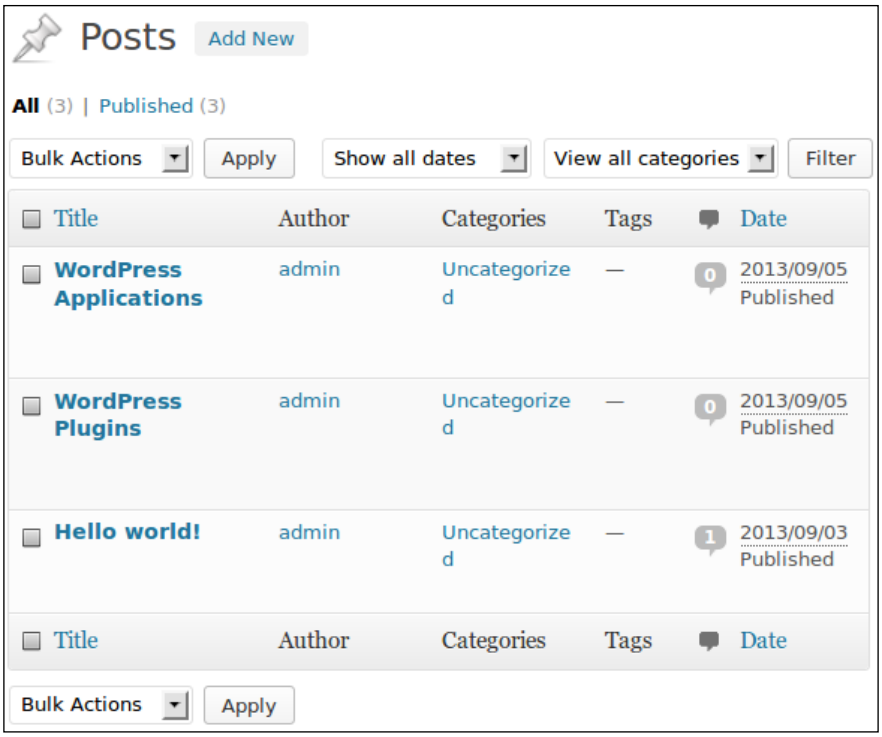
```
          <th scope="row">Option1</th>
          <td><input type="text" name="option1" value="<?php echo
get_option('option1'); ?>" /></td>
        </tr>
        <tr valign="top">
          <th scope="row">Option2</th>
          <td><input type="text" name="option2" value="<?php echo
get_option('option2'); ?>" /></td>
        </tr>
      </table>
      <?php submit_button(); ?>
      </form>
    </div>
<?php } ?>
```

It's important to define the form action as `options.php` to get the default functionality provided by WordPress. Then, we pass the previously defined options group name to the `settings_fields` function. This function will generate a set of hidden variables needed to save the options. Next, we define the existing values of the two options by using the `get_option` function. We have to make sure that we use the same name for the field as well as the `register_setting` function. Finally, we call the `submit_button` function to generate the submit button.

Once the form is submitted, WordPress will look for the field names that match the settings registered through the `register_setting` function. Then, it will automatically save the data into the `wp_options` table. This process is quite useful in scenarios where you don't want to rely on third-party plugins for the creation of options panels. Be sure to test both the techniques to identify the pros and cons.

Using feature-packed admin list tables

In web applications, you will find a heavy usage of CRUD operations. Therefore, we need tables to display the list of records. These days, developers have the choice of implementing common lists using client-side JavaScript as well as PHP. These lists contain functionalities such as pagination, selections, sorting, and so on. Building these types of lists from scratch is not recommended unless you are planning to build a common library. WordPress offers a feature-packed list for its core features using the `WP_List_Table` class located in the `wp-admin/includes/wp-list-table.php` file. We have the ability to extend this class to create application-specific custom lists. First, we'll look at the default list used for core features, as shown in the following screenshot:



As you can see, most of the common tasks such as filtering, sorting, custom actions, searching, and pagination are built into this list, which is easily customized by creating child classes. In the next section, we are going to implement extended lists to cater for portfolio-application-specific requirements.

Building extended lists

The extended version of `WP_List_Table` can be created by manually overriding each and every function in the base class. But we are going to take a simpler approach by using an existing template to extend the lists. The WordPress plugin directory contains a useful plugin called `Custom List Table Example` for the reusable template of the `WP_List_Table` class. You can grab a copy of the plugin at `http://wordpress.org/plugins/custom-list-table-example/` and get used to the code before we get started.

Create a new plugin called `wpwa-list-table` and copy the `list-table-example.php` file from the downloaded plugin and rename it as `class-wpwa-list-table.php`. You can then change the plugin descriptions and information if necessary. Now we are ready to customize the template.

Using an admin list table for following developers

In requirements planning, we identified two roles called followers and developers, where followers can subscribe to the activities of the developers. There are several ways of implementing such requirements within WordPress. Here, we will be using a custom list table to manage the subscription process. Following is the list of identified tasks for this implementation:

- Developers should be listed for subscriptions
- Followers should be able to select multiple developers for subscriptions
- On executing the custom action, the selected developers should be saved in a custom database table with follower details

Let's get started.

We need to have the `wp_subscribed_developers` table to implement this feature. It's not yet available in our database. So, make sure you use the `wpwa-database-manager` plugin provided with the source codes of this chapter and reactivate the plugin to create the `wp_subscribed_developers` table.

Step 1 – defining the custom class

Change the name of the `TT_Example_List_Table` class to a new unique name. Here, we have used `WPWA_List_Table` as the class name.

Step 2 – defining instance variables

This template offered by the `Custom List Table Example` plugin uses hardcoded data in a variable called `$example_data`. In real web applications, we need to dynamically get this data from the database, file, or any external source. Therefore, set the `$example_data` variable to an empty array as shown in the following line of code:

```
var $example_data = array();
```

Step 3 – creating the initial configurations

We need to configure the necessary settings inside the `WPWA_List_Table` class constructor, as given in the following code:

```
function __construct() {
  global $status, $page;
```

```
    //Set parent defaults
    parent::__construct(array(
      'singular' => 'developer', //singular name of the listed
  records
      'plural' => 'developers', //plural name of the listed records
      'ajax' => false          //does this table support ajax?
    ));
  }
```

Inside the array of configurations, we have to define the singular and plural names for the records. This should be a unique name and should have no relation to database tables or columns. We can also define the support for AJAX, although this will not be discussed in this book.

Step 4 – implementing custom column handlers

In this step, we need to define methods for handling each of the columns to be displayed in the list. The developer list will contain a single column called `Developer Name`, and hence we need only the following function implementation:

```
  function column_developer_name($item) {
    //Return the developer name contents
    return sprintf('%1$s ',
      /* $1%s */ $item['developer_name']
    );
  }
```

Before explaining the code, I would like you to have a look at the structure of our final dataset using the following code:

```
  Array
  (
    [0] => Array
    (
    [ID] => 24
      [developer_name] => John Doe
    )
    [1] => Array
    (
      [ID] => 22
      [developer_name] => Mark
    )
  )
```

The preceding dataset is generated manually to contain custom keys. When we are using the direct database result for the dataset, these keys will be replaced by the database columns. Here, we are using a column called `developer_name`, which doesn't actually exist in the database. So, the `column_developer_name` function returns the contents of the `developer_name` key in the dataset.

Don't forget to create the `column_{column name}` functions for each and every column in your list, in case you decide to include multiple columns.

Step 5 – implementing column default handlers

In the previous step, we created column functions for the available columns in the list. In case you skip the definition of a specific function for a column, you should create a default fallback function called `column_default`, as shown in the following code:

```
function column_default($item, $column_name) {
  switch ($column_name) {
    case 'developer_name':
    return $item[$column_name];
    default:
    return print_r($item, true); //Show the whole array for
troubleshooting purposes
  }
}
```

Here, we need to define each and every column which will not be defined separately. Even though we have defined `developer_name`, it won't be used as we have a specific function called `column_developer_name`.

Step 6 – displaying the checkbox for records

Apart from the custom columns, we need to have a column with a checkbox for every record in the list. This checkbox will be used to select records and execute specific actions on the **Bulk Actions** drop-down menu. Let's consider the implementation using the `column_cb` function inside the template:

```
function column_cb($item) {
  return sprintf(
    <input type="checkbox" name="%1$s[]" value="%2$s" />',
    /* $1%s */ $this->_args['singular'], //Let's simply repurpose
```

```
the table's singular label ("movie")

    /* $2%s */ $item['ID'] //The value of the checkbox should be
the record's id
  );
}
```

The first parameter in the preceding statement uses the singular label we created inside the constructor to set the name of the checkbox as an array. The second parameter contains the ID for the row, as defined in our dataset. This value should be the ID of the record in the database table.

We can define any key for the ID in the dataset. But consistency is important in developing reusable stuff, and hence I prefer using `ID` for all the record IDs in each of the lists I create. You may decide your own key to be re-used across all custom lists.

Step 7 – listing the available custom columns

Now we need to define all the columns available to create the custom list by modifying the existing `get_columns` function, as illustrated in the following code:

```
function get_columns() {
  $columns = array(
    'cb' => '<input type="checkbox" />', //Render a checkbox
instead of text
    'developer_name' => 'Developer Name'
  );
  return $columns;
}
```

This is an in-built function that returns an array of columns. We don't need to change the details of the checkbox column as it's common to all the lists. Afterwards, we have to define all the custom columns using the column name as the key and the display name as the value.

Step 8 – defining the sortable columns of the list

This function is pretty straightforward like the previous one, where we defined the columns to be sortable. Consider the following modified implementation of the `get_sortable_columns` function for our requirements:

```
function get_sortable_columns() {
  $sortable_columns = array(
```

```
        'developer_name' => array('developer_name', false)
    );
    return $sortable_columns;
}
```

Here, we have only a single entry based on our requirements. You can add all the available columns for custom lists. The key of the array item contains the column name and the value contains the database column. Since we will be using a manually created dataset from the database, the key and value will be the same.

Step 9 – creating a list of bulk actions

In the default post list, we can see different options such as **Edit** and **Move to Trash** inside the **Bulk Actions** drop-down box. Similarly, we can include custom actions in custom lists. This is one of the most powerful features of this list, and it is useful when implementing complex requirements in web applications. Consider the following modified implementation of the `get_bulk_actions` function:

```
function get_bulk_actions() {
    $actions = array(
        'follow' => 'Follow'
    );
    return $actions;
}
```

The preceding function is pre-built and returns the list of actions to be included in the dropdown. In this scenario, we need followers to subscribe to developer activities. Hence, we use a custom action called `follow`.

Step 10 – retrieving list data

We have completed the configuration part of the list and now we are moving on to the exciting part of adding real data and executing actions. The default template contains a function called `prepare_items` to set the data required for the custom table. We can include the necessary SQL queries inside this function to generate data. But I prefer keeping the function in its default state and providing the data through the `example_data` instance variable.

You can use the extensive code comments of this function to understand the functionality of each section and make the customizations when necessary.

Step 11 – adding the custom list as a menu page

Having created the list, we need a specific location to access this list as it's not available in any of the navigation menus. So, we are going to include the list into the left navigation menu as an admin menu page. The following code should be placed in `class-wpwa-list-table.php` after the `WPWA_List_Table` class:

```
function wpwa_followers_menu() {
  add_menu_page('Follow Developers', 'Follow  Developers',
'follow_developer_activities', 'wpwa_subscriptions',
'followers_list_page');
}
add_action('admin_menu', 'wpwa_followers_menu');
```

A new menu page is created on the `admin_menu` action using the `add_menu_page` function. Only users with the user role `follow_developer_activities` will have access to this function, since we have specified the capability on `add_menu_page`.

This functionality can be provided for multiple user roles by creating a new common capability for the preferred user roles.

Finally, we have defined the callback function as `followers_list_page` to generate the HTML contents for the list.

Step 12 – displaying the generated list

First, we have to set the database results to the list table. So, consider the initial part of the `followers_list_page` function for querying the database, as shown in the following code:

```
function followers_list_page() {
  //Create an instance of our package class...
  $testListTable = new WPWA_List_Table();
  $user_query = new WP_User_Query(array('role' => 'developer'));
  foreach ($user_query->results as $developer) {
    array_push($testListTable->example_data, array("ID" =>
$developer->data->ID, "developer_name" => $developer->data-
>display_name));
  }
  //Fetch, prepare, sort, and filter our data...
  $testListTable->prepare_items();
}
```

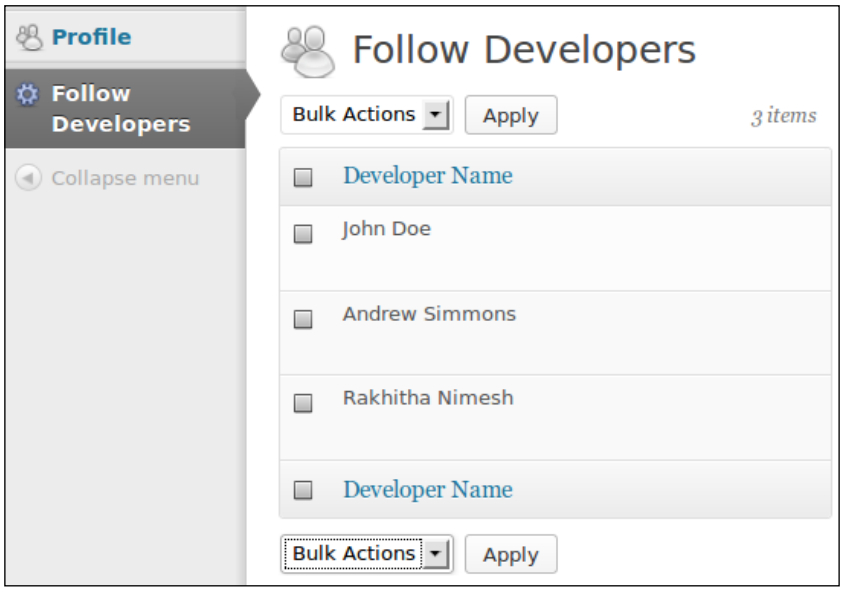
We can begin the implementation by initializing an object of the `WPWA_List_Table` class. Then, we execute a WordPress query using the built-in `WP_User_Query` class to retrieve the list of users with the role of a developer. We have chosen to manually create the dataset by traversing through the database results and assigning it to the dataset structure defined earlier. Keep in mind that we are passing the dataset to the `WPWA_List_Table` class by using an instance variable called `$example_data`. Finally, we call the `prepare_items` function to get the data ready with features such as sorting, paginations, and so on.

Having completed the explanations on the initial part, we can move on to the HTML-generation part of the `followers_list_page` function, as illustrated in the following code:

```
<div class="wrap">
  <div id="icon-users" class="icon32"><br/></div>
  <h2>Follow Developers</h2>
  <!-- Forms are NOT created automatically, so you need to wrap
the table in one to use features like bulk actions -->
  <form id="movies-filter" method="POST">
    <!-- For plugins, we also need to ensure that the form posts
back to our current page -->
    <input type="hidden" name="page" value="<?php echo
$_REQUEST['page'] ?>" />
    <!-- Now we can render the completed list table -->
    <?php $testListTable->display() ?>
  </form>
</div>
```

Here, we have a basic HTML form and the necessary heading and labels. The actual list generation is done through the display function of `WPWA_List_Table`. This function is available on the `WP_List_Table` class and hasn't been overridden on the template class. Hence, a call to display will use the function in the parent class. You can also override the display function on the child class to provide different behaviors to the default design.

Now, your custom list should look like something similar to the following screenshot:



Even though we have completed the custom list implementation, the list doesn't have any functionality until we implement the custom action to allow the followers to subscribe to developers. Let's move back to the `process_bulk_action` function of the `WPWA_List_Table` class, as shown in the following code:

```
function process_bulk_action() {
  global $wpdb;
  //Detect when a bulk action is being triggered...
  if ('follow' === $this->current_action()) {
    $developers = $_POST['developer'];
    $user_ID = get_current_user_id();
    foreach ($developers as $developer) {
      $wpdb->insert(
        $wpdb->prefix . "subscribed_developers",
        array(
          'developer_id' => $developer,
          'follower_id' => $user_ID
        ),
        array(
          '%d',
          '%d'
        )
      );
```

```
        }
        $msg = "Succefully completed.<a href='" . admin_url() .
    "?page=wpwa_subscriptions'>
        Follow More Developers</a>";
        wp_die($msg);
      }
    }
```

The preceding function is used to execute all the actions defined in the **Bulk Actions** dropdown. Since we have one action, using an `if` statement on `current_action` seems appropriate. In scenarios where you have multiple actions, switch statements will be ideal over `if-else` statements.

The followers have to tick the checkboxes of the developers they wish to follow. Then, they can select the follow action and click on the **Apply** button to execute the action. Once the button is clicked, we can get the selected developer IDs as an array using `$_POST['developer']`. Also, we can get the ID of the follower using the `get_current_user_id` function.

In *Chapter 3*, *Planning and Customizing the Core Database*, we created a custom table called `subscribed_developers` to be used for developer-subscription management. Now, we need to insert records to this table using a custom query as shown in the preceding code. Finally, we display the message on the same page with a link back to the developers' list.

Now we have a fully featured custom list with all the basic grid functionalities. We can create as many lists as possible by creating new templates or creating a reusable class for this library. It's for the future, but for now, you can test the list by following developers.

An awesome visual presentation for the admin dashboard

In general, users who visit websites or applications don't understand the technical aspects. Such users evaluate systems based on the user friendliness, simplicity, and richness of the interface. Hence, we need to think about the design of the admin pages. Most WordPress clients don't prefer the default interface as it is seen commonly by the users. This is where admin themes become handy in providing application-specific designs. Even with admin themes, we cannot change the structure as it affects the core functionality. But we can provide eye-catching interfaces by changing the default styles of the admin theme.

Building a complete admin theme is a time-consuming task beyond the scope of this chapter, as we need to define custom styles for all the existing CSS selectors. Therefore, we are going to provide a head start to the admin theme design by altering the main navigation menu. Let's start by creating another plugin called `wpwa-admin-theme` and the main file as `class-wpwa-admin-theme.php`. The initial definition of the plugin will not be discussed as we have already done it several times in the previous chapters.

Let's start by defining the stylesheet for the admin theme using the following plugin code:

```
<?php
class WPWA_Admin_Theme {
  public function __construct() {
    add_action('admin_enqueue_scripts', array($this,
'wpwa_admin_theme_style'));
    add_action('login_enqueue_scripts', array($this,
'wpwa_admin_theme_style'));
  }
public function wpwa_admin_theme_style() {
    wp_enqueue_style('my-admin-theme', plugins_url('css/wp-
admin.css', __FILE__));
  }
}
$admin_theme = new WPWA_Admin_Theme();
```

We begin the implementation by defining the necessary actions for including the CSS file. Usually, we use `admin_enqueue_scripts` to include scripts and styles in the admin area. The `login_enqueue_scripts` action is used to enable styles on the login screen. You can omit the `login_enqueue_scripts` action if you are not intending to customize the login screen.

Then, we add the CSS file specific to the plugin using the `wp_enqueue_style` function. That's all we need to implement in order to create admin themes. The rest of the designing stuff will be handled through the CSS file. So, make sure you create a new CSS file called `wp-admin.css` inside a folder called `CSS`.

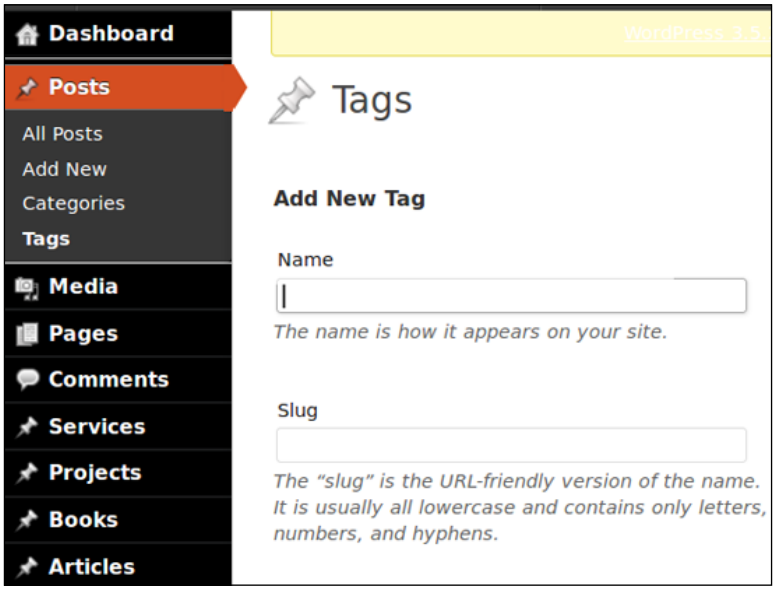
A CSS file used for the admin theme is loaded after the default WordPress admin stylesheets. Therefore, it will override the existing styles provided by the core.

In this section, we will be styling the main navigation menu of WordPress. You can update the CSS file with menu-specific styles as illustrated in the following code:

```
#adminmenuback,#adminmenuwrap { background: #000; }
#adminmenu a{ color : #FFF; }
#adminmenu a.menu-top, #adminmenu .wp-submenu .wp-submenu-head {
  border-bottom-color: #191A1B;
  border-top-color: #191A1B;
}
#adminmenu .wp-submenu, .folded #adminmenu a.wp-has-current-
submenu:focus + .wp-submenu, .folded #adminmenu .wp-has-current-
submenu .wp-submenu {
  background-color: #363636;
}
#adminmenu li.wp-menu-separator {
  background: none repeat scroll 0 0 #DFDFDF;
  border-color: #454545;
}
#adminmenu div.separator { background:#000; }
#adminmenu li.wp-menu-separator {
  background: none repeat scroll 0 0 #000;
  border-color: #000;
}
#adminmenu .wp-submenu li.current, #adminmenu .wp-submenu
li.current a, #adminmenu .wp-submenu li.current a:hover {
  color: #FFFFFF;
}
#adminmenu .wp-submenu a:hover,
#adminmenu .wp-submenu a:focus {
  background-color: #d54e21;
  color: #fff;
}
#adminmenu li.menu-top:hover,#adminmenu li.opensub > a.menu-top,
#adminmenu li > a.menu-top:focus {
  background-color: #d54e21;
  color:#fff;
  font-weight:bold;
}
#adminmenu li.wp-has-current-submenu a.wp-has-current-submenu,
#adminmenu li.current a.menu-top, .folded #adminmenu li.wp-has-
current-submenu, .folded #adminmenu li.current.menu-top, #adminmenu
.wp-menu-arrow, #adminmenu .wp-has-current-submenu .wp-
submenu .wp-submenu-head {
  background: #d54e21 !important;
}
```

```
#adminmenu .wp-menu-arrow div { background:#d54e21 !important; }
a, #adminmenu a, #the-comment-list p.comment-author strong a,
#media-upload a.del-link, #media-items a.delete, #media-items
a.delete-permanently, .plugins a.delete, .ui-tabs-nav a {
  color: #FFFFFF;
}
```

Now you can preview the navigation menu of the admin section using the screen shown in the following screenshot:



Customizing the menu was a very simple task, and now we have a slightly different interface with a different color scheme. Similarly, we have to define the styles for all the available themes to make a complete admin theme.

The main style file of WordPress is the `wp-admin.css` file located in the `wp-admin/css` directory. I suggest you have a look at this file to understand the styles of the various components in WordPress. Fortunately, this file is well-commented into sections for easier identification. The following code shows the main components inside the `wp-admin.css` file:

```
TABLE OF CONTENTS:
------------------
 1.0 - Text Elements
 2.0 - Forms
 3.0 - Actions
 4.0 - Notifications
```

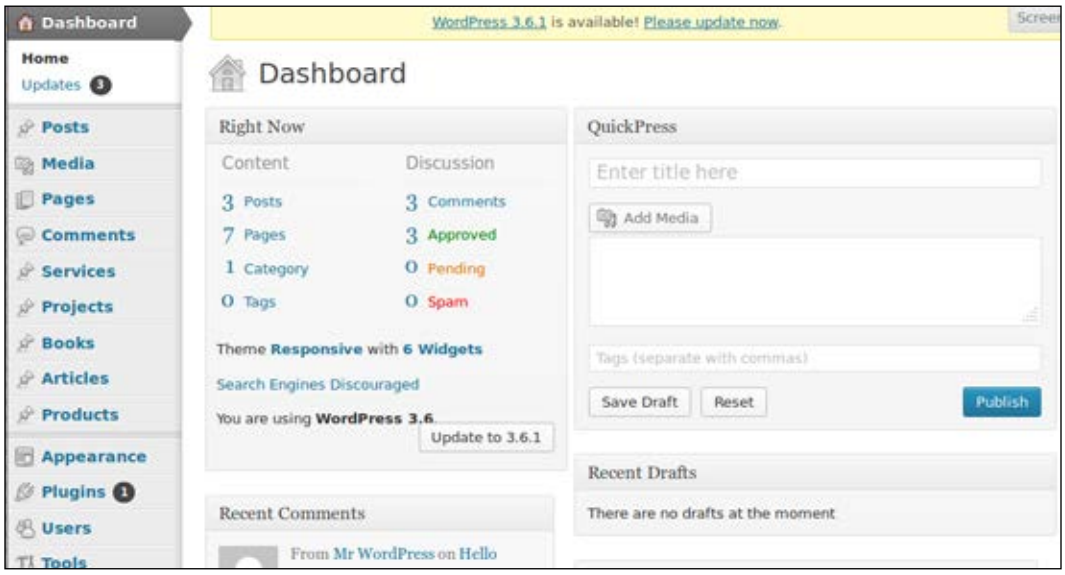
```
 5.0 - TinyMCE
 6.0 - Admin Header
    6.1 - Screen Options Tabs
    6.2 - Help Menu
 7.0 - Main Navigation
 8.0 - Layout Blocks
 9.0 - Dashboard
10.0 - List Posts
    10.1 - Inline Editing
11.0 - Write/Edit Post Screen
    11.1 - Custom Fields
    11.2 - Post Revisions
    11.3 - Featured Images
12.0 - Categories
13.0 - Tags
14.0 - Media Screen
    14.1 - Media Library
    14.2 - Image Editor
15.0 - Comments Screen
16.0 - Themes
    16.1 - Custom Header
    16.2 - Custom Background
    16.3 - Tabbed Admin Screen Interface
17.0 - Plugins
18.0 - Users
19.0 - Tools
20.0 - Settings
21.0 - Admin Footer
22.0 - About Pages
23.0 - Full Overlay w/ Sidebar
24.0 - Customize Loader
25.0 - Misc
```

Apart from the `wp-admin.css` file, WordPress uses the `colors-fresh.min.css` file located in the `/wp-admin/css` folder to style certain elements. We can override the styles of both these stylesheets using the custom CSS file created with the plugin. Use the TOC provided in the `wp-admin.css` file to style the remaining components and build a complete admin theme.

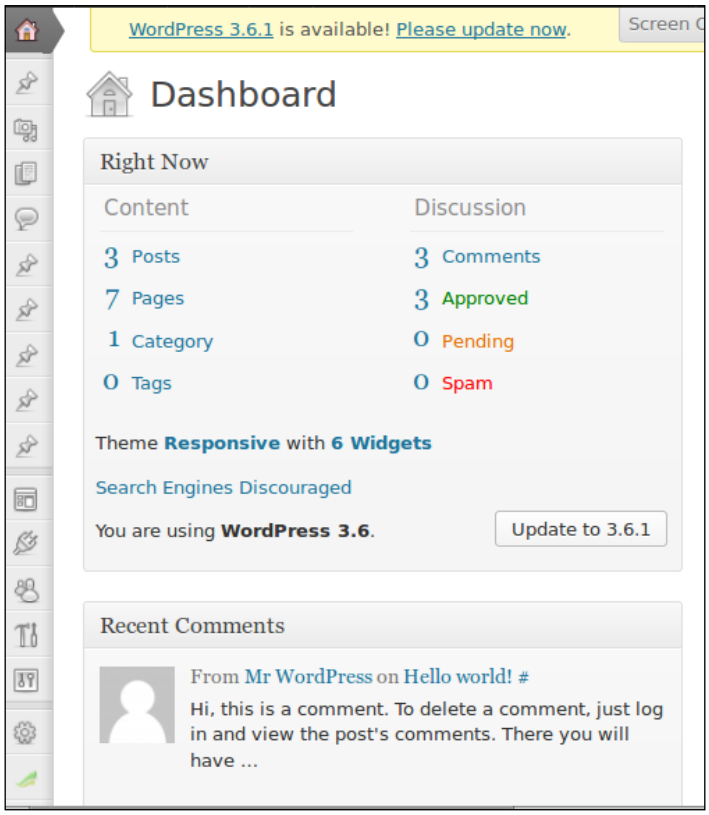
The responsive nature of the admin dashboard

Responsive design has become one of the major trends in web application development with the increase in the usage of mobile-based devices. Responsive applications are built using stylesheets that adapt to various screen resolutions with the help of media queries. Fortunately, the WordPress admin dashboard is responsive by default, and hence we can make responsive backends without major implementations.

Let's consider the following screenshot of the admin dashboard in its default resolution to understand the responsive nature:



Now, let's preview the mobile version of the same screen using the following screenshot:



With the low screen resolution, the theme has made adjustments to keep the responsiveness by minimizing the main navigation menu and increasing the size of dashboard widgets. This is an example of the responsive nature of the WordPress admin section. Try other screens to get an idea of how elements are adjusted to keep the responsiveness.

Since the admin section is responsive by default, we don't have to do anything else to make it responsive. But keep in mind that the plugins we create and use will not be responsive by default. Hence, it's important to design your plugin screens using percentage dimensions to keep the responsive nature.

Time for action

In this chapter, we covered the basics of the admin-dashboard-related functionality to be compatible with web applications. Now it's time for you to take these things beyond the basics by implementing the following actions:

- We created a default type of admin list table to allow subscriptions. Now, try to include an AJAX-based star rating system to allow followers to rate developers by implementing a custom column in the existing list.
- The SMOF framework was used to set up the application settings page and it was limited to handling options in the `wp-options` table. Try to create a new admin menu page to create dynamic forms that allow users to submit content to other application-specific tables.
- We started the implementation of the custom admin theme by setting up styles for the left navigation menu. Try to complete the theme by styling the remaining components.

Summary

Throughout this chapter, we looked at some of the exciting features of the WordPress admin dashboard and how we can customize them to suit complex applications. We started by customizing the admin toolbar and main navigation menu for different types of users. Most of the access permissions to the menu were provided through user capabilities, and hence we didn't need the manual permission checking while building the menu.

Typical web applications contain a large number of dynamic forms and lists to manage and display the application data. Therefore, we looked at the SMOF options framework for managing the AJAX-based form submissions. Also, we extended the existing WordPress admin list tables to cater to the custom functionality beyond the core implementation.

Finally, we looked at the importance of a responsive web design and how the WordPress admin dashboard adapts to responsive layouts while showing a glimpse of the WordPress admin theme design.

In *Chapter 7*, *Adjusting Themes for Amazing Frontends*, we are going to explore how we can manage the existing WordPress themes to build complex web application layouts using modern techniques.

7
Adjusting Themes for Amazing Frontends

Generally, users who visit web applications don't have any clue about the functionality, accuracy, or quality of the code of the application. Instead, they decide the value of the application based on user interfaces and the simplicity of using its features. Most expert-level web developers tend to give more focus to development tasks in complex applications. However, the application design plays a vital role in building the initial user base. WordPress uses themes which allow you to create the frontend of web applications with highly extendable features which go beyond conventional layout designs. Developers and designers should have the capability to turn default WordPress themes into amazing frontends for web applications.

In this chapter, we will be focusing on the extendable capabilities of themes while exploring the roles of the main theme files for web applications. Widgetized layouts are essential for building flexible applications, and hence we will also be looking at the possibilities of integrating widgetized layouts with WordPress action hooks. It's important to have a very good knowledge of working with WordPress template files to understand the techniques discussed in this chapter. By the end of this chapter, you will be able to design highly customizable layouts to adapt future enhancements.

In this chapter, we will cover the following topics:

- Basic file structure of a WordPress theme
- Understanding the template execution hierarchy
- Web application layout creation techniques
- Use of template engines inside WordPress
- Building a portfolio application's home page
- Widgetizing application layouts

- Generating an application's frontend menu
- Creating pluggable and extendable templates
- Planning action hooks for layouts

Introduction to a WordPress application's frontend

WordPress powers its frontend with a concept called themes, which consist of a set of predefined template files to match the structure of the default website layouts. In contrast to web applications, a WordPress theme works in a unique way. In *Chapter 1*, *WordPress As a Web Application Framework*, we had a brief introduction to the role of a WordPress theme and the most common layout. Preparing a theme for web applications can be one of the more complicated tasks that is not discussed widely in the WordPress development community. Usually, web applications are associated with unique templates, which are entirely different from the default page-based nature of websites.

Basic file structure of a WordPress theme

As WordPress developers, you should have a fairly good idea about the default file structure of WordPress themes. Let's have a brief introduction of the default files before identifying their usage in web applications. Think about a typical web application layout where we have a common header, footer, and content area. In WordPress, the content area is mainly populated by pages or posts. The design and the content for pages are provided through the `page.php` template, while the content for posts is provided through one of the following templates:

- `index.php`
- `archive.php`
- `category.php`
- `single.php`

Basically, most of these post-related file types are developed to cater to the typical functionality in blogging systems, and hence can be omitted in the context of web applications. Since custom posts are widely used in application development, we need more focus on templates such as `single-{post_type}` and `archive-{post_type}` than `category.php`, `archive.php`, and `tag.php`.

Even though default themes contain a number of files for providing default features, only the `style.css` and `index.php` files are enough to implement a WordPress theme. Complex web application themes are possible with the standalone `index.php` file.

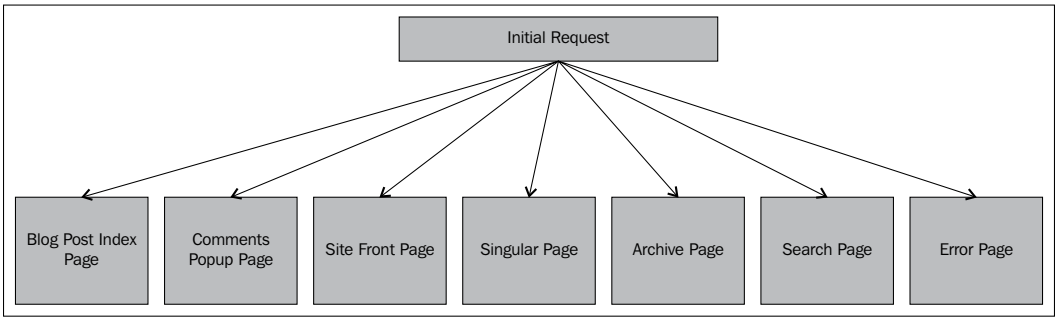
In normal circumstances, WordPress sites have a blog built on posts, and all the remaining content of the site is provided through pages. When referring to pages, the first thing that comes to our mind is the static content. But WordPress is a fully functional CMS, and hence the page content can be highly dynamic. Therefore, we can provide complex application screens by using various techniques on pages. Let's continue our exploration by understanding the theme file execution hierarchy.

Understanding template execution hierarchy

WordPress has quite an extensive template execution hierarchy compared to general web application frameworks. However, most of these templates will be of minor importance in the context of web applications. Here, we are going to illustrate the important template files in the context of web applications. The complete template execution hierarchy can be found at:

`http://codex.wordpress.org/images/1/18/Template_Hierarchy.png`

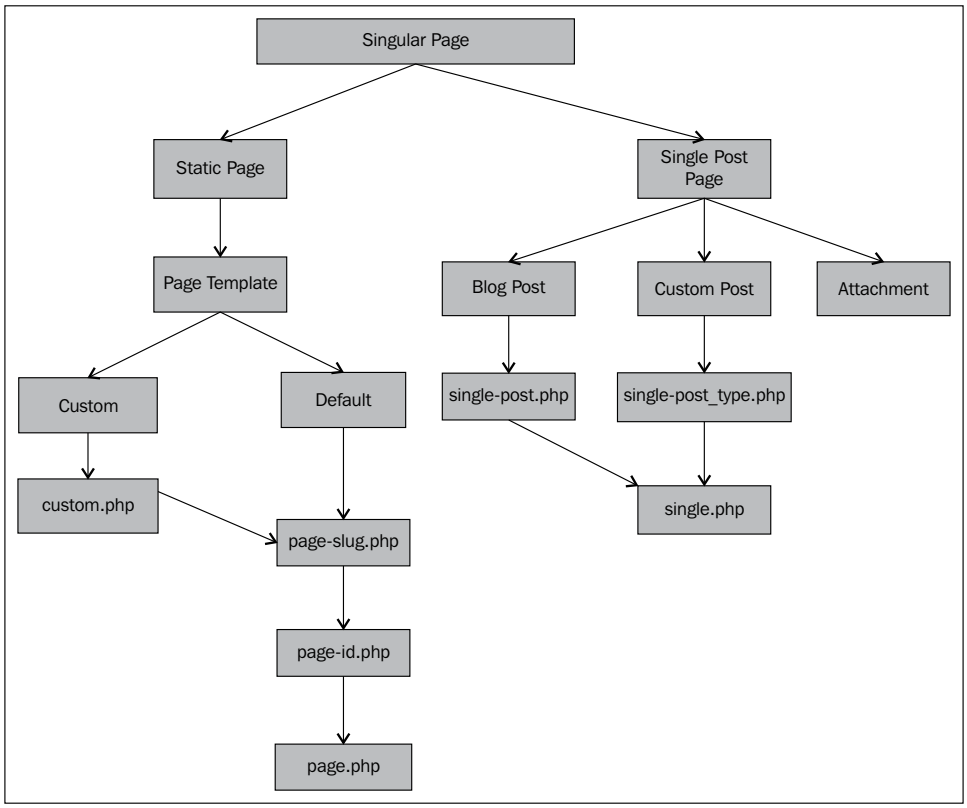
An example of the template execution hierarchy is as shown in the following diagram:



Once the **Initial Request** is made, WordPress looks for one of the main starting templates as illustrated in the preceding screenshot. It's obvious that most of the starting templates such as front page, comments popup, and index pages are specifically designed for content management systems. In the context of web applications, we need to put more focus into both singular and archive pages, as most of the functionality depends on top of those templates. Let's identify the functionality of the main template files in the context of web applications:

- **Archive pages**: These are used to provide summarized listings of data as a grid.
- **Single posts**: These are used to provide detailed information about existing data in the system.
- **Singular pages**: These are used for any type of dynamic content associated with the application. Generally, we can use pages for form submissions, dynamic data display, and custom layouts.

Let's dig deeper into the template execution hierarchy on the **Singular Page** path as illustrated in the following diagram:



Singular Page is divided into two paths that contain posts or pages. **Static Page** is defined as **Custom** or **Default** page templates. In general, we use **Default** page templates for loading website pages. WordPress looks for a page with the slug or ID before executing the default `page.php` file. In most scenarios, web application layouts will take the other route of **Custom** page templates where we create a unique template file inside the theme for each of the layouts and define it as a page template using code comments.

We can create a new custom page template by creating a new PHP file inside the theme folder and using the `Template Name` definition in code comments illustrated as follows:

```
<?php
/*
*  Template Name: My Custom Template
*/
?>
```

To the right of the preceding diagram, we have **Single Post Page**, which is divided into three paths called **Blog Post**, **Custom Post**, and **Attachment Post**. Both **Attachment Posts** and **Blog Posts** are designed for blogs and hence will not be used frequently in web applications. However, the **Custom Post** template will have a major impact on application layouts. As with **Static Page**, **Custom Post** looks for specific post type templates before looking for a default `single.php` file.

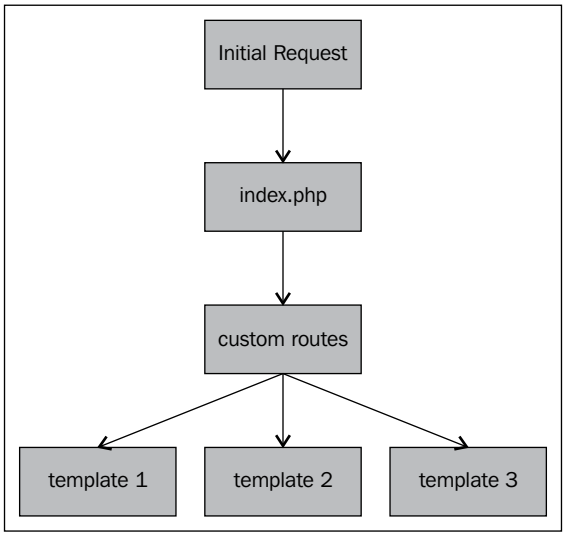
The execution hierarchy of an **Archive Page** is similar in nature to posts, as it looks for post-specific archive pages before reverting to the default `archive.php` file.

Now we have had a brief introduction to the template loading process used by WordPress. In the next section, we are going to look at the template loading process of a typical web development framework to identify the differences.

Template execution process of web application frameworks

Most stable web application frameworks use a flat and straightforward template execution process compared to the extensive process used by WordPress. These frameworks don't come with built-in templates, and hence each and every template will be generated from scratch.

Consider the following diagram of a typical template execution process:



In this process, **Initial Request** always comes to the `index.php` file, which is similar to the process used by WordPress or any other framework. It then looks for **custom routes** defined within the framework. It's possible to use **custom routes** within a WordPress context, even though it's not used generally for websites or blogs. Finally, **Initial Request** looks for the direct template file located in the templates section of the framework. As you can see, the process of a normal framework has very limited depth and specialized templates.

Keep in mind that `index.php` referred to in the preceding section is the file used as the main starting point of the application, not the template file. In WordPress, we have a specific template file named `index.php` located inside the `themes` folder as well.

Managing templates in a typical application framework is a relatively easy task when compared to the extensive template hierarchy used by WordPress. In web applications, it's ideal to keep the template hierarchy as flat as possible with specific templates targeted towards each and every screen.

In general, WordPress developers tend to add custom functionalities and features by using specific templates within the hierarchy. Having multiple templates for a single screen and identifying the order of execution can be a difficult task in large-scale applications, and hence should be avoided in every possible instance.

Web application layout creation techniques

As we move into developing web applications, the logic and screens will become complex, resulting in the need of custom templates beyond the conventional ones. There is a wide range of techniques for putting such functionality into the WordPress code. Each of these techniques have their own pros and cons. Choosing the appropriate technique is vital in avoiding potential bottlenecks in large-scale applications. Here is a list of techniques for creating dynamic content within WordPress applications:

- Static pages with shortcodes
- Page templates
- Custom templates with custom routing

Shortcodes and page templates

We discussed static pages with shortcodes and page templates in *Chapter 2, Implementing Membership Roles, Permissions, and Features*. The shortcodes technique should not be used in web applications due to the lack of control it displays within the source code. Even though page templates are not the best solution, we can use them to cater for advanced requirements in web applications.

In the preceding two techniques, the site admin has the capability of changing the structure and core functionality of an application through the dashboard by changing the database content. Usually, the site admin is someone who is capable of managing the site, not someone who has the knowledge of web application development. As developers, we should always keep the controlling logic and core functionality of an application within our control by implementing inside the source code files. The site admin should only be allowed to change the application data and behavior within the system rather than the application's control logic. Let's consider the following scenario to help you understand the issues in the preceding techniques.

Assume that we want to create a sign-up page for our application. So, we create a page named **sign-up** from the admin dashboard and assign a shortcode or page template to it to display the sign-up form. Afterwards, users can use the sign-up page from the frontend to get registered. Then, we get a new requirement to add some information to the sign-up page. While updating the page, we get the part of shortcode that was deleted by mistake and save it even without knowing. Now the application's sign-up page is broken, and users will not be able to use the system. This is the risk of using the preceding techniques—we can easily break the core controlling functionality of an application.

Instead, we should be allowing both data and behavior changes using the admin dashboard. For example, we can allow the admin to choose the necessary fields for the sign-up form using settings. We can alter the behavior of the sign-up form, but we can never break the sign-up page. Therefore, the preceding two techniques are not ideal in large-scale web application layouts.

Custom templates with custom routing

This technique allows us to have complete control over the template generation process as developers. In *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*, we looked at the basics of custom templates with routing while creating the frontend login and registration pages. Now let's move further by inspecting the advanced aspects of custom template techniques. We have the choice of two techniques for using custom templates in web applications:

- Pure PHP templates
- Template engines

Using pure PHP templates

Pure PHP templates are a widely used technique within popular frameworks, including WordPress. In this technique, template files are created as separate PHP files to contain the visual output of the data. The separation of models, views, and controllers allows us to manage each concern of the application development independent from one another, increasing maintainability and extensibility. In ideal situations, views should have very limited business logic, or if possible, no logic at all. Most probably, designers who don't have much of an idea about PHP coding will be working with the views. Therefore, it's important to keep the views as simple as possible with display logic and data. The data required for views should be generated from models by executing the business logic.

Even though this technique is widely used, it doesn't fulfill the expectation of using views completely. These PHP templates will always have some PHP code included. The main problem with this technique is when someone who doesn't have PHP knowledge makes a mistake in the PHP code placed inside templates; the application will break because of this. PHP was originally meant to be a template engine, and hence we won't have many problems in using PHP templates other than the preceding issue.

The WordPress way of using templates

WordPress uses a function named `get_template_part` for reusing templates as pure PHP files. This function locates the given template parts inside your theme files and makes a file inclusion under the hood. Consider the following code which shows the usage of the `get_template_part` function:

```
get_template_part( $slug, $name );
```

The first parameter, `$slug`, is mandatory, and it is used to load the main template. The second parameter, `$name`, is optional, and it is used to load a specialized version of the template. Let's look at some different usages of this function:

```
get_template_part("project");
get_template_part("project", "wordpress");
```

The first line of code will include the `project.php` file inside the `themes` folder. The second line of code will include the `project-wordpress.php` file, which will be a specialized version of the `project.php` file in typical scenarios. Typically, we pass the necessary data to templates when using template systems. But the `get_template_part` function does not provide the option of passing data as it's a pure file inclusion. However, we have access to data within this context, as this is a pure file inclusion. Also, we have the option of accessing the necessary data through global variables.

If you decide to use the WordPress technique of using template files, make sure to create each and every template file inside your `themes` folder.

Direct template inclusion

Developers who don't prefer the WordPress method of including templates can create their own style of template inclusion. In *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*, we used the direct template method with custom routing. Let's recall the implementation to understand the process:

```
add_action('template_redirect', array($this, 'front_controller'));
public function front_controller() {
  global $wp_query;
  $control_action = $wp_query->query_vars['control_action'];
  switch ($control_action) {
    case 'register':
      do_action('wpwa_register_user');
      break;
    case 'login':
       do_action('wpwa_login_user');
       break;
```

```
        case 'activate':
            do_action('wpwa_activate_user');
            break;
    }
}
```

We intercepted the default template-locating procedure using the `template_redirect` action and used a query variable to switch routes. Then, we implemented the action hooks to contain template inclusion and functionality as shown in the following code:

```
public function activate_user() {
  // Implementing neccessary functions and data generation
  include dirname(__FILE__) . '/templates/info.php';
  exit;
}
```

In this scenario, the `info.php` template has access to all of the data generated inside the `activate_user` function. Developers should execute all of the business logic in the top section of the `activate_user` function. Even though `info.php` has access to all of the data, it's good practice to put the data necessary for the template inside a specific array so that anyone can identify the data used in the template just by looking at the `activate_user` function.

With this technique, we can create template files inside the `themes` folder or `plugins` folders and load it where necessary, making it more flexible than the `get_template_part` function of WordPress.

Theme versus plugin templates

In typical web applications, templates will be created inside a separate folder from the other main components such as models and controllers. WordPress is mainly used for general websites and content management systems. So, the visual representation is much more important than a web application. Hence, theme templates become the top priority in WordPress development, where we can create template files.

Now, the most important question is whether to place web application templates within the `themes` or `plugins` folder. The decision between the theme and plugin templates purely depends on your personal preference and the type of application. First, we have to keep in mind that the most existing theme templates are used for generating CMS-related functionality, and hence they will have lesser impact in advanced web applications. Most web application templates need to be created from scratch. So, answering the following question will simplify your decision making process.

Are you planning to create an application-specific theme?

What I mean by an application-specific theme is that you are willing to change the structure and code of the existing templates to suit your application. These kinds of themes will not be re-used across multiple applications, and switching themes will almost be impossible. The following list illustrates some of the tasks to be executed on existing files to make application-specific themes:

- Heavy usage of custom fields
- Removing existing components such as sidebars and comments
- Using custom action hooks with templates
- Using custom widgetized areas

If your answer is yes, all the templates should be placed inside the `themes` folder as it will not be used for any other application. On the other hand, if you are planning to design new templates for the application while keeping the existing templates without major customizations, it's a good practice to create the application-specific templates inside the `plugins` folder. This technique separates application-specific templates from the core templates, allowing you to switch the theme anytime without breaking the application. Also, maintenance becomes easier as core templates and application-specific templates are easily tracked separately.

Template engines

So far, we have talked about pure PHP templates. Template engines allow us to separate the HTML code from PHP using specific tag-based syntaxes. In *Chapter 4*, *The Building Blocks of Web Applications*, we looked at the Twig template engine while building custom post types for the portfolio application. Twig uses the `{{ data }}` tag to output the data to the templates. Let's consider some of the advantages of using a template engine over pure PHP templates.

Template engines use specific tags to define data. Therefore, the application will not break when the tags are used incorrectly, which makes it easier for designers to work with templates. Templates only have access to the passed data where you will get all the data in the context in pure PHP technique. Reusability of templates is increased with the use of template inheritance.

Personally, I prefer the use of template engines when working on large web applications. But with WordPress, we cannot implement template engines to its purest functionality. We will have to use template engines in combination with pure PHP templates. Let's find out the limitations of using template engines within WordPress:

- WordPress uses action hooks in themes to provide the functionality. Such action hooks include `wp_head`, `wp_footer`, and `comment_form`. The templates generated with frameworks don't contain the PHP code, and hence we cannot implement the necessary hooks within templates.

Web applications require an extensive amount of custom hook points, and hence it's not possible to include them inside the templates generated from these frameworks and libraries.

- In the existing templates, we use functions such as `get_header` and `get_sidebar` to get the dynamic template parts. We cannot pass the code generated from these functions to templates since WordPress allows changing this content dynamically through plugins.

Considering these reasons, it's obvious that we cannot completely use templates generated by these template engines within WordPress. But template engines can be used effectively to re-use the template parts inside main templates.

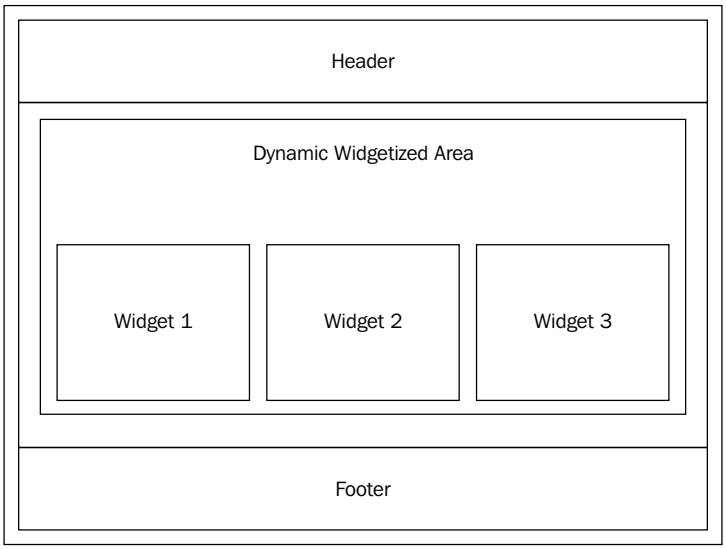
Building a portfolio application's home page

So far, we have learned the theoretical aspects of creating templates inside WordPress themes. Now it's time to put them into practice by creating the home page for a portfolio application. In this section, we are going to talk about the importance of widget-based layouts for web applications while building the home page.

What is a widget?

A widget is a dynamic module that provides additional features to your website. WordPress uses widgets to add content to the website sidebars. In most web applications, we won't get sidebars while creating layouts. However, we can take widgets beyond the conventional sidebar usage by creating fully widgetized layouts for increased flexibility. With WordPress, we can widgetize any part of the application layout, allowing developers to add content dynamically without modifying the existing source code.

Let's plan the structure of the home page layout by using widgets, as shown in the following diagram:



According to the preceding diagram, the home page will be fully widgetized by using a single widget area. At this stage of the project, we are going to include three widgets within the widgetized area to display the recent developers, recent projects, and recent followers. There is no limitation to the number of widget areas allowed per screen, and hence you can define multiple widget areas when necessary. Also, we can keep part of the layout static while widgetizing the other parts.

In web applications, widgets play a very important role compared to websites. By widgetizing layouts, we allow the content to be dynamic and flexible for future enhancements.

Widgetizing application layouts

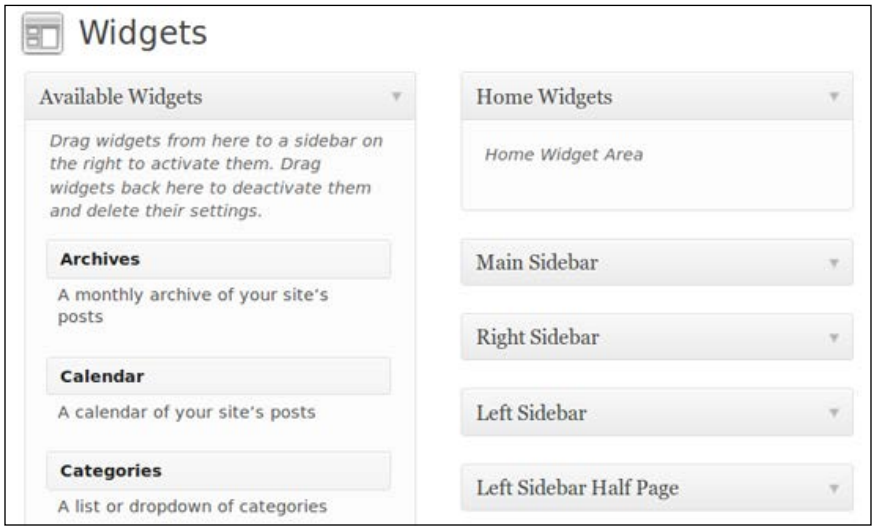
As mentioned earlier, we have the option of creating template files inside a theme or a plugin. Here, we are going to create the templates inside the `plugins` folder to improve flexibility. So, let's begin the process by creating another plugin named `WPWA Theme Manager`. Create a folder named `wpwa-theme` and define the main plugin file as `class-wpwa-theme.php` with the usual plugin definitions. Next, we can update the constructor code as follows to register the widgetized area for the home page:

```
class WPWA_Theme {

  public function __construct() {
    $this-> register_widget_areas();
  }
  public function register_widget_areas() {
    register_sidebar(array(
        'name' => __('Home Widgets','wpwa'),
        'id' => 'home-widgets',
        'description' => __('Home Widget Area', 'wpwa'),
        'before_widget' => '<div id="one" class="home_list">',
        'after_widget' => '</div>',
        'before_title' => '<h2>',
        'after_title' => '</h2>'
    ));
  }
}
```

Surprisingly, we don't have to use an action hook to create the widget areas. WordPress widgets are designed to work on sidebars, and hence the function name `register_sidebar` is used for defining widgetized areas. But we don't need an actual sidebar to define widget areas for various purposes. Most of the configurations used for this function are important for working with the widget areas that are explained as follows:

- `name`: This is used to define the name of the widgetized area. It will be used to load the widgetized area on frontend templates.
- `id`: This is used to uniquely identify the widget area.
- `before_widget` and `after_widget`: These are used to provide additional HTML content before and after the widget's contents.
- `before_title` and `after_title`: These are used to provide additional HTML content before and after the widget's title.

Once the previous code is implemented, you will get a dynamic widgetized area in the admin dashboard as shown in the following screenshot:



Creating widgets

Having defined the widget area, we can create some dynamic widgets to populate the home page content. Registering the widgets is similar to the process used for registering the widgetized areas. Let's create a function for including and registering widgets into the application:

```
public function register_widgets() {
  $base_path = plugin_dir_path(__FILE__);
  include $base_path . 'widgets.php';
  register_widget('Home_List_Widget');
}
```

We are going to create all three widgets required for the home page using the `widgets.php` file inside the `plugins` folder. First, we have to include the `widgets` file inside the `register_widgets` function. Second, we have to register each and every widget using the `register_widget` function. We can create three separate widgets for the home page. But we are going to create a single widget to illustrate the power of reusability for complex web applications.

Therefore, we have limited the widget's registration to a single widget named `Home_List_Widget`. This will be the class of the widget that extends the `WP_Widget` class. Finally, we have to update the constructor with the `widgets_init` action as shown in the following code:

```
add_action('widgets_init', array($this, 'register_widgets'));
```

In general, widgets contain a prebuilt structure that provides their functionality. Let's understand the process and functionality of a widget using its base structure, as illustrated in the following code:

```
class Home_List_Widget extends WP_Widget {
  function __construct() { }
  public function widget($args, $instance) { }
  public function form($instance) { }
  public function update($new_instance, $old_instance) {  }
}
```

First, each widget class should extend the `WP_Widget` class as the parent class. Then we need four components including the constructor to make a widget. Let's see the role of each of these functions within widgets:

- `__construct`: This function is used to register the widget by calling the parent class constructor with necessary parameters
- `widget`: This function is used to construct the frontend view of the widget using the processed data
- `form`: This function is used to create a backend form for the widget for defining necessary configurations and options
- `update`: This function is used to save and update the fields inside the form function of the database tables

Now, we have a basic idea about the prebuilt functions within widgets. Let's start the implementation of `Home_List_Widget` to create the home page content. Basically, this widget will be responsible for providing the home page content such as the recent developers, projects, and followers. Considering the current requirements, we need two form fields for defining the widget title and choosing the type of widget. Let's get things started by implementing the constructor as follows:

```
public function __construct() {
  parent::__construct(
    'home_list_widget', // Base ID
    'Home_List_Widget', // Name
    array('description' => __('Home List Widget', 'wpwa'),) // Args
    );
}
```

Here, we call the parent class constructor by passing ID, name, and description. This will initialize the main settings for the widget. We can then have a look at the implementation of the `form` function using the following code:

```
public function form($instance) {
  if (isset($instance['title'])) {
    $title = $instance['title'];
  }
  else {
    $title = __('New title', 'wpwa');
  }
  if (isset($instance['list_type'])) {
    $list_type = $instance['list_type'];
  }
  else {
    $list_type = 0;
  }
?>
  <p>
    <label for="<?php echo $this->get_field_name('title');
      ?>"><?php _e('Title:'); ?></label>
    <input class="widefat" id="<?php echo $this->
      get_field_id('title'); ?>" name="<?php echo $this->
      get_field_name('title'); ?>" type="text" value="
      <?php echo esc_attr($title); ?>" />
  </p>
  <p>
    <label for="<?php echo $this->get_field_name('list_type');
       ?>"><?php _e('List Type:'); ?></label>

    <select class="widefat" id="<?php echo $this->
      get_field_id('list_type'); ?>" name="<?php echo $this->
      get_field_name('list_type'); ?>" >

    <option <?php selected( $list_type, 0 ); ?>
       value='0'>Select</option>

    <option <?php selected( $list_type, "dev" ); ?>
       value='dev'>Latest Developers</option>
     </select>
  </p>
<?php
}
```

First, we check the existing values of form fields using the `$instance` array. This array will be populated with the existing values for the form fields from the database; initially, these fields will contain empty values. Next, we have defined the form fields required for the widget. The title of the widget is implemented as a text field, while the list type is implemented as a drop-down field with developers, projects, and followers as the options. Here, we have only defined the value for developers to simplify our explanations. You can use the plugin source code for complete values.

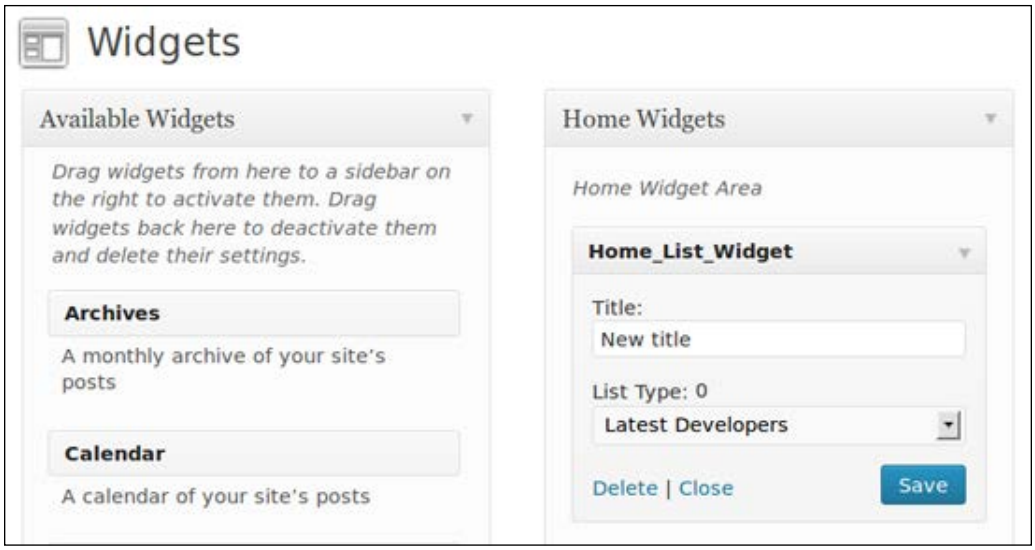
You might have noticed the use of the `get_field_name` and `get_field_id` functions inside the name and ID attributes of the form fields. These two functions are located in the parent class and are used to generate dynamic names in a common format. Use the view source of the browser window and you will find the field names as something similar to the following code:

```
widget-home_list_widget[1][title]
widget-home_list_widget[1][list_type]
```

These types of field names and IDs are assigned to make the automation easier. The widget's data-saving method is completely automated, and hence we have to only define the necessary form fields. Saving the data to the database will be done automatically by WordPress. Next, we need to implement the form-updating function as shown in the following code:

```
public function update($new_instance, $old_instance) {
  $instance = array();
  $instance['title'] = (!empty($new_instance['title']) )
  ?strip_tags($new_instance['title']) : '';
  $instance['list_type'] = (!empty($new_instance['list_type']) )
  ? strip_tags($new_instance['list_type']) : '';
  return $instance;
}
```

The `update` function is relatively simpler than other functions, as we just have to define the form field keys inside the `$instance` array. The rest of the database updating will be done automatically behind the scenes. Once those fields are defined, you will get a new widget item named **Home List Widget** in the **Available Widgets** area. You can drag the widget into **Home Widget** to include the widget in the home page. Now your screen should look like the following screenshot:



We need to implement the `form` function to complete the development of the home page widgets. Here, we want to display the list of developers, projects, or followers. We will be constructing the developer list for the purpose of explanation, and you can find the remaining widget implementations within the source code. This function generates the frontend display contents. Throughout this book, we have given higher priority to separating templates from business logic. Therefore, we need to use separate templates for generating the HTML required for a widget's frontend display. As discussed earlier, we can either use pure PHP templates or Twig templates for generating widget templates. Here, we are going to use PHP templates to understand the process of creating a custom template loader.

Creating a custom template loader

First, we have to add a function to the `WP_Theme` class of the plugin to include the template loader file, as shown in the following code. Also, we have to update the constructor to call the `template_init` function upon plugin initialization:

```
public function template_init() {
    include_once 'class-wpwa-template-loader.php';
}
```

Create a new file named `class-wpwa-template-loader.php` inside the plugin class to implement the template loader. Let's implement the loader by creating a new class named `WPWA_Template_Loader` inside the `class-wpwa-template-loader.php` file:

```
class WPWA_Template_Loader{
    private $template_dir;
    public function __construct(){
          $this->template_dir = "templates";
    }
    public function render($name,$data =  array() ){
       include plugin_dir_path(__FILE__)
       .$this->template_dir."/".$name.".php";
    }
}
```

The folder named `templates` will be used to create all the templates required for the theme. Then, we will add a function named `render` to include the requested template by filename. This loader will be used throughout the project for loading the custom template. Now we can move back to the implementation of the `widget` function of the home page widget as given in following code:

```
public function widget($args, $instance) {
    extract($args);
    $title = apply_filters('widget_title', $instance['title']);
    $list_type = apply_filters('widget_list_type',
      $instance['list_type']);
    echo $before_widget;
    if (!empty($title))
      echo $before_title . $title . $after_title;

      $tmpl = new WPWA_Template_Loader();
      switch ($list_type) {
        case 'dev':
          $user_query = new WP_User_Query(array('role' =>
            'developer'));
          $data = array();
          $data["records"] = array();
          foreach ($user_query->results as $developer) {
            array_push($data["records"],
            array("ID" => $developer->data->ID, "title" =>
            $developer->data->display_name));
      }
      $data["title"] = $title;
      $tmpl->render("home_list", $data);
      break;
    }
     echo $after_widget;
}
```

The first three lines of this function extracts the arguments passed to the `widget` function and retrieves the widget option values by applying the necessary filters. The next three lines are used to wrap the widget with dynamic content when required. Then we come to the most important part of the function where we generate the front layout and the data. The template loader class, `WPWA_Template_Loader`, is initialized into a variable named `$tmpl` for dynamic template loading. Then, we check the value of the drop-down field using the `switch` statement. In the widget form, we included `dev` as the key for developers. The other two option values can be found in the source code. Inside the `dev` case section, we query the database to retrieve the recently joined developers in the application using `WP_User_Query`. The user role for developers is used to filter the values. Then, we add the generated results into the `$data` array to pass them into the template.

Finally, we call the `render` function of the template loader object by passing the template name as `home_list`. You can create a PHP file named `home_list.php` inside the `templates` folder. The implementation of the home page widget's template is given in the following code:

```
<div class='home_list'>
    <div class='list_panel'>
      <?php foreach($data["records"] as $record){ ?>
      <div class='list_row'><a href=''>
         <?php echo $record['title']; ?></a>
       </div>
       <?php } ?>
    </div>
</div>
```

The preceding template generates the developers list using the data passed into the template. So far, we have created the necessary widgets and the widget areas for the home page. The final task of this process is to create the home page template itself.

Designing the home page template

We have to create a file named `home.php` inside the `templates` folder. At the beginning of the widget creation process, we planned the structure of the home page using a wireframe. Now we need to adhere to the structure while designing the home page, as shown in the following code:

```
<?php get_header(); ?>
<?php if ( !function_exists('dynamic_sidebar') || !dynamic_
sidebar('Home Widgets') ) :
```

```
endif;

?>
<?php get_footer(); ?>
```

These three lines of code make the complete design and the data for the home page. Usually, every page template contains a WordPress header and footer by using the `get_header` and `get_footer` functions. The code used between the header and footer checks for the existence of the `dynamic_sidebar` function for loading dynamic widget areas. Then, we load the widgetized area created in the *Widgetizing application layouts* section by using the sidebar name. This area is populated with the widgets assigned in the admin section.

The design and functionality of the home page is now completed and ready to be displayed in the application. But, we haven't given instructions to WordPress to load it as the home page. So, let's go to the main plugin file, `wpwa_theme.php`, to define the home page. First, we have to add the following line of code to the plugin constructor to customize the template-redirection process:

```
add_action('template_redirect', array($this,
  'application_controller'));
```

The implementation of the `application_controller` function will look like the following code:

```
public function application_controller() {
    global $wp_query;
    $control_action = isset (
       $wp_query->query_vars['control_action']) ?
       $wp_query->query_vars['control_action'] : '';

    if (is_home () && empty($control_action) ) {
      $tmpl = new WPWA_Template_Loader();
      $tmpl->render("home");
      exit;
    }
}
```

WordPress allows us to check the home page of the application using the `is_home` function. Our plan is to redirect the default home page to the custom home template created in the preceding sections. Hence, we intercept the default routing process and use the template loader class to dynamically use the home template using the `render` function. Also, we have to make sure that the `control_action` query variable is empty before rendering the home page.

Now, you should get a blank page with the header and footer in the home page. Next, you can log in as the admin and add the developers, followers, and projects widgets to the widgetized area in the admin section and save the changes. The final output of the home page will look like the following screenshot:

Generating an application's frontend menu

Typically, a web application's frontend navigation menu varies from the backend menu. WordPress has a unique backend with the admin dashboard. The logged-in users will see the backend menu on the top of frontend screens as well. In the previous chapter, we looked at various ways of customizing the backend navigation menu. Here, we are going to look at how the frontend menu works within WordPress.

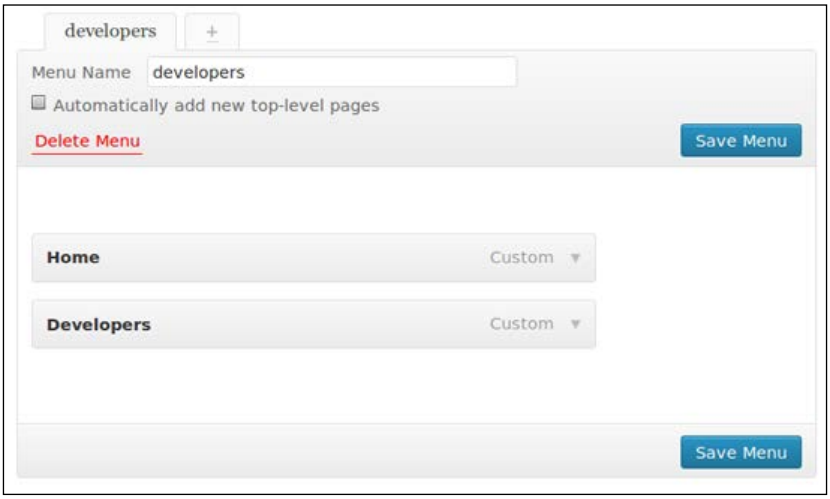
Navigate to the `themes` folder and open the `header.php` file of the Responsive theme. You will find the implementation for the frontend menu using the `wp_nav_menu` function. This function is used to display the navigation menus generated from the **Appearance** section of the WordPress admin dashboard. As far as the portfolio application is concerned, we need four different frontend menus for normal users, developers, followers, and members. By default, WordPress uses the assigned menu or the default page list to create the menu. Here, we are going to create four different navigation menus based on the user role.

For the purposes of explanation, we will be manually creating menus for each user role. In large applications, we need to figure out a method to dynamically generate menu items based on roles and permissions.

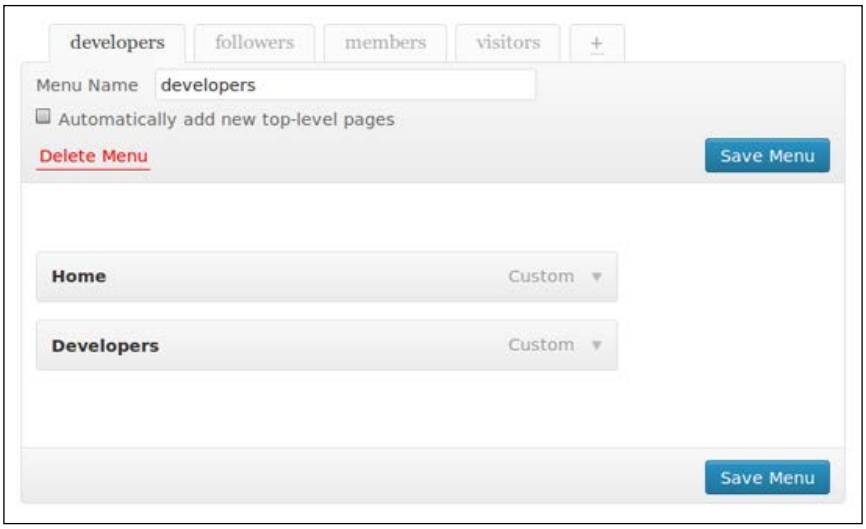
Creating a navigation menu

We have to log into the portfolio application as the admin and navigate to **Appearance** | **Menu** to create new frontend menus. Since most of you are familiar with creating menus for websites, I am going to keep the explanation process as short as possible. Now, enter a menu name and click on the **Save Menu** button to create the menu. Here, we are going to start by creating the developers menu. Now your screen will look something similar to the following screenshot:

Once created, we have to add the menu items specific to the developer's user role. Since we haven't got many frontend screens at this stage, we will be specifying **Home** and **Developers** as the menu items. Later, we can add the necessary submenus to the **Developers** menu item. Both the **Home** page and the **Developers** page will be created from scratch using the custom templates procedure. Therefore, we have to use custom links to create menu items. After adding two menu items, click on the **Save Menu** button and you will get something similar to the following screenshot:

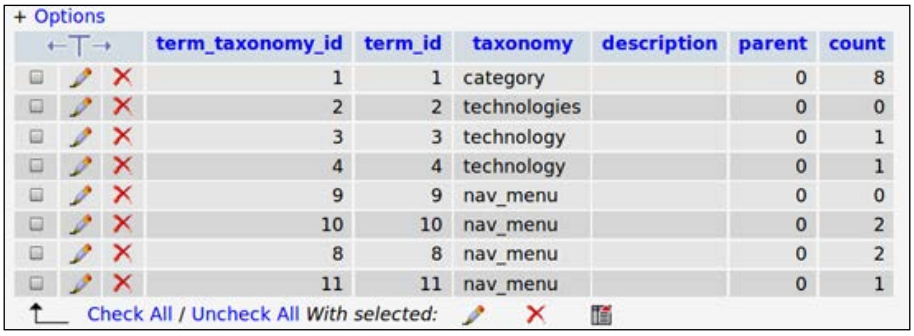


We have to follow this process for other user roles in the portfolio application. Make sure to use the menu names `followers`, `members`, and `visitors` for the remaining user roles. Once all four menus are created, your menu screen will look like the following screenshot:



Manual implementation of frontend menus is not practical in larger applications, and hence we should have a sound knowledge of working with menu-related database tables and fields to automate the process.

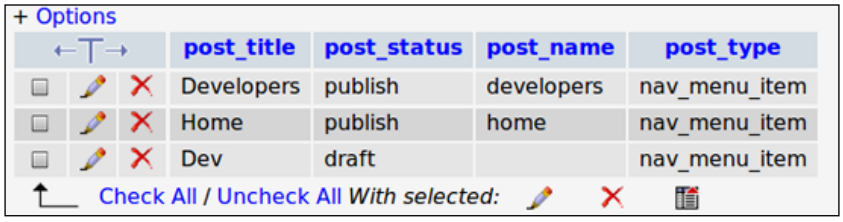
Once the custom menu is created, the `wp_terms` table will have a new entry for the menu with the name of the menu and the slug. Then, we have to look at the `wp_term_taxonomy` table for the related entry of the menu item with the `taxonomy` column defined as `nav_menu`. This database row will also contain the number of menu items inside the menu, which is displayed in the `count` column as illustrated in the following screenshot taken from phpMyAdmin:



term_taxonomy_id	term_id	taxonomy	description	parent	count
1	1	category		0	8
2	2	technologies		0	0
3	3	technology		0	1
4	4	technology		0	1
9	9	nav_menu		0	0
10	10	nav_menu		0	2
8	8	nav_menu		0	2
11	11	nav_menu		0	1

Check All / Uncheck All With selected:

Finally, we have to look for information about each and every menu item stored in the `wp_posts` table. Consider the following screenshot taken from the `wp_posts` table in phpMyAdmin:



As you can see, all the menu items are stored as table rows in the `wp_posts` table with a `post_type` column named `nav_menu_item`. The `post_status` column stored as `publish` means that the menu item is active, while `draft` means that the menu item is inactive or deleted. Developers can use a combination of these three database tables with user roles and permissions to automate the menu creation process.

Displaying user-specific menus on the frontend

Having created user-specific menus in the WordPress admin section, we can now move on to displaying them on the frontend. Open the `header.php` file of the Responsive theme and you will find the main menu defined as `header-menu` using the `theme_location` parameter. The main menu generation code is defined as follows:

```
<?php wp_nav_menu(array(
  'container'  => '',
  'theme_location'  => 'header-menu')
);
?>
```

By default, this code will load the menu used for the **Header Menu** drop-down box in **Appearance | Menus | Theme Locations**. In web applications, we need user-role-specific menus, and hence we can leave the **Theme Locations** section empty. Replace the preceding code with following code to display frontend menus based on a user role:

```
<?php
if (current_user_can('edit_posts')){
  wp_nav_menu( array('menu' => 'developers' ));

}elseif ( current_user_can('follow_developer_activities') ){
```

```
    wp_nav_menu( array('menu' => 'followers' ));

  }elseif(current_user_can('manage_membership')){
    wp_nav_menu( array('menu' => 'members' ));

  }else{
    wp_nav_menu( array('menu' => 'visitors' ));
  }
  ?>
```

WordPress doesn't have a proper method to check for user roles, including custom roles. Even though we can use `current_user_can` to check for roles, the WordPress documentation suggests that it might work incorrectly for custom roles. Therefore, we need a user-role-specific capability for checking the role. Here, we have implemented the four menus created in the admin section by checking the necessary capabilities. The name of the menu is used as the one and only parameter.

In web applications, create a user-role-specific capability to check various permissions. This capability doesn't have to provide any functionality. Instead, it will be used to provide role-based permissions.

Once logged in, each user will have a frontend menu specific to their user role, and now we have a basic user-role-based menu for a portfolio application.

Creating pluggable and extendable templates

Templates can be categorized into several types based on their functionality. Each of these types of templates plays a different role within complex applications. Proper combination of these template types can result in highly maintainable and reusable applications. Let's explore the functionality of various template types.

The simplest type of template contains the complete design for each and every screen in the application. These types of templates are not reusable. We can also have template parts that get included into some other main templates. These types of templates are highly reusable across multiple other templates. The header and footer are the most common examples of such templates.

Pluggable or extendable templates

These types of templates are highly adaptable and flexible for future enhancements or modification of existing features. In web applications, we use a technique similar to inheritance in order to provide extendable templates. These templates can be easily modified to add features without affecting the existing functionality. On the other hand, WordPress provides the capability to create similar types of templates with its pluggable architecture by using action hooks. There is a drastic difference between the way these two techniques work in WordPress and normal web applications. But the final output is quite similar in nature. Let's find out how extendable templates are used in web applications before digging into WordPress.

Extendable templates in web applications

Inheritance is a popular concept used to define the relationship between objects and is also a way for subclasses to inherit attributes from the parent class. Usually, inheritance is used for managing business or database logic, however some of the template engines offer inheritance capabilities for templates as well. In *Chapter 4*, *The Building Blocks of Web Applications*, we chose Twig as the template engine for creating reusable templates, and it does offer the inheritance capabilities. Let's explore the process of extendable templates through Twig. Consider the following code for the sample template code in a template file named `developer.html`:

```
<div class='content'>
  <div id="developer_list">
  {% block profile %}
  <table>
    <tr>
      <th>Name</th>
      <th>Role</th>
      <th>Experience</th>
   </tr>
    <tr>
      <td>John</td>
      <td>Web Developer</td>
      <td>3 Years</td>
    </tr>
  </table>
  {% endblock %}
   // Other developer related template code
  </div>
</div>
```

Here, we have a basic template that generates a list of developers in the application using a table-based layout. The Twig template engine allows us to use blocks of code within templates. In the preceding template, we have used a block named `profile` for the developers list. You can have many blocks within a single template.

Now assume that we have re-used the preceding template within several other templates. Then a new requirement comes in where we have to provide a developer list with header and pagination controls in one of the templates. Now, there is no point in creating a separate template by duplicating the contents of this template. Instead, let's see how we can extend the same template to provide a different functionality:

```
{% extends "developer.html" %}
{% block profile %}
    <div class='header'>Developer List Header</div>
      <table>
        <tr>
          <th>Name</th>
          <th>Role</th>
          <th>Experience</th>
        </tr>
        <tr>
          <td>John</td>
          <td>Web Developer</td>
          <td>3 Years</td>
        </tr>
      </table>
    <div class='pagination'>Pagination Control Buttons</div>
{% endblock %}
```

First, we create a new template file and extend the parent template named `developer.php` using the `extends` keyword. Now we have access to the complete content of the `developer.php` file within the child template. Then, we can redefine the profile block with additional requirements to replace the original profile block in the parent template. With this technique, we can extend a part of a template while reusing the other parts, making it highly adaptable for future modifications.

Pluggable templates in WordPress

WordPress uses hook-based architecture for adding new functionalities to existing screens. We can define certain hook points within templates and allow developers to plug dynamic content through plugins. Both web application templates and WordPress templates are extendable, but their functionality can be different.

WordPress action hooks are very powerful for adding new behavior to existing templates. But at the current stage, it's not as powerful as pure inheritance in web applications.

Now let's see how we can implement the previous scenario with the use of WordPress hooks. Consider the initial template for generating the developers list:

```
<div class='content'>
    <div id="developer_list">
    <?php do_action('before_developer_list'); ?>
    <table>
      <tr>
        <th>Name</th>
        <th>Role</th>
        <th>Experience</th>
       </tr>
       <tr>
         <td>John</td>
         <td>Web Developer</td>
         <td>3 Years</td>
       </tr>
    </table>
    <?php do_action('after_developer_list'); ?>
       // Other developer related template code
    </div>
</div>
```

In the preceding code, we have the same layout with two actions named `before_developer_list` and `after_developer_list`. The function named `do_action` is used to execute a function defined by the `add_action` hook. Once `do_action` is defined, plugin developers can customize the layouts and functionality using the `add_action` definitions. So, here we can implement the header and pagination controls by defining custom functions using `add_action` as illustrated in the following code:

```
add_action('before_developer_list', 'customize_before_list');
function customize_before_list(){
  echo "<div class='header'>Developer List Header</div>";
}
add_action('after_developer_list', 'customize_after_list');
function customize_after_list(){
  echo "<div class='pagination'>Pagination Control Buttons</div>";
}
```

In the preceding code, we have added the header and pagination controls using the available template hooks. Here, we have only used two parameters for action: name and function. Apart from the two required parameters, `add_action` can have optional parameters for defining the function's priority and arguments.

The priority parameter of `add_action` determines the order in which actions are executed when we have multiple implementations of the same action. The priority will be provided by the third parameter, which has the default value of `10`. We have to increase or decrease the priority value to get the components on the top or bottom of the page respectively.

Comparing WordPress templates with Twig templates

In the preceding sections, we looked at the implementation of a simple component using both WordPress' technique as well as the technique used by template engines. In contrast, the Twig template is highly flexible than the WordPress action hooks method.

Assume that we wanted to include a new column into the data table. With the Twig template, we can create a new template for the data table while accessing all other template parts from the parent template. But action hooks don't offer such flexibility in customizing the existing features. We can only place actions before or after a certain element. Therefore, we can't add dynamic columns to the data table with the two hooks used in the previous code.

We need to keep in mind that both the WordPress and Twig templates have pros and cons based on different scenarios. As developers, it's our responsibility to choose wisely between both the techniques for improved web application development.

Extending the home page template with action hooks

Let's identify the practical usage of action hooks for extending web application layouts. In the earlier sections, we developed the home page with three widgets with a reusable template inside a dynamic widget area. Now, we have to figure out the extendable locations of those widgets. Consider the following scenario:

Assume that we have been asked to add a button in front of each developer in the home page widget. Users who are logged into the application can click on the button to instantly follow the developers. Implementation of this requirement needs to be done without affecting or changing the other two widgets. Also, we have to plan for similar future requirements for other widgets.

The most simple and preferred way of many beginner-level developers is to create three separate templates for the widgets and directly assign the button into the widget by modifying the existing code. As developers, you should be familiar with the open/closed principle in application development. Let's see the Wikipedia definition of the open/closed principle:

> *software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

This means we should never change the working components of the application. Hence, changing the widget to add the new requirement is not the ideal method. Instead, we should be looking at extending the existing components. So, we need to make use of WordPress action hooks to define extendable areas in the widget.

Customize widgets to enable extendable locations

First, let's modify the widget template with extendable hooks. Ideally, this should have been done in the initial stage of widget creation. Open the `home_list` template inside the `templates` folder and replace the existing code with the following code:

```
<div class='home_list_item'>
  <div class='list_panel'>
    <?php foreach($data["records"] as $record){ ?>
       <div class='list_row'>
       <a href=''><?php echo $record['title']; ?></a>
    <?php do_action('wpwa_home_widgets_controls',
      $record['type'],$record['ID']); ?>
       </div>
    <?php } ?>
  </div>
</div>
```

Here, we have included an action hook named `wpwa_home_widgets_controls`, which takes two parameters for the action type and ID of the record. This widget layout is applied for multiple widgets, and hence we don't want the action to be executed for all the available widgets. Therefore, we use the type parameter to check for widgets that require the `do_action` call. The second parameter is that an ID will be used to execute functions for these records. Here, it will be used to identify the developer to be followed.

Next, we have to change the widget code to pass the type parameter. We must pass a value for widgets that require the execution of `wpwa_home_widgets_controls`. Otherwise, keep it blank to skip the action execution. In this scenario, we will be passing a value named `follow` as the type parameter.

Then we need to implement the `home_widgets_controls` action by including an `add_action` definition. Place the following action inside the constructor of the `WPWA_Theme` class in the main plugin:

```
add_action('wpwa_home_widgets_controls', array($this,
  'home_widgets_controls'), 10, 2);
```

The preceding code defines the `wpwa_home_widgets_controls` action with the default priority of `10` and two parameters. Finally, we have to implement the `home_widgets_controls` function to output the **Follow** button to the widget list as shown in the following code:

```
public function home_widgets_controls($type, $id) {
  if ($type == 'follow') {
     echo "<input type='button' class='$type' id='" . $type . "_"
       . $id . "' data-id='$id' value='" . ucfirst($type) . "'
     />";
  }
}
```

This function uses the two parameters passed by the `do_action` call. After validating the type, we can output the **Follow** button with the necessary attributes and CSS classes. Now you will have a screen similar to the following screenshot with the new **Follow** button:

Once the button is clicked, we can use the `data-id` attribute to get the developer ID and make an AJAX request to execute the follow and unfollow operations.

With the latest modifications, the home page widgets have become highly flexible for future modifications. Now developers have the ability to add more functionalities through the control buttons without changing the existing source code. Let's summarize the list of tasks for adding new features to the widgets:

- Pass a type value to the template with the widget data
- Implement the action using the `add_action` function with the necessary parameters
- Use the priority value to change the order of the control buttons
- Check the type value and generate the necessary HTML code

WordPress action hooks are a powerful technique for extending themes and plugins with dynamic features. Developers should always look to create extendable areas in their themes and plugins. Basically, you need to figure out the areas where you might get future enhancements and place action hooks upfront for easier maintenance.

Planning action hooks for layouts

Usually, WordPress theme developers build template files using unique designs and place the action hooks afterwards; these hooks are mainly placed before and after the main content of the templates. This technique works well for designing themes for websites. But a web application requires flexible templates, and hence we should be focusing on optimizing the flexibility as much as possible. So, the planning of hook points needs to be done prior to designing. Consider the following sample template code of a typical structure of a hook-based template:

```
<?php do_action('before_menu'); ?>
<div class='menu'>
<div class='menu_header'>Header</div>
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
</div>
<?php do_action('after_menu'); ?>
```

The preceding code is well structured for extending purposes using action hooks. But we can only add new content before and after the menu container. There is no way of changing the content inside the menu container. Let's see how we can increase the flexibility using the following code:

```
<?php do_action('before_menu'); ?>
<?php do_action('menu'); ?>
<?php do_action('after_menu'); ?>
```

Now the template contains three action hooks instead of hard-coded HTML. So, the original plugin or theme developer must implement the action hook using the following code:

```
add_action('menu','create_dynamic_menu');
function create_dynamic_menu(){
  echo "<div class='menu'>
  <div class='menu_header'>Header
  </div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
  </div>";
}
```

So, the base theme also uses hooks to embed the template code. Now it's possible to change the inner components of the menu using another set of action hooks.

Even though the `echo` statement is used to simplify explanations, this HTML code needs to be generated using a separate template file in ideal scenarios.

Let's see how we can override the original menu template with our own template at runtime using the following code:

```
remove_action('menu','create_dynamic_menu');
add_action('menu','create_alternative_dynamic_menu');
function create_alternative_dynamic_menu (){
  echo "<div class='menu'>
    <div class='menu_header'>Header</div>
    <ol>
      <li>Item 1</li>
      <li>Item 2</li>
    </ol>
    </div>";
}
```

First, we have to remove the original implementation of the menu using the `remove_action` function. The syntax of `remove_action` should exactly match with the `add_action` definition to make things work. Once removed, we implement the same action with a different function to provide a different template to the original template. This is a very useful way of extending and overriding an existing functionality. In order to use this technique, you have to plan the action hooks from the initial stage of the project. Now, you should have a clear idea about the advanced template-creation techniques in WordPress.

Time for action

In this chapter, we talked about some of the advanced techniques in WordPress themes. Developers who don't have exposure to advanced application development with WordPress might find it a bit difficult to understand these techniques. Therefore, I suggest you try out the following tasks to get familiar with advanced theme creation techniques:

- Automate the process of creating a frontend menu item. The admin should be able to add menu items to multiple navigation menus in a single event, or a complete menu should be generated based on permission levels.
- Complete the developer following process using AJAX and the necessary WordPress actions.
- Create an extendable layout for the developer portfolio page to contain personal information, projects, services, books, and articles. Make sure to optimize the flexibility of the layout.

Summary

The frontend of an application presents the backend data to the user in an interactive way. The possibility of requesting for frontend changes of an application is relatively high as compared to the backend. Therefore, it's important to make the application design as stylish and as flexible as possible. Advanced web applications will require complex layouts that can be extended by new features. Planning for the future is important, and hence we prioritized the content of this chapter to discuss extending the capabilities of the WordPress theme files using widgetized architecture and custom action hooks.

We also had a look at the integration of custom hooks with widgetized areas while building the most basic home page for the portfolio application.

A navigation menu is vital for providing access to templates based on user roles and permissions. Here, we looked at how we can create separate frontend menus based on user roles and how to display them on the frontend.

In the next chapter, we are going to look at the use of an open source plugin within WordPress. In web application development, developers usually don't get enough time to build things from scratch. So, it's important to make use of the existing open source libraries for rapid development. Get ready to experience the usage of a popular, open source plugin within WordPress.

8
Enhancing the Power of Open Source Libraries and Plugins

WordPress is one of the most popular open source frameworks, serving millions of people around the world. The WordPress core itself uses dozens of open source libraries to power the existing features. Web application development is a complex and time-consuming affair compared to generic websites. Hence, developers get very limited opportunities for building everything from the ground up, creating the need for using stable open source libraries.

With the latest versions, WordPress has given higher priority for using stable and trending open source libraries within its core. `Backbone.js`, an Underscore and jQuery masonry, have been the recent popular additions among such open source libraries. Inclusion of these types of libraries gives a hint about the improvement of WordPress as a web development framework.

We will be discussing the various usages of these existing open source libraries within the core and how to adapt them into our applications. This chapter also includes some of the popular techniques such as Twitter and Facebook logins to illustrate the integration of external libraries that don't come with the core WordPress framework.

In this chapter, we will cover the following topics:

- Open source libraries inside the WordPress core
- Open source JavaScript libraries in the WordPress core
- Creating a developer profile page with Backbone.js
- Integrating Backbone.js and Underscore.js

- Understanding the importance of code structuring
- Using PHPMailer for custom e-mail sending
- Implementing user authentication with Opauth
- Building a LinkedIn app
- Authenticating users to our application

So let's get started.

Why choose open source libraries?

Open source frameworks and libraries are taking control in web development. On the one hand, they are completely free and allow developers to customize and create their own versions. On the other hand, large communities are building around open source frameworks. Hence, these frameworks are improving in leaps and bounds at an increasing speed, providing developers with more stable and bug-free versions. WordPress uses dozens of open source libraries and there are thousands of open source plugins in its `plugins` directory. Therefore, it's important to know how to use these open source libraries in order to make our lives easier as developers. Let's consider some of the advantages of using stable open source products:

- Large community support
- Ability to customize existing features by changing source code
- No fees are involved based on per site or per person licensing
- Usually reliable and stable
- Possibility of having more features through forked versions

These reasons prove why WordPress uses these libraries to provide features, and why developers should be making use of them to build complex web applications to cater to time consuming tasks.

Open source libraries inside the WordPress core

As mentioned earlier, there are several libraries available within the core that have yet to be noticed by many WordPress developers. Most beginner-level developers tend to include such libraries in their plugins when it's already available inside the core framework. This happens purely due to the nature of WordPress development, where most of the development is done for generic websites with very limited dynamic content.

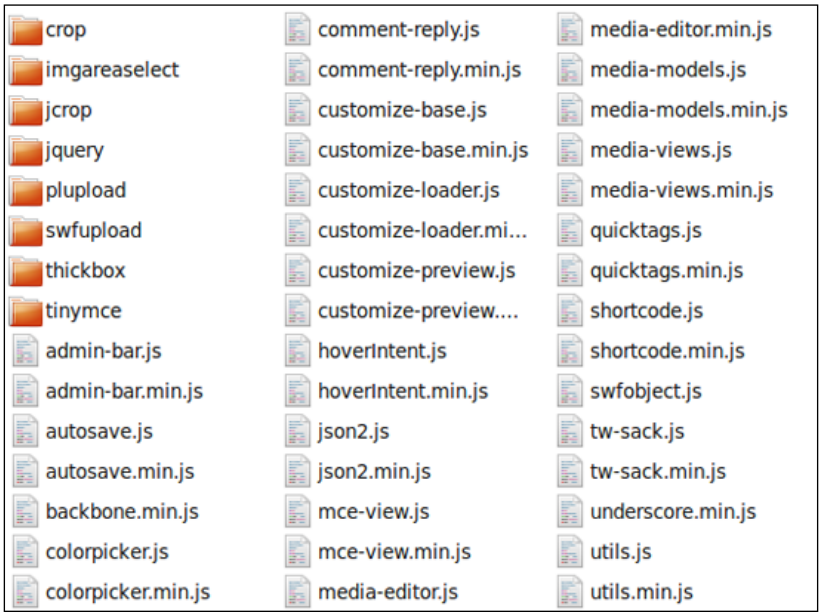
As we move into web application development, we should understand the need for using existing libraries whenever possible due to the following reasons:

- WordPress contains libraries that are more stable and compatible
- It also prevents the duplication of libraries, and reduces the size of the project files

I am sure you must be familiar with libraries and plugins such as jQuery and TinyMCE within WordPress. Let's discover the lesser known and recently added libraries in order to make full use of them with web applications.

Open source JavaScript libraries in the WordPress core

Most of the open source JavaScript libraries are located inside the `wp-includes` folder of your WordPress installation. Let's take a look at the following screenshot for JavaScript libraries available with WordPress:



As you can see, there is a large number of built-in libraries inside the `wp-includes/js` folder. WordPress uses jQuery for most of its core features, and hence there is a separate folder for jQuery-related libraries, for example, jQuery UI, Masonry, jQuery Form plugin, and many more. Developers can use all of these libraries inside their own plugins and themes without duplicating the files. Backbone.js and Underscore.js are the latest additions to the WordPress core. These two libraries are becoming highly popular among web developers for building modularized client-side code for large-scale applications. WordPress integration with Backbone.js and Underscore.js has been rarely explained in online resources. So we are going to look at the integration of these two libraries into the portfolio application and to identify the use of JavaScript libraries included in the WordPress core.

What is Backbone.js?

In recent months, Backbone.js has become one of the most trending open source libraries for building large-scale JavaScript-based applications. It's a light-weight library that depends on Underscore.js, and has the capability of easily integrating with libraries such as jQuery and Node.js. Let's look at the official definition from `http://backbonejs.org/` to identify its importance:

> *Backbone.js gives structure to web applications by providing* **models** *with key-value binding and custom events,* **collections** *with a rich API of enumerable functions,* **views** *with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.*

The preceding definition contains a number of important aspects required for web application development. We have already discussed the importance of separating the concerns in web application development throughout the first few chapters of this book. MVC architecture is heavily used in server-side implementation for separating the concerns. But most developers, including the experienced ones, don't use such techniques in client-side scripting, which results in code that is very hard to manage. Backbone.js provides a very flexible solution by structuring client-side code to work in the MVC type process. Even though Backbone.js is referred to as an MVC library, most of the implementations will be restricted to models and views, as the role of controllers is ambiguous. Generally, Backbone.js views play the role of controllers as well. Although it's not pure MVC, it's good enough to handle complex code structuring in the client side.

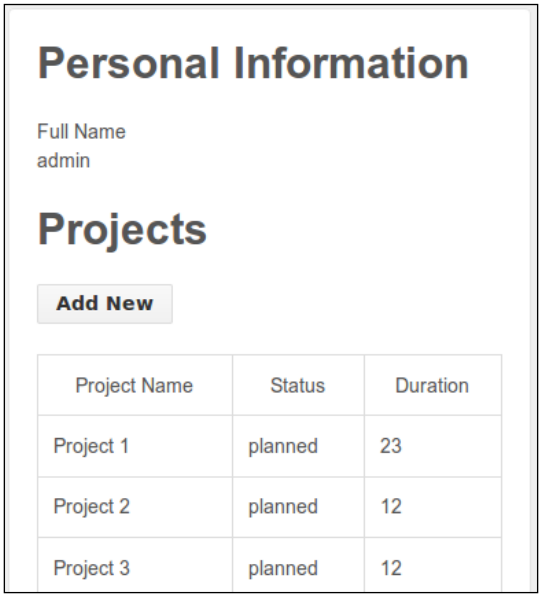
The last part of the Backbone.js definition mentions the RESTful JSON interface. REST is the acronym used for Representational State Transfer. REST is emerging as the popular architectural style and will become the trend in future with the use of JavaScript-based applications. You can learn more about RESTful architecture at:

`http://www.restapitutorial.com/lessons/whatisrest.html`

The future of web application development will heavily depend on JavaScript and HTML5 and hence, it's a must to get a head start as developers in learning these popular frameworks.

Understanding the importance of code structuring

Typically, WordPress developers tend to focus more on design aspects of the application compared to development aspects. Hence, we can find a large number of WordPress plugins with messy client-side codes filled up with jQuery events. Let's see the importance of structuring code by looking at a practical scenario. Consider the following screenshot for displaying the developer profile of the portfolio management application:



Consider the **Projects** section of the screenshot where we have a data grid generated on a page load using AJAX. There is an **Add New** button that creates new records on the database using another AJAX request. Even though it's not shown, we need the edit and delete actions in the future to complete the functionality of the grid. So let's see how developers implement these tasks with jQuery:

```
$(document).ready(function(){
  // Create the AJAX request to load the initial data
  // Generate the HTML code to update the grid
```

```
});
$("#add_btn").click(function(){
  // Create the AJAX request to save the data
  // Generate the HTML code to update the grid
});
$("#edit_btn").click(function(){
  // Create the AJAX request to update the data
  // Generate the HTML code to update the grid
});
$("#del_btn").click(function(){
  // Create the AJAX request to save the data
  // Generate the HTML code to update the grid
});
```

There are four events to implement the given tasks. As application scales, we will have hundreds of such events within the JavaScript files. The HTML code is placed all over inline with these event handling functions. It's almost impossible for another developer to identify the code required for any given screen.

Now, let's see how Backbone.js solves this issue in combination with Underscore.js. Let's take a look at the advantages of using Backbone.js, compared to the events-based structure of jQuery:

- Separates the concerns using models, views, and collections
- Template generation is done separately with Underscore.js
- Matching client-side data with server-side database models
- Organizing data into collections and synchronizing it with the server
- Ability to listen to changes in models instead of UI elements

Basically, Backbone.js offers all the features required for building scalable applications. Also, we can use other libraries with Backbone.js to cater to specific functionality such as DOM handling.

Integrating Backbone.js and Underscore.js

We started the process of integrating Backbone.js and Underscore.js to identify the use of the JavaScript libraries provided with the WordPress core. Let's begin the integration by loading these libraries into WordPress. The following code illustrates how we can include these existing libraries into plugins:

```
function include_scripts() {
  wp_enqueue_script('backbone');
}
add_action('wp_enqueue_scripts', 'include_scripts');
```

The preceding code loads the Backbone.js library on the frontend of the application with the necessary dependencies such as jQuery and Underscore.js. We can load all the other libraries with a similar technique. Take a look at the available JavaScript libraries by visiting the *Default Scripts Included and Registered by WordPress* section of `http://codex.wordpress.org/Function_Reference/wp_enqueue_script`. You can use the value of the **Handle** column as the parameter for the `wp_enqueue_script` function to load the library into the plugin.

Creating a developer profile page with Backbone.js

We looked at the basic structure for a developer's profile page at the beginning of the *Understanding the importance of code structuring* section. In this section, we are going to implement the mentioned tasks with the use of Backbone.js and Underscore.js. As usual, we have to start the process by creating another plugin for this chapter. Create a folder named `wpwa-open-source` inside the `plugins` folder, and define the main plugin file as `class-wpwa-open-source.php`.

Our first task will be to load the personal information for developers. If you were following each chapter of this book with the practical examples, you will understand that we have a problem at this stage. Take a moment to think about and figure out the problem.

Let's see what went wrong in our implementation. Here, we need to create another frontend template for the developer's profile page. In the previous chapter, we created a custom template loader that catered to theme-related tasks. The template loader file (`class-wpwa-template-loader.php`) and `templates` folder are placed inside the `wpwa-theme` plugin, and hence cannot be re-used inside the `wpwa-open-source` plugin.

This is the common trend among WordPress developers, where they activate a bunch of third-party plugins without knowing the actual contents. Throughout this book, I have intentionally used this approach to illustrate the common pitfalls in large-scale application development. At this stage, we have no option other than to create a duplicate template loader. In the final chapter, we are going to integrate all the plugins developed throughout this book into a single plugin to illustrate the best practices for structuring large applications.

Let's start by creating the necessary action hooks to handle the custom routing for this plugin. Include the following code inside the plugin's constructor:

```
register_activation_hook(__FILE__, array($this,
  'flush_application_rewrite_rules'));
add_action('init', array($this, 'manage_routes'));
add_filter('query_vars', array($this,
  'manage_routes_query_vars'));
```

These three lines of code should be familiar to you as we have used them in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*. Here, we have some code duplication due to individual plugins. Consider the following code for implementation of these three functions:

```
public function flush_application_rewrite_rules() {
  $this->manage_routes();
  flush_rewrite_rules();
}
public function manage_routes_query_vars($query_vars) {
  $query_vars[] = 'record_id';
  return $query_vars;
}
public function manage_routes() {
  add_rewrite_rule('^user/([^/]+)/([^/]+)/?',
    'index.php?control_action=$matches[1]&record_id=$matches[2]',
    'top');
}
```

In this scenario, we need another rewrite rule for working with the ID parameter. Hence we have introduced another rule to match `user/([^/]+)/([^/]+)`. The first parameter will take the control action, while a second parameter is used for the ID of the developer. Also, we have added another query variable named `record_id` for handling the developer IDs. This rule can be re-used across all the edit or load functions by ID.

The developer's profile can be accessed by using `/user/profile/ID`, where ID is the unique ID in the `wp_users` table.

Next, we have to load the profile page through a custom controller, which again will be duplicated for this plugin. Update the constructor to include the following action for redirecting templates:

```
add_action('template_redirect', array($this, 'front_controller'));
```

Now we can look at the following code for the initial loading of the developer profile page:

```
public function front_controller() {
  global $wp_query, $wp;
  $control_action = $wp_query->query_vars['control_action'];
  $record_id = $wp_query->query_vars['record_id'];

  switch ($control_action) {
    case 'profile':

      $developer_id = $record_id;
      $this->create_developer_profile($developer_id);
      break;
  }
}
```

This is the same technique we used in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*, for working with custom routing. Here, we have an additional query parameter for the developer ID. Finally, we can look at the `create_developer_profile` function for generating the developer profile data:

```
public function create_developer_profile($developer_id) {
  $user_query = new WP_User_Query(array('include' =>
    array($developer_id)));
  $data = array();
  foreach ($user_query->results as $developer) {
    $data['display_name'] = $developer->data->display_name;
  }

  $current_user = wp_get_current_user();

  $data['developer_status'] = (
    $current_user->ID == $developer_id);
  $data['developer_id'] = $developer_id;

  $tmp = new WPWA_Template_Loader_Duplicate();
  $tmp->render("developer", $data);
  exit;
}
```

First, we use the `WP_User_Query` class to get the profile details of the developer. At this stage, we only have the name of the developer in the profile. So we assign the name to the `$data` array to be passed into the template.

In the final chapter, we will be updating the developer's profile with additional information to be displayed in the frontend.

Next, we check whether the developer of this profile is logged into the application, to show or hide the **Add New** button on the **Projects** section. Finally, we render the developer template with data passed to the function. In order to complete the initial page loading, we need to execute the following tasks:

1. Create a duplicate template loader as `WPWA_Template_Loader_Duplicate` inside the `plugins` folder.
2. Create a folder named `templates` inside the `plugins` folder.
3. Create the `developer.php` template inside the `templates` folder.

Here are the initial contents of the `developer.php` template:

```
<?php get_header(); ?>
<div class='main_panel'>
  <div class='developer_profile_panel'>
    <h2>Personal Information</h2>
    <div class='field_label'>Full Name</div>
      <div class='field_value'><?php echo
         esc_html($data['display_name']); ?>
      </div>
    </div>
  </div>
<?php get_footer(); ?>
```

At this stage, our developer profile seems pretty simple as we only have one field to display. Now we come to the complex part of the template, where we load the projects dynamically.

Update the constructor with the `wp_enqueue_scripts` action to include Backbone.js in the plugin, as illustrated in the following code:

```
public function include_scripts() {
  wp_register_script('developerjs', plugins_url(
    'js/wpwa-developer.js', __FILE__), array('backbone'));
    wp_enqueue_script('developerjs');

  $config_array = array(
```

```
        'ajaxUrl' => admin_url('admin-ajax.php'),
        'developerID' => $developer_id
      );

      wp_localize_script('developerjs', 'wpwaScriptData',
        $config_array);
    }
```

We create a new JavaScript file named `wpwa-developer.js` inside the `wpwa-open-source/js` folder by providing the dependent library as Backbone.js. Now you will have both Backbone.js and Underscore.js included. Afterwards, we use the `wp_localize_script` function to pass the WordPress AJAX URL into the `wpwa-developer.js` file.

Structuring with Backbone.js and Underscore.js

Here, we come to the most exciting part of working with Backbone.js inside WordPress. Defining the models, views, and collections are a major part of working with Backbone.js. Let's plan the structure before we get into the implementation.

The developer profile should contain the list of projects created by the developer. So the *model* in this scenario is the project and *collection* will be the projects. The list of projects needs to be loaded as a dynamic table, and hence we need a *view* for the project list. First, we need to define these three components to get started with Backbone.js. Let's start with the creation of model as shown in the following code:

```
$jq =jQuery.noConflict();
$jq(document).ready(function(){
  var Project =  Backbone.Model.extend({
    defaults: {
      name: '',
      status: '',
      duration:'',
      developerId : ''
    },
  });
});
```

Here, we have a very simple model named `Project` for working with projects in the portfolio application. The details of the projects have been limited to name, status, and duration to simplify the explanations. Next, we can look at the collection of projects using the following code:

```
var ProjectCollection = Backbone.Collection.extend({
  model: Project,

  url: pwaScriptData.ajaxUrl
    +"?action=wpwa_process_projects&developer_id="
    +wpwaScriptData.developerID
});
```

Backbone.js uses collections to store lists of models for listening to changes in specific attributes. So, we have assigned the `Project` model to the `ProjectCollection` collection. Next, we have to define a URL for working with the model data from the server. Generally, this URL will be used to save, fetch, delete, and update data on the server.

These requests work in a RESTful manner. With WordPress custom routing, it's difficult to take advantage of RESTful requests in its purest form. We are going to use the WordPress AJAX handler URL to manipulate the various requests from models. Therefore, we have used the AJAX handler URL with an action named `wpwa_process_projects`, which will be responsible for handling all the requests for the `Project` model.

This is not the place for learning the basics of Backbone.js. So, I suggest you familiarize yourself with the basic concepts of Backbone.js using the official documentation at:

`http://backbonejs.org`

Now let's look at the view for displaying the project list for developers:

```
var projectsList;
var ProjectListView = Backbone.View.extend({
  el: $jq('#developer_projects'),
  initialize: function () {
    projectsList = new ProjectCollection();
  }
});

var projectView  = new ProjectListView();
```

Finally, we have the Backbone.js view for generating template data to the user screen. Here, you can see that initialization of these components is handled by creating a new object of the view class. Therefore, we can assume that the view acts as the controller on most occasions.

The main container of the view is defined by the `el` attribute. Then we initialized the collection of projects using the `ProjectCollection` class. Now let's take a look at the remaining sections of the `developer.php` file for understanding the view. The following code is included after the profile information section:

```
<div id='developer_projects'>
  <h2>Projects</h2>
  <div >
    <table id='list_projects'>

    </table>
  </div>
</div>
```

As defined in the view, this will be the main container used for displaying the developers list. Now we have the definition of all the Backbone.js components required for this scenario.

Displaying the projects list on page load

Once the page load is completed, we need to fetch the projects from the server to be displayed on the profile page. So let's update the view with the necessary functions as shown in the following code:

```
var ProjectListView = Backbone.View.extend({
    el: $jq('#developer_projects'),
    initialize: function () {
      projectsList = new ProjectCollection();
      projectsList.bind("change", _.bind(this.getData, this));
      this.getData();
    },
    getData: function () {
      var obj = this;
      projectsList.fetch({
        success: function () {
          obj.render();
        }
      });
    },
});
```

Inside the `initialize` function, we bind an event named `change` for the `projectsList` collection, to call a function named `getData`. This event will get fired whenever we change the items in the collection. Next, we call the `getData` function inside the `initialize` function.

The `getData` function is responsible for retrieving projects from the server. So we call the `fetch` function on the `projectsList` collection to generate a GET request to the server. Since the request is asynchronous, we have to wait till the `success` function is fired before proceeding with the callback to fetch projects. Once the request is completed, the `projectsList` collection will be populated with the list of projects from the server. Finally, we execute the `render` function to load the templates.

We have to understand the server-side implementation of this request before moving into the `render` function. So update the plugin constructor by adding the following action to enable AJAX requests on projects:

```
add_action('wp_ajax_nopriv_wpwa_process_projects', array($this,
  'process_projects'));
add_action('wp_ajax_wpwa_process_projects', array($this,
  'process_projects'));
```

Afterwards, we can look at the following code for the implementation of the `process_projects` function:

```
public function process_projects() {
  $request_data = json_decode(file_get_contents("php://input"));
  $project_developer = isset ($_GET['developer_id']) ?
    $_GET['developer_id'] : '0';
  if (is_object($request_data) && isset ($request_data->name)) {
   // Saving and updating models
  }
  else {
    $result = $this->list_projects($project_developer);
    echo json_encode($result);
    exit;
  }
}
```

All the requests to the server will be made by Backbone.js in a RESTful manner. So we have to use the PHP input stream accessing techniques to get the data passed by Backbone.js. Here, we have used `php://input`, which allows us to read the raw data from the request body.

The `php://input` stream is a read-only stream that allows you to read raw data from the request body. In the case of POST requests, it is preferable to use `php://input` instead of `$HTTP_RAW_POST_DATA` as it does not depend on the special `php.ini` directives. Moreover, for those cases where `$HTTP_RAW_POST_DATA` is not populated by default, it is a potentially less memory intensive alternative to activating `always_populate_raw_post_data`. The `php://input` stream is not available with `enctype="multipart/form-data"`. More information on accessing various input/output streams can be found at:

`http://php.net/manual/en/wrappers.php.php`

Backbone's `fetch` function uses `GET` request with no parameters, and hence the `$request_data` variable will be empty. So the `else` part of the code will be invoked to call the `list_projects` function, to generate the projects list as shown in the following code:

```
public function list_projects($developer_id) {
  $projects = new WP_Query(array('author' => $developer_id,
    'post_type' => 'wpwa_project', 'post_status' => 'publish',
    'posts_per_page' => 15, 'orderby' => 'date'));
  $data = array();
  if ($projects->have_posts()) : while (
    $projects->have_posts()) : $projects->the_post();
  $post_id = get_the_ID();
  $status = get_post_meta($post_id, '_wpwa_project_status', TRUE);
  $duration = get_post_meta($post_id, '_wpwa_project_duration',
    TRUE);
  array_push($data, array("ID" => $post_id,
    "name" => get_the_title(), "status" => $status,
     "duration" => $duration));
  endwhile;
  endif;
  return $data;
}
```

The latest projects of the specified developer are retrieved using the `WP_Query` class. Each project is set up as an array to be used as a model from the client side. Having completed the server-side code, now we can move back to the render function of the view as shown in the following code:

```
render: function () {
  var template_data = _.template($jq(
    '#project-list-template').html(), {
    projects: projectsList.models
```

```
    });
    var header_data = $jq('#project-list-header').html();
      $jq(this.el).find("#list_projects")
      .html(header_data+template_data);
    return this;
  }
```

We start the `render` function by loading a template named `#project-list-template` using the Underscore.js template system. The data returned from the `fetch` request is passed into a variable named `projects`. It's hard to understand the rest of the render function without looking at the template. Let's take a look at the two templates stored inside the `wpwa-developer.php` file.

```
<script type="text/template" id="project-list-template">
  <% _.each(projects, function(project) { %>
    <tr class="project_item">
      <td><%= project.get('name') %></td>
      <td><%= project.get('status') %></td>
      <td><%= project.get('duration') %></td>
    </tr>
  <% }); %>
</script>
<script type="text/template" id="project-list-header">
    <tr >
      <th>Project Name</th>
      <th>Status</th>
      <th>Duration</th>
    </tr>
</script>
```

Using `text/template` as `script type` is a common way of defining templates inside the HTML files. The browser doesn't execute these client-side scripts due to the `text/template` type. Here, we have two templates, where the first one generates the list of projects using the Underscore.js template variables. We can access the `project` variable passed from the `render` function to generate the list. The data from the Backbone.js models can be accessed using the `get` function by providing the necessary attributes. The second template is used for the table header of the projects list.

Now let's get back to the `render` function. The `render` function loads both of the preceding templates with the necessary data. Then we use the `el` element of the view to find the `#list_projects` table, and populate the output generated from the templates as the content using the `html` function. Now we will get the data grid populated with the project list on the page upload. So far, we have discussed the usage of the Backbone.js model, view, and collection. Next, we need the ability for the developer to create new projects from the frontend.

Creating new projects from the frontend

The developer profile page should be visible to all types of users, including the ones who are not logged in. So far, we have displayed the project list on the initial page load. The developer of the profile page should be able to add, edit, or delete projects from the frontend after logging in to the application. This improves the user experience by avoiding the need to switch between the backend and frontend to update and preview data. Here, we are going to implement the project creation from the frontend. Therefore, the logged in developer should get an **Add New** button on top of the project list, as shown in the following screenshot:

Let's update the `developer.php` template to include the **Add New** button as illustrated in the following code:

```
<div id="msg_container"></div>
<?php if ($data['developer_status']) {   ?>
   <input type='button' id="add_project" value="Add New" />
<?php } ?>
```

The preceding code is placed after the Projects heading in the `#developer_projects` container. We want the button to be displayed only for the owner of the profile. So we use the `developer_status` value passed from the `create_developer_profile` function. Once the button is clicked, the developer should get a form for saving new projects.

Initially, the form is hidden and will be displayed on the click event of the button. So let's add the hidden form to the `developer.php` template after the preceding code:

```
<div id='pro_add_panel' style='display:none' >
  <div class='field_row'>
    <div class='field_label'>Project Name</div>
```

```
        <div class='field_value'><input type='text'
          id='pro_name' /></div>
      </div>
    <div class='field_row'>
      <div class='field_label'>Status</div>
      <div class='field_value'><select id="pro_status">
        <option value="0">Select</option>
        <option value="planned">Planned</option>
<option value="pending">Pending</option>
                        <option value="failed">Failed</option>
                        <option value="completed">Completed</option>
                    </select></div>
            </div>
            <div class='field_row'>
                <div class='field_label'>Duration</div>
                <div class='field_value'><input type='text' id='pro_
duration' /></div>
            </div>
            <div class='field_row'>
                <div class='field_label'></div>
                <div class='field_value'><input type='button' id='pro_
create' value='Save' /></div>
            </div>
</div>
```

The CSS `display:none` attribute is used to hide the `pro_add_panel` container on the initial page load. Inside the container, we have three fields for project name, status, and duration with a save button called `#pro_create`. Now we have to display the form on clicking the **Add New** button. Generally, we use a jQuery event handler on the button to cater to such requirements. However, we already looked at how the code becomes hard to understand with the usage of jQuery events. So, we are going to use Backbone.js events to structure the code properly.

Integrating events to the Backbone.js views

Backbone.js allows you to define events on each view, making it possible to restrict the scattering of events. This technique allows developers to quickly understand the events used for any given screen without having to search through all the code. Here, we need two events to display the project creation form and submit the data to the server. Let's add the following code to the `ProjectListView` created previously:

```
events: {
  'click #add_project': 'addNewProject',
  'click #pro_create': 'saveNewProject'
},
```

```
addNewProject: function(event) {
  $jq("#pro_add_panel").show();
}
```

As you can see, all the events are separated into a section named `events` inside the view. We have used a click event on `#add_project` and `#pro_create` to call the `addNewProject` and `saveNewProject` functions respectively. The `addNewProject` function uses jQuery's `show` function to make the project creation form visible to the developer. Once the button is clicked, your screen should look like the following screenshot:

We used several rewriting rules, similar in structure throughout this book. So it's important to keep the rewrite rules in the proper order to get the desired results. We can use the **Rewrite Rules Inspector** plugin to flush the rules after making updates to code.

Next, we need to concentrate on the saving and validating process of new projects.

Validating and creating new models on the server

Form validation is very important in web development to avoid harmful invalid data, and to keep the consistency in the database. Backbone.js automatically calls a validation function on the execution of the create function on a collection. Let's add the `validate` function to the `Project` model with basic validations on name, status, and duration as shown in the following code:

```
var Project = Backbone.Model.extend({
```

```
    defaults: {
      name: '',
      status: '',
      duration:''
    },
    validate: function(attrs) {
      var errors = this.errors = {};
      if (!attrs.name)
        errors.name = 'Project name is required';
      if (attrs.status == 0)
        errors.status = 'Status is required';
      if (!attrs.duration)
        errors.duration = 'Duration is required';
      if (!_.isEmpty(errors)){
        return errors;
      }
    }
  });
```

The `validate` function is automatically executed before saving data to the server, by passing the attributes of the model as a parameter. We can execute the necessary validation within this function and return the errors as an object.

Creating new models on the server

This is the final section of the process for saving new projects to the database. Now we have to implement the `saveNewProject` function defined in the events section, as shown in the following code:

```
  saveNewProject: function(event) {
    var options = {
      success: function (response) {
        console.log(response);
      },
      error: function (model, error) {
        console.log(error);
      }
    };
    var project = new Project();
    var name = $jq("#pro_name").val();
    var duration = $jq("#pro_duration").val();
    var status = $jq("#pro_status").val();
    var developerId = $jq("#pro_developer").val();
```

```
    projectsList.add(project);

    projectsList.create({
      name: name,
      duration:duration,
      status : status,
      developerId : developerId
    },options);
  }
```

First, we have a variable named `options` for defining the `success` and `failure` functions for the project creation. Here, we are logging the result values to the browser console. In real implementations, the result should be displayed to the user as a message.

Next we create a new object of the `Project` model by passing the data retrieved from the form fields. Later, the newly created model is assigned to the original projects collection retrieved in the initial page load. Remember that we defined the following line of code while implementing the initialize function of the view:

```
projectsList.bind("change", _.bind(this.getData, this));
```

This event gets fired whenever the items in the collection are changed. Here, we have assigned a new model to the collection and hence this event will get fired. So, the project list will be updated without refreshing the page to contain the new model. Finally, we execute the `create` function on the collection to save the new project to the database. This will generate a POST request to the AJAX action handler named `wpwa_process_projects`. Finally, we complete the process by implementing the rest of the `process_projects` function as shown in the following code:

```
public function process_projects() {
  $request_data = json_decode(file_get_contents("php://input"));
  $project_developer = isset ($_GET['developer_id']) ?
    $_GET['developer_id'] : '0';
  if (is_object($request_data) && isset($request_data->name)) {
    $project_name = $request_data->name;
    $project_status = $request_data->status;
    $project_duration = $request_data->duration;
    $project_developer = $request_data->developerID;
    $err = FALSE;
    $err_message = '';
    if ($project_name == '') {
```

```
        $err = TRUE;
        $err_message .= 'Project name is required.';
      }
      if ($project_status == '0') {
        $err = TRUE;
        $err_message .= 'Status is required.';
      }
      if ($project_duration == '') {
        $err = TRUE;
        $err_message .= 'Duration is required.';
      }
      if ($err) {
        echo json_encode(array('status'=>'error',
          'msg'=> $err_message));
        exit;
      } else {
        $current_user = wp_get_current_user();
        $post_details = array(
          'post_title' => esc_html($project_name),
          'post_status' => 'publish',
          'post_type' => 'wpwa_project',
          'post_author' => $current_user->ID
        );
        $result = wp_insert_post($post_details);
        if (is_wp_error($result)) {
          echo json_encode(array('status'=>'error',
            'msg'=> $result));
        } else {
          update_post_meta($result, "_wpwa_project_status",
            esc_html($project_status));
          update_post_meta($result, "_wpwa_project_duration",
            esc_html($project_duration));
           echo json_encode(array('status'=>'success'));
        }
      }
    exit;
    } else{
      $result = $this->list_projects($project_developer);
      echo json_encode($result);
      exit;
    }
  }
```

Earlier, we discussed the initial request data gathering technique and the `else` statement while fetching data from the server. Now we are going to look at the `if` statement for saving new project data. We can get the project data from the `$request_data` object. It's important to validate data in both the client side as well as the server side. Earlier, we implemented the client-side validation using Backbone.js. Here, we have used the server-side validations for all the input parameters. In case validation errors are generated, we pass a JSON array containing the status and the error message.

When the input date is successfully validated, we create a new project using the `wp_insert_post` function. Once the project is successfully saved, we can execute the `update_post_meta` function to save additional metadata.

So we have come to the end of a long process for identifying the basic usage of Backbone.js and Underscore.js in the WordPress frontend. In the last part, we learned the uses of Backbone.js create and validate functions plus their way of handling effects.

Keep in mind that edit and delete functionality will also go through the `if` statement of the `process_projects` function. We need to figure out a way to separate these functions by filtering the request data send on create, update, and delete events.

Now you should be able to understand the value of Backbone.js for creating well-structured client-side code. In the next section, we will be looking at the integration of existing PHP libraries inside WordPress.

Using PHPMailer for custom e-mail sending

In the previous section, we looked at the use of Backbone.js and Underscore.js to identify the usage of open source JavaScript libraries within the WordPress core. Now it's time to identify the use of open source PHP libraries within the WordPress core, so we are going to choose **PHPMailer** among a number of other open source libraries. PHPMailer is one of the most popular e-mail sending libraries, used inside many frameworks as a plugin or library. This library eases the complex tasks of creating advanced e-mails with attachments and third-party account authentications.

PHPMailer has been added to GitHub for improving its development process. You can get more information about this library at:

`https://github.com/PHPMailer/PHPMailer`

WordPress has a copy of this library integrated into the core for all e-mail-related tasks. We can find the **PHPMailer** library inside the `wp-includes` folder within the file named `class.phpmailer.php`.

Usage of PHPMailer within the WordPress core

Basically, the functionality of this library is invoked using a common function named `wp_mail`, located at the `pluggable.php` file inside the `wp-includes` folder. This function is used to handle all the e-mail sending tasks within WordPress. The `wp_mail` function is very well structured to cater for various functionalities provided by PHPMailer. Customizations to the existing behavior of this function can be provided through various types of hooks as listed here:

- `wp_mail_from`: Changes the e-mail of the sender, which defaults to `wordpress@{sitename}`
- `wp_mail_from_name`: Changes the name of the sender, which defaults to `WordPress`
- `wp_mail_content_type`: Changes the e-mail content type, which defaults to `text/plain`
- `wp_mail_charset`: Changes the e-mail character set, which defaults to `charset` assigned in settings

In web applications, we might need advanced customizations, which go beyond the capabilities of these existing hooks. In such situations, we have to use a customized e-mail sending functionality. The existing PHPMailer library can be easily used to create custom versions of e-mail sending functionalities. There are two ways of creating custom e-mail sending functionality, listed as follows:

- Creating a custom version of the pluggable `wp_mail` function
- Loading PHPMailer inside plugins and creating custom functions

Creating a custom version of the pluggable wp_mail function

We briefly introduced the pluggable functions in *Chapter 5*, *Developing Pluggable Modules*. These functions are located inside the `pluggable.php` file, which is inside the `wp-includes` folder. As developers, we can override the existing behavior of these functions by providing a custom implementation. `wp_mail` is a pluggable function and hence, we can create custom implementations within plugins. We just duplicate the contents of the existing function within a plugin, and change the code where necessary to provide custom features.

Loading PHPMailer inside plugins and creating custom functions

This is a straightforward task, where we have to include the files and initialize the PHPMailer class as shown in the following code:

```
require_once ABSPATH . WPINC . '/class-phpmailer.php';
require_once ABSPATH . WPINC . '/class-smtp.php';
$mailer = new PHPMailer( true );
```

Then we can use the `$mailer` variable object to configure and send e-mails as described in the official documentation.

By default, WordPress uses the default mail server of your web host to send e-mails. This doesn't involve sender e-mail authentication and becomes the reason behind spam e-mails.

We can prevent e-mail spamming by authenticating the user and using SMTP to e-mail, so we are going to implement the custom PHPMailer function while implementing the subscriber notification process of the portfolio management application. Let's list the tasks for this process:

- Create a custom function to use PHPMailer
- Send e-mails in SMTP using an authenticated account
- Retrieve the subscribers from the database
- Send notifications on new projects, articles, books, and services

We can start the process by including the following action inside the constructor of the `WPWA_Open_Source` class inside the `wpwa-open-source` plugin:

```
add_action('new_to_publish', array($this,
  'send_subscriber_notifications'));
add_action('draft_to_publish', array($this,
  'send_subscriber_notifications'));
add_action('pending_to_publish',array($this,
  'send_subscriber_notifications'));
```

Here, we have used three actions related to post status transfer. Subscribers should get notifications about all the projects, articles, books, and services, once they are published. So we use the preceding post status transfer actions to execute a custom function on the publish status change. Let's look at the `send_subscriber_notifications` function for sending e-mail notifications with PHPMailer:

```
public function send_subscriber_notifications($post) {
  global , $wpdb;

    $permitted_posts  = array('wpwa_book','wpwa_project',
     'wpwa_services', 'wpwa_article');
    if (in_array($_POST['post_type'],$permitted_posts)) {

     require_once ABSPATH . WPINC . '/class-phpmailer.php';
     require_once ABSPATH . WPINC . '/class-smtp.php';
     $phpmailer = new PHPMailer(true);

     $phpmailer->From = "exaple@gmail.com";
     $phpmailer->FromName = "Portfolio Application";
     $phpmailer->SMTPAuth = true;
     $phpmailer->IsSMTP(); // telling the class to use SMTP
     $phpmailer->Host = "ssl://smtp.gmail.com"; // SMTP server
     $phpmailer->Username = "example@gmail.com";
     $phpmailer->Password = "password";
     $phpmailer->Port = 465;
     $phpmailer->addAddress('admin@example.com', 'Admin');

     $phpmailer->Subject = "New Activity on Portfolio
       Application";
     $sql = "SELECT user_nicename,user_email
       FROM $wpdb->users
       INNER JOIN " . $wpdb->prefix . "subscribed_developers
       ON " . $wpdb->users . ".ID = " . $wpdb->prefix .
       "subscribed_developers.follower_id
       WHERE " . $wpdb->prefix .
       "subscribed_developers.developer_id =
       '$post->post_author'";
       $subscribers = $wpdb->get_results($sql);
       foreach ($subscribers as $subscriber) {
         $phpmailer->AddBcc($subscriber->user_email,
         $subscriber->user_nicename);
       }
       $phpmailer->Body = "New Update from your favorite
         developers " . get_permalink($post->ID);
       $phpmailer->Send();
    }
}
```

We start the `send_subscriber_notifications` function by filtering necessary post types to prevent code execution for unnecessary post types. Next, the PHPMailer and SMTP class will be included to load the library as discussed in the earlier section. Once the `$phpmailer` object is created, we can define the necessary parameters for sending e-mails. We start by defining them from the e-mail and display name.

Next, we have configured the SMTP settings for e-mail authentication and sending through a custom SMTP server. We define the SMTP authentication by using `true` for the `SMTPAuth` parameter. Then we have to define the host, username, password, and port of the SMTP account that will be used to authenticate e-mails.

Next, we need to retrieve the subscribers of the developer of the published book, article, project, or service. Custom SQL query is used to retrieve the respective subscribers from the database. Then we add the e-mail of each subscriber into the `$phpmailer` object, while looping through the subscribers. Here, we need to use the `AddBcc` function to keep the confidentiality of e-mail addresses. Finally, we send the notification with common e-mail content using the permalink of the published post, so we have a very basic notification system for subscribers of the application.

This technique works fine for basic scenarios where we have a limited number of subscribers. In situations where we have a large number of subscribers, we can't use this technique as it will delay the publishing of post. So let's look at other possible solutions for these situations:

- We can track the latest published posts in a separate database table and schedule a cron job for sending notifications periodically.
- We can create a developer specific RSS feed with a custom feed URL. Subscribers can then use third-party e-mail services to get notifications on feed updates.

Here, we are not going to implement these techniques as they are beyond the scope of this chapter. You can look at the book's website for guides on using the preceding techniques.

Implementing user authentication with Opauth

Logging in with open authentication has become a highly popular method among application users, as it provides quicker authentication compared to the conventional registration forms. So many users prefer the use of social logins to authenticate themselves and try the application before deciding to register.

OpenAuth is becoming the standard third-party authentication system for providing such functionality. Most existing web applications offer the user authentication using OpenAuth and hence, it's important to know how we can integrate OpenAuth login into WordPress. Here, we are going to build a plugin that lets users log in and register through popular social networking sites such as Twitter, Facebook, and LinkedIn.

We will be using **Opauth**, which is a library that standardizes the open authentication strategies among all the providers. First, download a copy of the library from `http://opauth.org/`. We have the choice of downloading the core files or downloading the library with examples. I suggest you download Opauth bundled with the examples package.

Now let's create a new plugin folder, `wpwa-opauth`, with the main plugin file as `class-wpwa-opauth.php`. Next, you can extract the contents of the Opauth library directly into the `wpwa-opauth` folder. Now we are ready to implement the OpenAuth login integration for our portfolio application. This portfolio application will contain the OpenAuth login using Twitter, Facebook, and LinkedIn. So we need three links or buttons just under our login screen.

Basically, we have two ways of assigning these links in the login screen:

- Directly embed the HTML code under the default login button
- Define an action hook and implement the hook within a plugin

Even though both techniques do the same job, we have more of an advantage in choosing the action hooks technique, as it allows us to add or remove any number of login-related components without affecting existing functionality. So we have to modify the template used for the login in the `wpwa-user-manager` plugin created in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*. Open the `login.php` file inside the `templates` folder of the `wpwa-user-manager` plugin and modify the code as follows:

```
<form method="post" action="<?php echo site_url(); ?>/user/login"
id="login_form" name="login_form">
  <!-- Rest of the HTML fields -->
  <li>
    <label class="frm_label" > </label>
    <input  type="submit"  name="submit" value="Login" />
  </li>
  <?php do_action('wpwa_login_addons'); ?>
  </ul>
</form>
```

Here, we execute an action named `wpwa_login_addons` with the `do_action` function. This allows you to add dynamic content to the login screen using plugins. Next, we have to generate the necessary login links to populate the `wpwa_login_addons` area. Open the `class-wpwa-opauth.php` file and include the following code to get things started:

```
<?php
// Plugin definition using comments
class WPWA_Opauth {
  public function __construct() {
    add_action('wpwa_login_addons', array($this, 'login_addons'));
  }
  public function login_addons() {
    echo '<li><a href="facebook">Facebook</a></li>
    <li><a href="twitter">Twitter</a></li>
    <li><a href="linkedin">LinkedIn</a></li>';
  }
}
$opauth = new WPWA_Opauth();
```

We have implemented the `wpwa_login_addons` action with the use of the `login_addons` function inside the `WPWA_Opauth` class. The simplicity of the HTML code led me to write the links using an `echo` statement. Ideally, we should be using template parts for loading the display code. The `href` attribute contains direct keywords instead of properly formatted URL. The reasons for including such keywords will be explained in the next few steps.

Configuring login strategies

Opauth defines each OpenAuth provider as a strategy. We have three strategies for the login process. We need to create apps for each of these strategies and get the necessary keys. Open the `opauth.conf.php` file to get a better understanding about the strategy configurations. The following code shows the default strategy definitions:

```
'Strategy' => array(
  // Define strategies and their respective configs here
  'Facebook' => array(
    'app_id' => 'YOUR APP ID',
    'app_secret' => 'YOUR APP SECRET'
  ),
  'Google' => array(
    'client_id' => 'YOUR CLIENT ID',
    'client_secret' => 'YOUR CLIENT SECRET'
   ),
```

```
        'Twitter' => array(
          'key' => 'YOUR CONSUMER KEY',
          'secret' => 'YOUR CONSUMER SECRET'
         ),
    ),
```

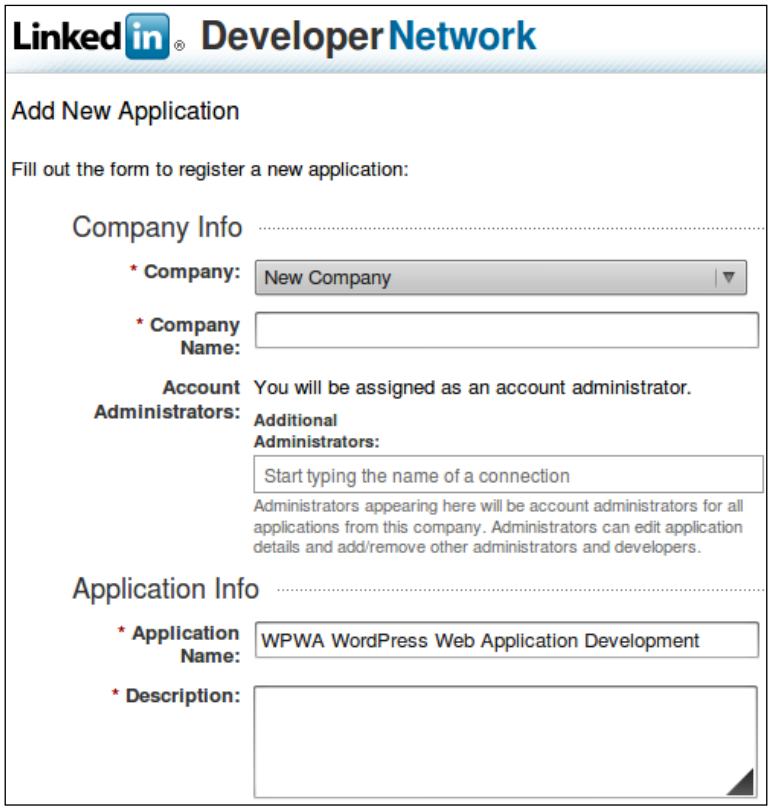
Each strategy has a specific app key and ID to authenticate the application. We need to create the apps and get the respective keys assigned to the Opauth configurations. Now let's build an app to generate these keys for strategies. App creation for the strategies mentioned here is explained commonly in many online resources. Hence, we are going create a LinkedIn app to define a new strategy.

Those who are not familiar with the creation of Twitter, Facebook, and Google+ apps can use the code bundle for this chapter available at the Packt Publishing website for step-by-step app creations for each of these strategies.

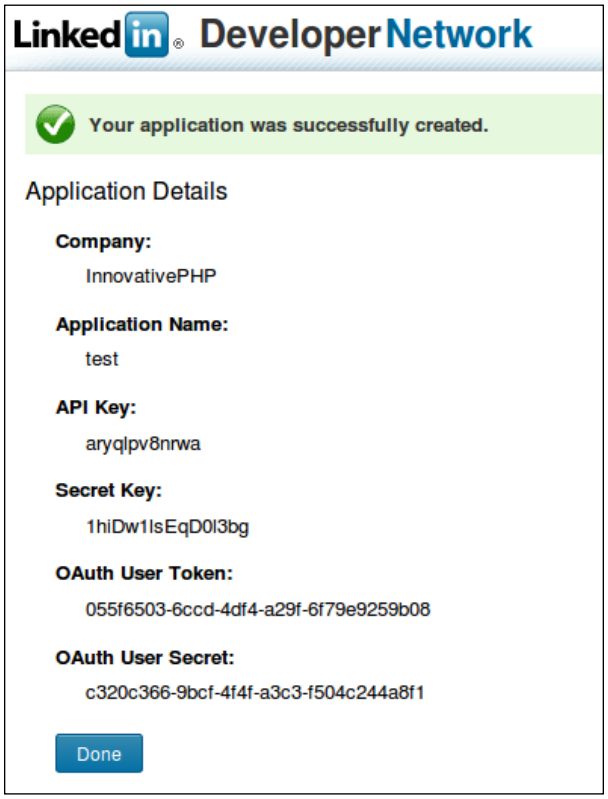
Building a LinkedIn app

First, you have to log in to your existing LinkedIn account. Then visit the developer section using `https://www.linkedin.com/secure/developer`. You will get a screen similar to the following screenshot with the existing apps, if there are any:

You can click on the **Add New Application** link that will result in the following screenshot:



Make sure you fill out all the mandatory values in the given form. Choose the permissions in the **Default Scope** section as appropriate to meet your application requirements. Once you click on the **Add Application** button, you will get a screen similar to the following screenshot with all the application-specific details, including **API Key** and **Secret Key**.



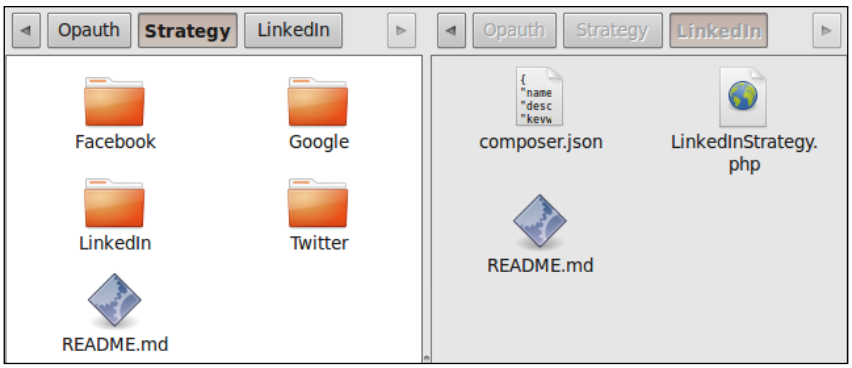
Now we can move back to the configuration file of Opauth and include the LinkedIn strategy as shown in the following code:

```
'LinkedIn' => array(
  'key1' => 'aryqlpv8nrwa',
  'key2' => '1hiDw1lsEqD0l3bg''
),
```

Here, we have added the `LinkedIn` strategy with the API key as `key1` and the secret key as `key2`. The next question is how to find the proper values for `key1` and `key2` as those keys vary based on the strategy. So we need to look for the LinkedIn strategy, which we haven't downloaded yet.

Opauth, bundled with the examples package contains Facebook, Twitter, and Google+ strategies by default and you can find them under the `lib/Opauth/Strategy` folder.

Go back to `http://opauth.org/`, and visit the *Available strategies* section. There you can find all the available strategies, which are compatible with the Opauth library. Now download the ZIP file for the LinkedIn strategy, and extract it into a folder named `LinkedIn` inside the `lib/Opauth/Strategy` folder. Your folder structure should look similar to the following screenshot, where the left pane shows the available strategies and right pane shows the contents of the LinkedIn strategy.



The preceding screenshot shows three folders for `Facebook`, `Twitter`, and `Google`. You can include these sources using the same technique by downloading the strategy folders from `Opauth.org` or creating your own strategies.

Now we can look at the remaining configurations before moving on to the login implementation. Open the `opauth.conf.php` file and find the `path` and `callback_url` options. The configuration value for `path` will be used to access Opauth, where you need to pass various strategies. So your path should be configured similarly to the following code:

```
'path' => '/user/login/',
```

According to the preceding definition, Opauth requests will be accessed with the path `/user/login/` from the root folder of your website. The sample URL might look like the following:

```
http://www.yoursite.com/user/login
```

Opauth can be accessed with a sublevel URL like `http://www.yoursite.com/login/opauth`, by configuring `/login/opauth/` as the path. Finally, if you are working on localhost with a folder named `yoursite`, the path should be `/yoursite/opauth/`.

Keep in mind that this path is a conceptual URL and has no relation to the files of the library. Hence you can use whatever you prefer as the path.

Next, we have the `callback_url` option which defines the response handling URL, once the user authenticates the application and returns to the application. This can be configured in a similar way to the path option. Here, we will be using `/opauth_success/` as `callback_url`.

Finally, you have to define a unique application-specific value for the `security_salt` option. Now, we have a fully configured version of the Opauth library for integration.

Process of requesting the strategies

We have the login links assigned to the login screen with the `href` values as Facebook, Twitter, and LinkedIn. The URL of the login screen is `http://www.yoursite.com/user/login/` and once you click on the link for LinkedIn, you will get a URL like `http://www.yoursite.com/user/login/linkedin`. Earlier, we configured the path as `/user/login`. So, Opauth removes the path component from the URL and assigns the remaining component as the strategy. Here, the Opauth strategy will be assigned to LinkedIn. The rest of the process will be carried inside the Opauth library to redirect the user to LinkedIn's site.

Initializing the Opauth library

Let's start the process by including a template redirect action to intercept the request for initializing Opauth requests and handling the response. The following code displays the modified constructor of the Opauth plugin:

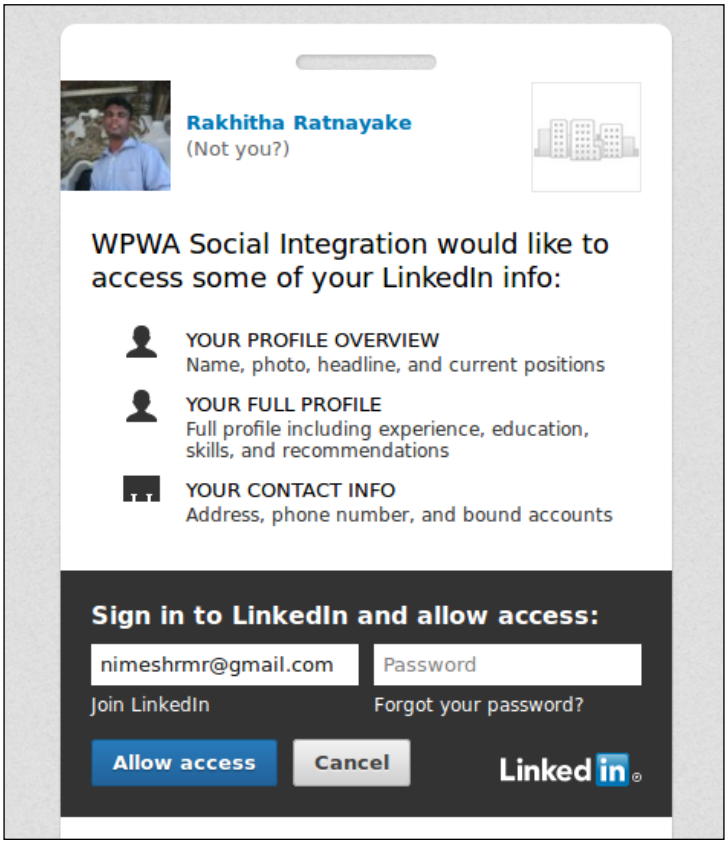
```
public function __construct() {
  add_action('template_redirect', array($this, 'load_opauth'));
  add_action('wpwa_login_addons', array($this, 'login_addons'));
}
```

Next, we need to load the Opauth configuration files, and initialize the Opauth class to work with the request redirection process. So let's consider the initial code of the `load_opauth` function for including the Opauth library:

```
public function load_opauth() {
  global $wp;
  $url_prefix = 'user/login/';
  $allowed_requests = array(  $url_prefix.'twitter',
  $url_prefix.'twitter/oauth_callback',
  $url_prefix.'linkedin',
  $url_prefix.'linkedin/oauth2callback',
  $url_prefix.'facebook',
  $url_prefix.'facebook/int_callback' );
    if (in_array($wp->request,
      $allowed_requests) || $wp->request == "opauth_success") {
      define('CONF_FILE', dirname(__FILE__) . '/' .
        'opauth.conf.php');
      define('OPAUTH_LIB_DIR', dirname(__FILE__) .
        '/lib/Opauth/');
  /**
     *Load config
  */
      if (!file_exists(CONF_FILE)) {
        trigger_error('Config file missing at ' . CONF_FILE,
        E_USER_ERROR);
        exit();
      }
      require CONF_FILE;

       /**
        * Instantiate Opauth with the loaded config
       */
       require OPAUTH_LIB_DIR . 'Opauth.php';
       $Opauth = new Opauth($config);
    }
}
```

First, we check for the value of `$wp->request` to prevent the loading of Opauth files in every request. To do this we filter the `opauth_success` and LinkedIn paths for including the library files. As the number of strategies increases, you will have to implement an advanced filter to include all the strategies.

Inside the preceding code, we have included the config and Opauth library files. The final part of the code contains the initialization of the `Opauth` class with the defined configurations. Once a user clicks on the **LinkedIn** link from the login screen, Opauth will handle the redirection using the application keys and configurations. When the user is redirected to LinkedIn, the following screen will appear asking the user to authenticate the application by logging in:



LinkedIn will redirect the user to our application once a user grants the permissions for the application. The callback URL configured in the config file will be used as the redirection path. The next step in this process is to handle the response and authenticate the user in our application.

Authenticating users in our application

Once a user is successfully authenticated inside LinkedIn, the request will be redirected to our application with the profile details of the user. Opauth's library provides the ability to work with the response parameters. We can configure third-party applications such as Facebook, Twitter, and LinkedIn to send various types of user details and activities. But each of these services will provide different types of data and hence it's difficult to match them into a common format. As developers, we should be relying on the most basic and common data across all services for OpenAuth login and registrations. Let's understand the process of authentication, before moving into the response handling part:

- The user clicks on the **LinkedIn** link
- The user is redirected to LinkedIn for granting permissions for our application
- The user is redirected to our application on successful authentication with profile details
- The application checks whether the user already exists, using the username
- Existing users will be automatically logged into the WordPress application
- Non-existing users will be saved in the application as a new user, using the details retrieved from LinkedIn and redirected to the profile for completing the remaining details

Now let's see how we can handle the Opauth response object to authenticate users into our application. Consider the following code for handling the profile details of the user:

```
if ($wp->request == 'opauth_success') {
  $response = null;
  switch ($Opauth->env['callback_transport']) {
    case 'session':
      session_start();
      $response = $_SESSION['opauth'];
      unset($_SESSION['opauth']);
      break;
    case 'post':
      $response = unserialize(base64_decode($_POST['opauth']));
      break;
    case 'get':
      $response = unserialize(base64_decode($_GET['opauth']));
      break;
```

```
      default:
        echo '<strong style="color: red;">Error:
          </strong>Unsupported callback_transport.' . "<br>\n";
        break;
    }
}
```

First, we have to check whether the request is successful by using the `$wp->request` parameter to match `opauth_success`. Opauth provides an environment variable named `callback_transport`, containing the type of response as `session`, `get`, or `post`. When the response of `callback_transport` is returned as `session`, we can directly access the response details from the PHP session variables. The response for the `get` and `post` type requests will be delivered as a Base64 encoded string and hence needs to be decoded prior to being used.

Now let's process the response for handling various strategies using the following code, which is located after the `switch` statement:

```
$provider = $response['auth']['provider'];
$username = "";
$first_name = "";
$email = "";
$pass = wp_generate_password();
$user_info = $response['auth']['info'];
switch ($provider) {
  case 'Facebook':
    $username = $user_info['email'];
    $first_name = $user_info['name'];
    $email = $user_info['email'];
    break;
  case 'Twitter':
    $username = $user_info['nickname'];
    $first_name = $user_info['name'];
    $email = '';
    break;
  case 'LinkedIn':
    username = $user_info['email'];
    $first_name = $user_info['name'];
    $email = $user_info['email'];
    break;
}
```

The response object contains a key format named `provider` to define the respective strategy of the request. We need to filter them using another `switch` statement. In this scenario, the value of the provider will be `linkedin`. Next, we define three variables for username, first name, and password. Here, we assign a default password, which needs to be updated once the user logs in.

Inside each strategy, we have to get the username, first name, and e-mail. Different services will have different keys for these parameters. You can get the respective keys for these parameters by looking at the {service} Strategy class located inside each service.

Most services don't provide the e-mail address with the default app settings. Hence, we have to use e-mail permission settings for each service, while creating the application to get access to the user's e-mail address.

Here we have assigned the e-mail as the username for Facebook and LinkedIn, while using the username for Twitter, as the e-mail is not provided by the Twitter API. Then we assign the e-mail from the response object. We have to use the following code after the switch statement for authenticating users:

```
if (username_exists($username)) {
  $user = get_userdatabylogin($username);
  wp_set_auth_cookie( $user->ID, false, is_ssl() );
  wp_redirect( admin_url( '' ) );
} else {
  $user_id = wp_insert_user(array('user_login' => $username,
    'user_pass' => $pass,
    'first_name' => $first_name ,
    "user_email"=>$email,"role"=>"developer"));
  if (is_wp_error($user_id)) {
    echo $user_id->get_error_message();
    exit;
  } else {
     wp_set_auth_cookie($user_id, false, is_ssl());
     wp_redirect(admin_url('profile.php'));
  }
}
```

We start the process by checking for the existence of the username in the database. Existence of a username means that the user has already registered using third-party authentication, so the user will be automatically authenticated and redirected to the dashboard using the `wp_redirect` function with an empty value.

WordPress uses the `wp_set_auth_cookie` function for authenticating users into the application. This method becomes handy in situations where we want to log the user in without knowing the password.

If the user doesn't exist, we create a new user account using the `wp_insert_user` function. Here, we have used developer as the default role, which was defined in *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*. For other user roles, we have to either create a separate set of login links or allow users to change the role after registration.

Once registered, the user will be authenticated by setting the WordPress authenticate cookie and will be redirected to the profile page to fill out their details. At this stage, the user will have a default password. Therefore, it's mandatory to update the user's profile with an e-mail and a new password to complete the registration.

The intention of this implementation was to learn how to integrate third-party open source plugins into WordPress web applications, so we chose Opauth for integrating third-party services for OpenAuth login. Now we have completed the plugin integration and OpenAuth login. Make sure to test the user's registration and login using different services. Having completed this process, you should be capable of integrating other plugins into WordPress applications.

Using third-party libraries and plugins

We discussed the importance of open source libraries in detail. Most WordPress developers prefer the creation of a web application by installing a bunch of third-party plugins. Throughout this book, we developed an application with a number of individual plugins. In this chapter, we've identified the pitfalls of creating a bunch of separate plugins. Ideally, developers should be focusing on limiting the number of plugins within an application to improve the code structure and eliminate possible conflicts.

On the other hand, some third-party libraries could contain malicious code that enables security holes in your applications. Even though there are some tools for checking malicious code, none of them are 100 percent accurate, and we can't guarantee the results. As developers, we should be always looking for stable and consistent libraries for our projects. So it's preferable to work with existing WordPress libraries and stable third-party plugins as much as possible when developing web applications.

Time for action

As usual, we discussed plenty of practical usages of open source libraries within WordPress. We completed the implementation of a few scenarios and left out some tasks for future development. As developers, you should take this opportunity to get the experience in integrating various third-party libraries:

- Implement the edit and delete functionality for the projects list using Backbone.js
- Integrate OpenAuth login for Facebook and Twitter
- Implement the subscriber notification process with a cron job and custom RSS feed

Summary

The open source nature of WordPress has improved developer engagement to customize and improve existing features by developing plugins, and contributing to the core framework. Inside the core framework, we can find dozens of popular open source libraries and plugins. We planned this chapter to understand the usage of trending open source libraries within the core.

First, we looked at the open source libraries inside the core. Backbone.js and Underscore.js are trending as popular libraries for web development and hence have been included in the latest WordPress version. Throughout this chapter, we looked at the use of Backbone.js inside WordPress, while building the developer profile page of the portfolio application. We looked into Backbone.js concepts such as models, collections, validation, views, and events.

Later on, we looked at the usage of existing PHP libraries within WordPress by using PHPMailer to build a custom e-mail sending interface. In web applications, developers don't always get the opportunity to build everything from scratch. So it's important to make use of existing libraries as much as possible.

As developers, you should have the know-how to integrate the third-party libraries as well as the existing ones. Hence we chose the third-party plugin called Opauth for integrating social network logins into our application. We completed the chapter by integrating the LinkedIn login page into the portfolio application.

In the next chapter, we are going to look at WordPress XML-RPC functions to build a simple yet flexible API for the portfolio application. Until then, make sure to try out the actions given in this chapter.

9
Listening to Third-party Applications

The complexity and size of web applications prompts developers to think about a rapid development process through third-party applications. Basically, we use third-party frameworks and libraries to automate the common tasks of web applications. Alternatively, we can use third-party services to provide functionalities that are not directly related to the application's core logic. Using APIs is a popular way of working with third-party services. The creation of an API opens the gates for third-party applications to access our application's data.

WordPress provides the ability to create an API through its built-in API powered by XML-RPC. The existing API caters to the blogging and CMS functionalities, while allowing developers to extend the APIs with custom functionality. This chapter covers the basics of an existing API, while building the foundation of a portfolio management system API. Here, you will learn the necessary techniques for building complex APIs for larger applications.

In this chapter we will cover the following topics:

- Introduction to APIs
- WordPress XML-RPC API for web applications
- Building the API client
- Creating custom API
- Integrating API user authentication
- Integrating API access tokens
- Providing API documentation

Let's get started.

Introduction to APIs

API is the acronym for **Application Programming Interface**. According to the definition on Wikipedia, an API specifies a set of functions that accomplishes specific tasks or allow working with specific software components. As web applications grow larger, we might need to provide the application services or data to third-party applications. We cannot let third-party applications access our source code or database directly due to security reasons. So APIs allow the access of data and services of the application through a restricted interface, where users can only access the data provided through API. Typically, users are requested to authenticate themselves by providing usernames and necessary passwords or API keys. So, let's look at the advantages of having an application specific API.

Advantages of having an API

Often, we see the involvement of APIs with popular web and mobile applications. As the owner of the application, you have many direct and indirect advantages by providing an API for third-party applications. Let's go through some of the distinct advantages of having an API:

- Access to an API can be provided to third-party applications as a free or premium service
- User traffic increases, as more and more applications use the API
- By offering an API, you get free marketing and popularity among people who aren't familiar with your application
- Generally, an API automates tasks that require user involvement, allowing for a much better and quicker experience for the users

With the increasing usage of mobile devices, API-based applications are growing faster than ever before. Most of the popular web applications and services have opened up their API for third-party applications, and the others are looking to build their API to compete in this rapidly changing world of web development. Here are some of the most popular existing APIs used by millions of users around the world:

- **Twitter REST API**: `http://goo.gl/5Nukrb`
- **Facebook Graph API**: `http://goo.gl/RwgKsT`
- **Google Maps API**: `http://goo.gl/aoaLo9`
- **Amazon Product Advertising API**: `http://goo.gl/6iLxfw`
- **Youtube API**: `http://goo.gl/MMFAFa`

Considering the future of web development, it's imperative to have knowledge in building an API to extend the functionalities of web applications. So, we are going to look at the WordPress XML-RPC API in the next section.

WordPress XML-RPC API for web applications

With the latest versions, WordPress API has matured into a secure and flexible solution that easily extends to cater to complex features. It was considered to be an insecure feature that exposes the security vulnerabilities of WordPress, hence, was disabled by default in earlier versions. As of Version 3.5, XML-RPC is enabled by default and the enable/disable option from the admin dashboard has been completely removed. As developers, now we don't have to worry about the security risks identified in earlier releases.

The existing APIs mainly focus on addressing functionalities for blogging and CMS related tasks. In web applications, we can make use of these API functions to build an API for third-party applications and users. The following list contains the existing components in the WordPress API:

- Posts
- Taxonomies
- Media
- Comments
- Options
- Users

The complete list of components and their respective API functions can be found in the WordPress codex at `http://codex.wordpress.org/XML-RPC_WordPress_API`. Let's see how we can use the existing API functions of WordPress.

Building the API client

WordPress provides the support for its API through the `xmlrpc.php` file located inside the root of the installation directory. Basically, we need two components to build and use an API:

- **API server**: This is the application where the API function resides
- **API client**: This is a third-party application or service that requests the functionality of an API

Since we are going to use the existing API functions, we don't need to worry about the server as it's built inside the core. So, we are going to build a third-party client to access the service. Later, we will be improving the API server to implement custom functionalities that go beyond the existing API functions. The API client is responsible for providing the following features:

- Authenticate the user with the API
- Make XML-RPC requests to the server through the `curl` command
- Define and populate API functions with necessary parameters

Having the preceding features in mind, let's look at the implementation of an API client:

```
class WPWA_XMLRPC_Client {
  private $xml_rpc_url;
  private $username;
  private $password;
  public function __construct($xml_rpc_url, $username, $password){
    $this->xml_rpc_url = $xml_rpc_url;
    $this->username = $username;
    $this->password = $password;
  }
  public function api_request($request_method, $params) {
    $request = xmlrpc_encode_request($request_method, $params);
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_POSTFIELDS, $request);
    curl_setopt($ch, CURLOPT_URL, $this->xml_rpc_url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
    curl_setopt($ch, CURLOPT_TIMEOUT, 1);
    $results = curl_exec($ch);
    $response_code = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    $errorno = curl_errno($ch);
    $error = curl_error($ch);
    curl_close($ch);
    if ($errorno != 0) {
      return array("error" => $error);
    }
    if ($response_code != 200) {
      return array("error" => "Request Failed : $results");
    }
    return xmlrpc_decode($results);
  }
}
```

First, we define three instance variables for API URL, username, and password. The constructor is used to initialize the instance variables through the parameters provided by users in object initialization. Next, we have the `api_request` function for making the `curl` requests to the API. Here, we take two parameters as request method and attributes. WordPress API provides the request method for each API function. A user can pass the necessary parameter values for the API call through the `$params` array.

Inside the `api_request` function, we use the `xmlrpc_encode_request` function provided by PHP to generate an XML file from the passed parameters and request data. Then, we can pass the converted XML file to a curl request to invoke the API functions on server. Based on the server response, we can generate an error or retrieve the decoded result by using the `xmlrpc_decode` function. Generally, the result returned from the server will be in string, integer, or array format.

Official PHP documentation states that the `xmlrpc_encode_request` and `xmlrpc_decode` functions are experimental and should be used at our own risk. We have used those functions here, considering the scope of this chapter. We can create the XML request manually, or use the third-party XML-RPC library to provide a much more stable solution. In case, you decide to implement manual XML request creation, use the following format to generate the parameters:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>subscribeToDevelopers</methodName>
      <params>
         <param>
            <value><string>username</string></value>
         </param>
      </params>
</methodCall>
```

Now, we have the basic API client for requesting or sending server data. Next, we need to define functions inside the client for invoking various API functions. Here, we are going to implement API functions for accessing the services and projects of the portfolio management system. So, let's update the API client class with the following two functions:

```
function getLatestProjects() {
    $params = array(0, $this->username, $this->password, array("post_
type" => "wpwa_project"));
    return $this->api_request("wp.getPosts", $params);
```

```
}
function getLatestServices() {
    $params = array(0, $this->username, $this->password, array("post_
type" => "wpwa_services"));
    return $this->api_request("wp.getPosts", $params);
}
```

We have implemented two functions with similar code for accessing WordPress posts. WordPress provides the ability to access any post type through the `wp.getPosts` request method. The request is invoked by passing the request method and parameters to the `api_request` function created created earlier in this section.

The parameters array contains four values for this request. The first parameter of `0` defines the blog ID. Next, we have the username and password of the user who wants to access the API. Finally, we have an array with optional parameters for filtering results. It's important that you pass the values in the preceding order as WordPress looks for the parameter values by its index.

Here, we have used `post_type` as the optional parameter for filtering services and projects from the database. The following is a list of allowed optional parameters for the `wp.getPosts` request method:

- `post_type`
- `post_status`
- `number`
- `offset`
- `orderby`
- `order`

In the codex, it mentions that the response of `wp.getPosts` will only contain posts that the user has permission to edit. Therefore, a user will only receive the permitted post list. If you want to allow public access to all the post details, a custom API function needs be developed to query the database.

Now, let's take a look at how to invoke the API function by initializing the API client, as illustrated in the following code:

```
$wpwa_api_client = new WPWA_XMLRPC_Client("http://www.yoursite.com/
xmlrpc.php",
  "username", "password");
$projects = $wpwa_api_client->getLatestProjects();
$services = $wpwa_api_client->getLatestServices();
```

We can invoke the API by initializing the `WPWA_XMLRPC_Client` class with the three parameters, as we discussed at the beginning of this process. So, the final output of the `$projects` and `$services` variables will contain an array of posts.

Keep in mind that an API client is a third-party application or service. So, we have to use this code outside the WordPress installation to get the desired results.

So far, we looked at the usage of existing API functions within WordPress. Next, we are going to look at the possibilities of extending the API to create custom methods.

Creating a custom API

Custom APIs are essential for adding web application specific behaviors, which goes beyond the generic blogging functionality. We need the implementation for both the server and client to create a custom API. Here, we are going to build an API function that outputs the list of developers in a portfolio application. As usual, we are going to get started by creating another plugin folder named `wpwa-xml-rpc-api` with the main file named `class-wpwa-xml-rpc-api.php`.

Let's look at the initial code to build the API server:

```
class WPWA_XML_RPC_API {
  public function __construct() {
    add_filter('xmlrpc_methods', array($this, 'xml_rpc_api'));
  }
  public function xml_rpc_api($methods) {
    $methods['wpwa.getDevelopers'] = array($this,
'developers_list');
    return $methods;
  }
}
new WPWA_XML_RPC_API();
```

First, we use the plugin constructor to add the WordPress filter named `xmlrpc_methods`, which allows us to customize the API functions assigned to WordPress. The preceding filter will call the `wpwa_xml_rpc_api` function by passing the existing API methods as the parameter. The `$methods` array contains both existing API methods as well as methods added by plugins.

Inside the function, we need to add new methods to the API. WordPress uses `wp` as the namespace for the existing methods. Here, we have defined the custom namespace for application specific functions as `wpwa`. The preceding code adds a method named `getDevelopers` in the `wpwa` namespace to call a function named `developers_list`. The following code contains the implementation of the `developers_list` function for generating all the developers list as the output:

```
public function developers_list($args) {
    $user_query = new WP_User_Query(array('role' => 'developer'));
    return $user_query->results;
}
```

The list of developers are generated through the `WP_User_Query` object by using the developer role as the filter. Now, we have the API server ready with the custom function. Consider the following code to understand how custom API methods are invoked by the client:

```
function getDevelopers(){
    $params = array();
    return $this->api_request("wpwa.getDevelopers", $params);
}

$wpwa_api_client = new
  WPWA_XMLRPC_Client("http://www.yoursite.com/xmlrpc.php",
  "username", "password");

$developers = $wpwa_api_client->getDevelopers();
```

As we did earlier, the definition of the `getDevelopers` function is located inside the client class and the API is initialized from outside the class. Here, you will receive a list of all the developers in the system.

A similar process can be used to get a list of projects and services, instead of using `wp.getPosts`, which limits the posts based on permission.

Now we know the basics of creating a custom API with WordPress. In the next section, we are going to look at the authentication for custom API methods.

Integrating API user authentication

Building a stable API is not one of the simplest tasks in web development. But, once you have one, hundreds of third-party applications will be requesting to connect to the API, including potential hackers. So it's important to protect your API from malicious requests and avoid an unnecessary overload of traffic. Therefore, we can request an API authentication before providing access to the user.

The existing API functions come built-in with user authentication, hence, we had to use user credentials in the section where we retrieved a list of projects and services. Here, we need to manually implement authentication for custom API methods. Let's create another API method for subscribing to developers of portfolio application. This feature is already implemented in the admin dashboard using admin list tables. Now, we are going to provide the same functionality for API users. Let's get started by modifying the `xml_rpc_api` function as follows:

```
public function xml_rpc_api($methods) {
   $methods['wpwa.subscribeToDevelopers'] = array($this, 'developer_
subscriptions');
   $methods['wpwa.getDevelopers'] = array($this, 'developers_list');
   return $methods;
}
```

Now, we can build the subscription functionality inside the `developer_subscriptions` function using the following code:

```
public function developer_subscriptions( $args ) {
  global $wpdb;
  $username = isset( $args['username'] ) ? $args['username'] : '';
  $password = isset( $args['password'] ) ? $args['password'] : '';
  $user = wp_authenticate( $username, $password );
  if (!$user || is_wp_error($user)) {
    return $user;
  }
  $follower_id = $user->ID;
  $developer_id = isset( $args['developer'] ) ?
     $args['developer'] : 0 ;
  $user_query = new WP_User_Query( array( 'role' => 'developer',
    'include' => array( $developer_id ) ) );
  if ( !empty($user_query->results) ) {
    foreach ( $user_query->results as $user ) {
      $wpdb->insert(
        $wpdb->prefix . "subscribed_developers",
         array(
            'developer_id' => $developer_id,
```

```
                'follower_id' => $follower_id
            ),
            array(
                '%d',
                '%d'
            )
        );
      return array("success" => "Subsciption Completed.");
      }
    } else {
      return array("error" => "Invalid Developer ID.");
    }
    return $args;
}
```

First, we retrieve the username and password from the arguments array and call the built-in the `wp_authenticate` function by passing them as parameters. This function will authenticate the user credentials against the `wp_users` table. If the credentials fail to match a user from the database, we return the error as an object of the `WP_Error` class.

Notice the use of array keys for retrieving various arguments in this function. By default, WordPress uses array indexes as 0, 1, 2, and so on, for retrieving the arguments as the ordering of arguments is important. Here we have introduced key-based parameters, so that users have the freedom of sending parameters without worrying about the order.

Once the user is successfully authenticated, we can access the ID of the user to be used as the follower. We also need the ID of a preferred developer through method parameters. Next, we get the details of a preferred developer by using the `WP_User_Query` class by passing the role and developer ID. Finally, we insert the record into the `wp_subscribed_developers` table to create a new subscription for a developer.

Other parts of the code contain the necessary error handlings based on various conditions. Make sure to keep a consistent format for providing error messages.

Now, we can implement the API client code by adding the following code into the API client class:

```
function subscribeToDevelopers($developer_id){
  $params = array("username"=>$this->username,
    "password"=>$this->password,"developer"=>$developer_id);
  return $this->api_request("wpwa.subscribeToDevelopers",
    $params);
}
```

Here, we call the custom API method named `wpwa.subscribeToDevelopers` with the necessary parameters. As usual, we invoke the API by initializing an object of the `WPWA_XMLRPC_Client` class, as shown in the following code:

```
$wpwa_api_client = new
  WPWA_XMLRPC_Client("http://yoursite.com/xmlrpc.php", "follower",
  "follower123");
$subscribe_status = $wpwa_api_client->subscribeToDevelopers(1);
```

Once implemented, this API function allows followers to subscribe to the activities of developers.

Integrating API access tokens

In the preceding section, we introduced API authentication to prevent unnecessary access to the API. Even the authenticated users can overload the API by unnecessarily accessing it, therefore, we need to implement user tokens to limit the use of the API. There can be many reasons for limiting requests to an API. We can think of two main reasons for limiting the API access as listed here:

- Avoid unnecessary overloading of server resources
- Bill the users based on API usage

If you are developing a premium API, it's important to track the usage for billing purposes. Various APIs uses unique parameters to measure API usage. Here, are some of the unique ways of measuring API usage:

- Twitter API uses number of requests per hour to measure the API usage
- Google Translate uses number of words translated to measure the API usage
- Google Cloud SQL uses input and output storage to measure the API usage

Here, we are not going to measure the usage or limit the access to the portfolio API. Instead, we will be creating user tokens for measuring the usage in the future. We are going to allow API access to follower user role. First, we have to create an admin menu page for generating user tokens. Let's update the `WPWA XML-RPC API` plugin constructor by adding the following action:

```
public function __construct() {
  add_filter('xmlrpc_methods', array($this, 'xml_rpc_api'));
  add_action('admin_menu', array($this, 'api_settings'));
}
```

The following code contains the implementation of the `api_settings` function to create an admin menu page for token generation:

```
public function api_settings() {
  add_menu_page('API Settings', 'API Settings',
    'follow_developer_activities', 'wpwa-api', array($this,
    'user_api_settings'));
}
```

We are not going to discuss the preceding code in detail, as we have already done it in previous chapters. Only followers are allowed to access API through tokens, and thus, a capability named `follow_developer_activities` is used to integrate the screen for the followers only. Now, we can look at the `user_api_settings` function for the implementation of the token generation screen as follows:

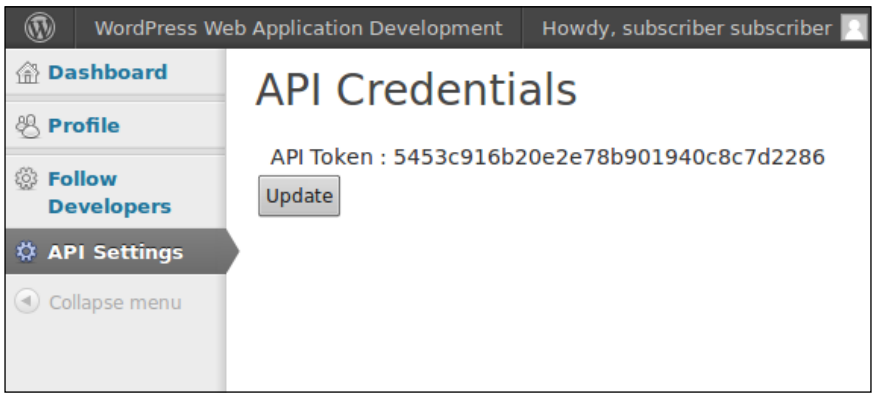
```
public function user_api_settings() {
  $user_id = get_current_user_id();
  if ( isset( $_POST['api_settings'] ) ) {
    $api_token = $this->generate_random_hash();
    update_user_meta( $user_id, "api_token", $api_token );
  } else {
    $api_token = (string) get_user_meta($user_id, "api_token",
      TRUE);
      if ( empty($api_token) ) {
        $api_token = $this->generate_random_hash();
        update_user_meta( $user_id, "api_token", $api_token );
      }
  }
  $html = '<div class="wrap"><form action="" method="post"
    name="options">
  <h2>API Credentials</h2>
  <table class="form-table" width="100%" cellpadding="10">
  <tbody>
    <tr>
      <td scope="row" align="left">
      <label>API Token : ' . $api_token . '</label>
      </td>
    </tr>
  </tbody>
  </table>
  <input type="submit" name="api_settings" value="Update"
/></form></div>';

  echo $html;
}
```

In the preceding code, the same function is used for generating the screen as well as handling the form submission. We have an HTML form that displays the current API token for the user. Once the form is submitted, a new token needs to be generated as a hashed string.

First, we check for the submission of the form. Then, we generate a new token using a custom function named `generate_random_hash`. The following code shows the implementation of the `generate_random_hash` function inside the `WPWA_XML_RPC_API` class:

```
public function generate_random_hash($length = 10) {
  $characters =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
  $random_string = '';
  for ($i = 0; $i < $length; $i++) {
    $random_string .= $characters[rand(0,
      strlen($characters) - 1)];
  }

  $random_string = wp_hash($random_string);
  return $random_string;
}
```

We can generate a dynamic hashed string by passing a random string to the existing `wp_hash` function provided by WordPress. Then, we update the generated token using the `update_user_meta` function. If a user is loading the screen without submission, we display the existing token to the user. Now, you will have a menu item called **API Settings** in the admin menu bar for generating the API token screen, as shown in the following screenshot:



Having created the token, we now have to check the token values before providing the access to the API. So consider the updated version of the `developer_subscriptions` function as shown in the following code:

```
public function developer_subscriptions( $args ) {
    global $wpdb;
    $username = isset($args['username']) ? $args['username'] : '';
    $password = isset($args['password']) ? $args['password'] : '';
    $user = wp_authenticate( $username, $password );
    if (!$user || is_wp_error($user)) {
      return $user;
    }
    $follower_id = $user->ID;
    $api_token = (string) get_user_meta($follower_id, "api_token",
      TRUE);
    $token = isset( $args['token'] ) ? $args['token'] : '';
    if ( $args['token'] == $api_token) {
      $developer_id = isset( $args['developer'] )
          ? $args['developer'] : 0 ;
      $user_query = new WP_User_Query( array(
         'role' => 'developer', 'include' => array( $developer_id
        ) ) );
      if ( !empty($user_query->results) ) {
        foreach ( $user_query->results as $user ) {
          $wpdb->insert(
            $wpdb->prefix . "subscribed_developers",
            array(
              'developer_id' => $developer_id,
              'follower_id' => $follower_id
             ),
            array(
               '%d',
               '%d'
            )
          );
        return array("success" => "Subsciption Completed.");
        }
      }
      else {
          return array("error" => "Invalid Developer ID.");
      }
    }
    else {
      return array("error" => "Invalid Token.");
    }
    return $args;
}
```

Now the user will have to log into the WordPress admin, and generate a token before using the API. In a premium API, you either can bill the user for purchasing the API token or bill the user based on their usage.

So now, the API client code also needs to be changed to include the token parameter. The following code contains the updated call to the API with the inclusion of tokens:

```
$wpwa_api_client = new WPWA_XMLRPC_Client("http://www.yoursite.com/
xmlrpc.php",
  "username", "password");
$subscribe_status = $wpwa_api_client->
  subscribeToDevelopers("developer id", "api token");
```

Providing the API documentation

Typically, most popular APIs provide complete documentation for accessing the API methods. Alternatively, we can use a new API method to provide details about all other API methods and parameters. This allows the third-party users to request an API method and get the details about all other functions.

WordPress uses the API method named `system.listMethods` for listing all the existing methods inside the API. Here, we are going to go one step further by providing the API method parameters with the complete list.

We can start the process by adding another API method to the `xml_rpc_api` function, as shown in the following code:

```
public function xml_rpc_api($methods) {
    $methods['wpwa.subscribeToDevelopers'] = array($this,
      'developer_subscriptions');
    $methods['wpwa.getDevelopers'] = array($this,
      'developers_list');
    $methods['wpwa.apiDoc'] = array($this, 'wpwa_api_doc');
    return $methods;
}
```

Once updated, we can use the following code to provide details about the API methods:

```
public function api_doc() {
    $api_doc = array();
    $api_doc["wpwa.subscribeToDevelopers"] =
      array("authentication" => "required",
            "api_token" => "required",
```

```
                "parameters" => array("Developer ID", "API Token"),
                "result" => "Subscribing to Developer Activities"
          );

        $api_doc["wpwa.getDevelopers"] =
        array("authentication" => "optional",
              "api_token" => "optional",
              "parameters" => array(),
              "result" => "Retrieve List of Developers"
        );
        return $api_doc;
    }
```

Here, we have added all of the custom API functions with all the necessary details for making use of them. The `authentication` parameter defines whether a user needs to provide login credentials for accessing the API. The `api_token` parameter defines whether a user needs a token to proceed. The `parameters` parameter defines a list of parameters allowed to the API method and finally, the `result` parameter defines what a user will receive after accessing the API method.

Now we have completed the process of working with APIs in WordPress. You should be able to build complex APIs for web applications by using the discussed techniques.

Time for action

Throughout this chapter, we looked at the various aspects of the WordPress XML-RPC API while developing practical scenarios. In order to build stable and complex custom APIs, you should have the practical experience of preceding techniques. So, I recommend that you should try the following tasks, to extend the knowledge that you gathered in this chapter:

- Measure the API usage through number of requests.
- We created an API function to list all of the projects and services of the application. Try to introduce filtering of results with additional parameters.
- The custom API created in this chapter returns an array as the result. Introduce different result formats such as JSON, XML, and array so that developers can choose their preferred format.

Summary

We started this chapter with the intention of building an XML-RPC based API for web applications. Then, we discussed the usage of existing API functions while building an API client from scratch.

Complex applications will always exceed the limits of an existing API, hence, we looked at the possibility of creating a custom API. User authentication and API tokens were necessary for preventing unnecessary API access and measured the API usage. Finally, we looked at the possibility of creating the API documentation through another API function. Having completed the API creation techniques, you should now be able to develop complex APIs to suit any kind of web application.

In the final chapter, we are going to integrate all of the individual plugins created throughout this book to illustrate the techniques for structuring large web applications, while looking at some of the uncompleted areas of portfolio management application. So be prepared for an exciting finish to this book.

10
Integrating and Finalizing the Portfolio Management Application

Building a large web application is a complex task which should be planned and managed with well-defined processes. Typically, we separate large applications into smaller submodules, where each submodule is tested independently. Finally, we integrate all the modules to complete the application. Integration of modules is one of the most difficult tasks in application development.

Many existing developers who are familiar with building simple websites tend to use a bunch of third-party plugins, and hence they seem to lack the knowledge of building proper applications with a large number of submodules. Here, we will be addressing this issue by restructuring the plugin-based application into an integrated standalone application. After the completion of this chapter, you should be able to build similar or more complex applications without any difficulty.

In this chapter, we will be covering the following topics:

- Integrating and structuring a portfolio application
- Restructuring the custom post manager
- Integrating a template loader into the user manager
- Working with a restructured application
- Updating user profiles with additional fields
- Scheduling subscriber notifications

- Lesser known WordPress features
- Time for action
- Final thoughts

Let's get started.

Integrating and structuring a portfolio application

Throughout the first nine chapters, we implemented the functionality of the developer portfolio management system using several independent plugins. While developing, we identified the pitfalls of creating a large number of plugins for an application. Here are some of the issues we faced with multiple plugin architecture:

- Reusable functions and libraries had to be duplicated inside many plugins
- Separation of concerns was scattered in multiple places
- Difficulty in understanding the execution flow

So, now we need to restructure the application to resolve the preceding issues and build a solid foundation to develop any type of web application with WordPress. We will be going through a step-by-step process to refactor the complete application from scratch. Let's get started.

Step 1 – deactivating all the plugins used in this book

In this process, we are going to integrate all the plugins into a standalone plugin for our portfolio application, and hence it's necessary to remove the existing plugins by deactivating them through the admin dashboard.

We can also automatically deactivate all the plugins at once by moving them into a different folder.

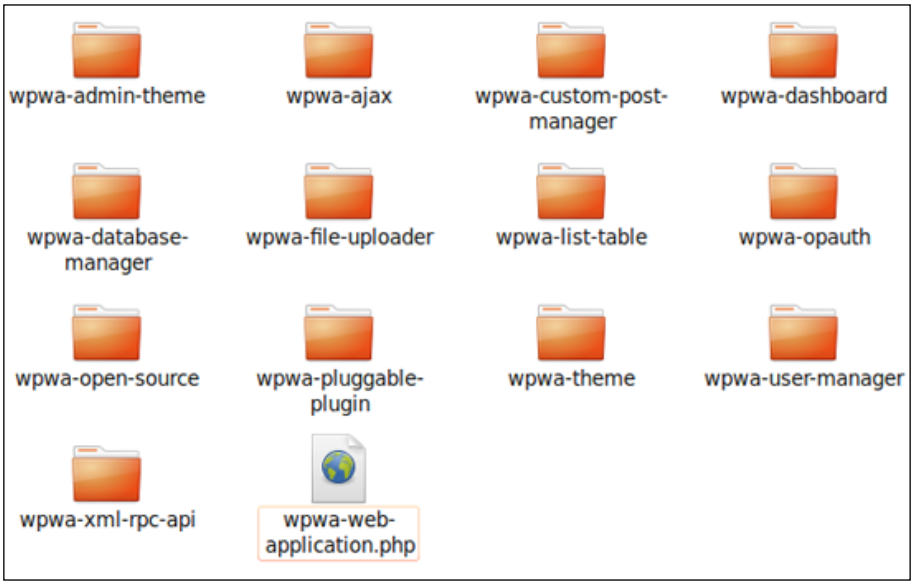
Step 2 – creating a new standalone plugin

Create a folder named `wpwa-web-application` with the main file as `wpwa-web-application.php`. We are going to integrate all the other plugin functionalities into this plugin. Consider the following code for the plugin definition:

```
<?php
/*
  Plugin Name: WPWA Web Application
  Plugin URI: http://www.innovativephp.com/
  Description: Building a Portfolio Management System to
  illustrate the power of WordPress as a web application
  development framework
  Author: Rakhitha Nimesh
  Version: 1.0
  Author URI: http://www.innovativephp.com/
*/
```

Step 3 – moving all the plugins into wpwa-web-application

All the existing plugins will be contained within the main plugin folder to provide the existing functionality. Here, all the existing plugins are going to act as components instead of actual WordPress plugins. The following screenshot illustrates the folder structure of `wpwa-web-application`, with all the other plugins as components:



Step 4 – removing plugin definitions

As mentioned earlier, all the existing plugins need to work as components instead of plugins. So, we need to remove the existing plugin definitions from the main files.

Now we are ready to start the restructuring process of the application. Before we move on to the next step, it's important to understand the things to be restructured from the original code. So, let's identify the requirements for solving the issues discussed in the beginning of this section:

- Controlling should be handled from a single location instead of spreading them across each component
- Models, views, and controllers should be merged into a single location
- Routing and the necessary routing rules need to be separated
- Loading of scripts and styles should be provided in a single location
- The single-plugin-activation handler should be able to provide all the activation-related tasks
- WordPress actions should be minimized to increase reusability

With these goals in mind, let's start the next step of this process.

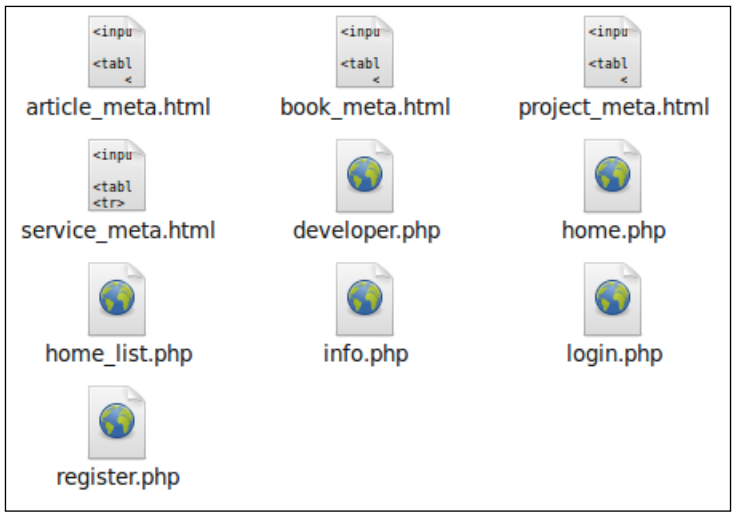
Step 5 – creating common folders

We have created plugin-specific folders to contain scripts, styles, templates, and so on. This technique reduced the reusability of code, and hence we had to work with the duplicate code in many occasions. The creation of a duplicate template loader in *Chapter 8*, *Enhancing the Power of Open Source Libraries and Plugins*, is a perfect example of the issues with this technique. So, we have to create common folders in the root plugin folder for CSS, JS, templates, models, and controllers.

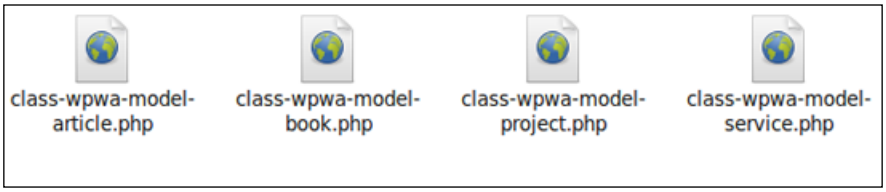
Then, we have to copy all the CSS, JavaScript, templates, and model files into the respective folders. The following screenshot illustrates the contents of the `css` folder after all the files are copied:

The following screenshot previews the contents of the `js` folder after the files are integrated into a single plugin:

The following screenshot shows the contents of the `templates` folder after the files are integrated into a single plugin**:**



The following screenshot previews the contents of the `models` folder after the files are integrated into a single plugin:



Step 6 – loading components to the main plugin

Now all the existing plugins will work as components, and hence they need to be loaded into the main plugin before usage. We can either include them directly to the main file or use an autoloader function to load them on demand. Here, we will be implementing the direct file inclusion using the `require_once` statement. Consider the following code for including all the component files into the `wpwa-web-application.php` file:

```
require_once 'wpwa-list-table/class-wpwa-list-table.php';
require_once 'wpwa-admin-theme/class-wpwa-admin-theme.php';
require_once 'wpwa-ajax/class-wpwa-ajax.php';
require_once 'wpwa-custom-post-manager/class-wpwa-custom-post-
manager.php';
require_once 'wpwa-dashboard/class-wpwa-dashboard.php';
require_once 'wpwa-database-manager/class-wpwa-database-
manager.php';
require_once 'wpwa-file-uploader/class-wpwa-file-uploader.php';
require_once 'wpwa-opauth/class-wpwa-opauth.php';
require_once 'wpwa-open-source/class-wpwa-open-source.php';
require_once 'wpwa-pluggable-plugin/wpwa-pluggable-plugin.php';
require_once 'wpwa-theme/class-wpwa-theme.php';
require_once 'wpwa-user-manager/class-wpwa-user-manager.php';
require_once 'wpwa-xml-rpc-api/class-wpwa-xml-rpc-api.php';
```

Only the main file of each component is loaded in the preceding code. Subfiles within those components will be loaded by the main files, as we did in the original code.

Step 7 – creating the template loader

We used duplicate versions of the custom template loader inside the `wpwa-open-source` and `wpwa-theme` plugins. So, we need to re-use the template loader across many components. Therefore, the template loader is included in the main file and removed from the two components mentioned previously:

```
include_once 'class-wpwa-template-loader.php';
```

Step 8 – reusing the autoloader

We created an autoloader function to load the classes on demand for the custom post type manager component. The autoloader was placed inside this component and hence was not reusable inside other components. Now we have to move the autoloader into the main plugin file to enable future enhancements with different autoloading rules.

Step 9 – defining main plugin functions

Controlling the user request and assigning the request into subcomponents is the main functionality of the `WPWA Web Application` plugin. Consider the following code for initial function definitions:

```
class WPWA_Web_Application{
  public function initialize_controllers() { }
  public function initialize_app_controllers() { }
}
```

Here, we have defined two functions for controlling the logic of our application. First, we handle the common controlling tasks through the `initialize_controllers` function. Application-specific controlling is handled by the `initialize_app_controllers` function.

WordPress uses action hooks to initialize most of the functionalities, and hence we can consider action hooks as controllers. We have defined actions inside the constructor of each component. It's ideal to keep the component constructors free by separating the controlling code.

We have identified four components that can be treated as controllers within our application:

- **Template_Router**: This manages request routing and routing rule definitions
- **Activation_Controller**: This manages plugin-activation-related tasks
- **Script_Controller**: This manages the loading of scripts and style files
- **Admin_Menu_Controller**: This manages all the admin-menu-related actions in the dashboard

As applications grow larger, we might get more and more reusable components to be used as separate controllers.

Step 10 – building the template router

First, create a new file named `class-template-router.php` inside the `controllers` folder and define a class named `Template_Router`. This class is responsible for handling template redirections, routing rules, and WordPress query variables. Let's look at the implementation of `Template_Router`:

```
class Template_Router {
  public function redirect_templates() {
    add_action('template_redirect', array($this, 'main_router'));
    add_filter('query_vars', array($this,
    'manage_routes_query_vars'));
    add_action('init', array($this, 'manage_routing_rules'));
  }

  public function main_router() {
    $opauth = new WPWA_Opauth();
    $opauth->load_opauth();
    $this->front_controller();
    $app_theme = new WPWA_Theme();
    $app_theme->application_controller();
  }
  public function manage_routes_query_vars($query_vars) {
    $query_vars[] = 'control_action';
    $query_vars[] = 'record_id';
    return $query_vars;
  }
  public function manage_routing_rules() {
    add_rewrite_rule('user/([^/]+)/?',
    'index.php?control_action=$matches[1]', 'top');
    add_rewrite_rule('user/([^/]+)/([^/]+)/?',
    'index.php?control_action=$matches[1]&record_id=$matches[2]',
    'top');
  }
  public function flush_rewriting_rules(){
    $this->manage_routing_rules();
    flush_rewrite_rules();
  }
  public function front_controller() {
    global $wp_query;
    $control_action = $wp_query->query_vars['control_action'];
    switch ($control_action) {
      case 'register':
        do_action('wpwa_register_user');
        break;
```

```
        case 'login':
          do_action('wpwa_login_user');
          break;
        case 'activate':
          do_action('wpwa_activate_user');
          break;
        case 'profile':
          $developer_id = $wp_query->query_vars['record_id'];
          $app_theme = new WPWA_Open_source();
          $app_theme->create_developer_profile($developer_id);
          break;
      }
    }
  }
```

We have used the `redirect_templates` function to configure all the actions related to the routing functionality. This enables us to keep routing rules and query variables in a single location. Once the common routing code is defined, we use the `main_router` function to invoke the component-specific routing functions. Originally, routing rules were used in the plugins named `wpwa-user-manager`, `wpwa-theme`, `wpwa-open-source`, and `wpwa-opauth`. We used a different type of routing process inside these plugins. The routing procedure used in `wpwa-theme` and `wpwa-opauth` is unique, and hence we invoke the routing function within these components. On the other hand, `wpwa-user-manager` and `wpwa-open-source` contain a common routing process, and hence we can merge them into a common function named `front_controller` inside the `Template_Router` class.

You might notice the invocation of `front_controller` between the `WPWA_Opauth` and `WPWA_Theme` functions. Try to figure out the reason and solve the issue by restructuring the code.

Make sure you remove the action and function implementations from the components once defined in the controllers. Now, let's look at the code for invoking `Template_Router` from the main plugin as illustrated in the following code:

```
public function initialize_controllers(){
  require_once 'controllers/template_router.php';
  $template_router = new Template_Router();
  $template_router->redirect_templates();
}
```

Step 11 – building the activation controller

We used activation hooks inside several plugins to execute one-time tasks on plugin activation. Definition of the `register_activation_hook` function was duplicated across many plugins. Therefore, we create a separate controller for executing all the activation functions through a single call to the `register_activation_hook` function. Create a file named `class-activation-controller.php` with a class named `Activation_Controller`. The following code illustrates the final implementation of `Activation_Controller`:

```
class Activation_Controller{
  public function initialize_activation_hooks(){
    register_activation_hook("wpwa-web-application/wpwa-web-
    application.php",array($this,'execute_activation_hooks' ));
  }
  public function execute_activation_hooks(){
    $database_manager = new WPWA_Database_Manager();
    $database_manager->create_custom_tables();
    $user_manager = new WPWA_User_Manager();
    $user_manager->add_application_user_roles();
    $user_manager->remove_application_user_roles();
    $user_manager->add_application_user_capabilities();
    $template_router = new Template_Router();
    $template_router->flush_rewriting_rules();
  }
}
```

Here, we have handled all the activation-related code through a single `register_activation_hook` definition. The `execute_activation_hooks` function is responsible for calling activation-specific functions inside each component. In the original code, we had activation hooks inside the `wpwa-database-manager` and `wpwa-user-manager` plugins, and hence we have initialized two objects of these classes to call the respective functions on plugin activation.

In the original code, we used the `register_activation_hook` function inside the main plugin file. The first parameter of this function takes the path of the main plugin file, and hence we were able to use `__FILE__` as the path. Here, we are executing the `register_activation_hook` function through a subfile. Therefore, the path to the main plugin file will be `wpwa-web-application/wpwa-web-application.php`.

Inside the `Template_Router` class, we create new routing rules, which need to be updated through flushing. It's a resource-expensive task, and hence we execute on plugin activation. So, we call the `flush_rewriting_rules` function of the `Template_Router` class to update the new rewriting rules. Finally, we update the `initialize_controllers` function of the `WPWA_Web_Application` class to initialize the `Action_Controller` class as shown in the following code:

```
require_once 'controllers/class-activation-controller.php';
$activation_controller = new Activation_Controller();
$activation_controller->initialize_activation_hooks();
```

Step 12 – building the script controller

Generally, we use scripts and styles inside every plugin. Originally, we used multiple plugins and hence we couldn't prevent folder duplication. Now we need to merge all the scripts and styles to a common controller. So, let's create another controller named `class-script-controller.php` to contain a class named `Script_Controller`. With the new implementation, we should be able to load all the scripts and styles through a single enqueue statement. Let's consider the code for `Script_Controller`:

```
class Script_Controller{
  public function enque_scripts(){
    add_action('wp_enqueue_scripts', array($this,
    'include_scripts_styles'));
    add_action('admin_enqueue_scripts', array($this,
    'include_admin_scripts_styles'));
    add_action('login_enqueue_scripts', array($this,
    'include_login_scripts'));
  }
  public function include_scripts_styles(){
    wp_register_script('wpwa_ajax', plugins_url('js/wpwa-ajax.js',
    dirname(__FILE__)), array("jquery"));
    wp_enqueue_script('wpwa_ajax');
    $nonce = wp_create_nonce("unique_key");
    $ajax = new WPWA_AJAX();
    $ajax->initialize();
    $config_array = array(
      'ajaxURL' => admin_url('admin-ajax.php'),
      'ajaxActions' => $ajax->ajax_actions,
      'ajaxNonce' => $nonce,
      'siteURL' => site_url(),
    );
    wp_localize_script('wpwa_ajax', 'wpwa_conf', $config_array);
    wp_register_style('user_styles', plugins_url('css/wpwa-
    style.css', dirname(__FILE__)));
```

```
        wp_enqueue_style('user_styles');
        wp_register_style('wpwa_theme', plugins_url('css/wpwa-
        theme.css',dirname(__FILE__)));
        wp_enqueue_style('wpwa_theme');
        wp_register_script('developerjs', plugins_url('js/wpwa--
        developer.js', dirname(__FILE__)), array('backbone'));
        wp_enqueue_script('developerjs');
        $config_array = array(
          'ajaxUrl' => admin_url('admin-ajax.php')
        );
        wp_localize_script('developerjs', 'wpwaScriptData',
        $config_array);
      }
      public function include_admin_scripts_styles(){
        wp_enqueue_script('jquery');
        if (function_exists('wp_enqueue_media')) {
          wp_enqueue_media();
        } else {
          wp_enqueue_style('thickbox');
          wp_enqueue_script('media-upload');
          wp_enqueue_script('thickbox');
        }
        wp_register_script('wpwa_file_upload', plugins_url('js/wpwa-
        file-uploader.js', dirname(__FILE__)), array("jquery"));
        wp_enqueue_script('wpwa_file_upload');
        // wp_enqueue_style('my-admin-theme', plugins_url('css/wp-
        admin.css', dirname(__FILE__)));
      }
      public function include_login_scripts(){
        // wp_enqueue_style('my-admin-theme', plugins_url('css/wp-
        admin.css', dirname(__FILE__)));
      }
    }
```

As you can see, all the script and style files are encapsulated into three actions named `wp_enqueue_scripts`, `admin_enqueue_scripts`, and `login_enqueue_scripts`. An explanation of the preceding code is not necessary as we only moved the scripts and styles from the components to a common controller.

But, there are two changes that need to be discussed in this context. First, we have to use `dirname(__FILE__)` instead of `__FILE__` as the scripts and styles are loaded from a file inside a subfolder. Next, we don't have the access to the `ajax_actions` instance variable of the `WPWA_AJAX` class to configure the data for the AJAX script. Therefore, we have to initialize an object of the `WPWA_AJAX` class and call the `initialize` function to make things work as before.

At this stage, we don't have a function named `initialize` inside the `WPWA_AJAX` class. The implementation and the necessity for creating the function will be discussed later.

Here is the code for initializing the script controller from the main plugin class:

```
require_once 'controllers/class-script-controller.php';
$script_controller = new Script_Controller();
$script_controller->enque_scripts();
```

So far, we have created three controllers for merging the common controlling code. The remaining controllers will contain the actions necessary for working with the admin area.

Step 13 – building the admin menu controller

Throughout this application, we used admin-menu-related actions inside the `wpwa-xml-rpc-api` and `wpwa-dashboard` components. So, we can merge them into a common controller named `Admin_Menu_Controller`. This controller offers fewer functionalities compared to the other three controllers created in the earlier sections. Let's take a look at the implementation using the following code:

```
class Admin_Menu_Controller{
  public function initialize_admin_menu(){
    add_action('admin_menu', array($this, 'execute_admin_menu'));
  }
  public function execute_admin_menu(){
    $xml_rpc = new WPWA_XML_RPC_API();
    $xml_rpc-> api_settings();
    $dashboard = new WPWA_Dashboard();
    $dashboard-> customize_main_navigation();
  }
}
```

We have re-used the `admin_menu` action in the preceding code. The menu-handling functions are different in both occasions, and hence we call the menu-handling functions inside the respective components. We can invoke `Admin_Menu_Controller` using a similar set of code from the main plugin file, and hence I am going to omit the code and explanations.

Step 14 – creating class initializations

Up to this point, we have created objects of most of the component classes. As the application grows larger, we might need more objects to execute specific functionalities. We used to define WordPress action hooks inside the class controller. Whenever an object is created, the constructor will redefine the set of actions, creating unnecessary performance overhead. Therefore, we need to remove all the action definitions from the component constructors. So far, we have moved the common actions into controllers. But we have more actions left inside the constructors of each component.

We have to create an `initialize` function for each of the component main classes and include all the remaining actions within the `initialize` function. Once completed, all the component constructors will be free from action definitions.

We are not going to include all the code for the `initialize` functions due to the extensive length. You can find the complete code modification inside the source code folder. In the original code, we had object initializations inside each main plugin file. Now we have to remove those initializations as we are requesting objects from several locations.

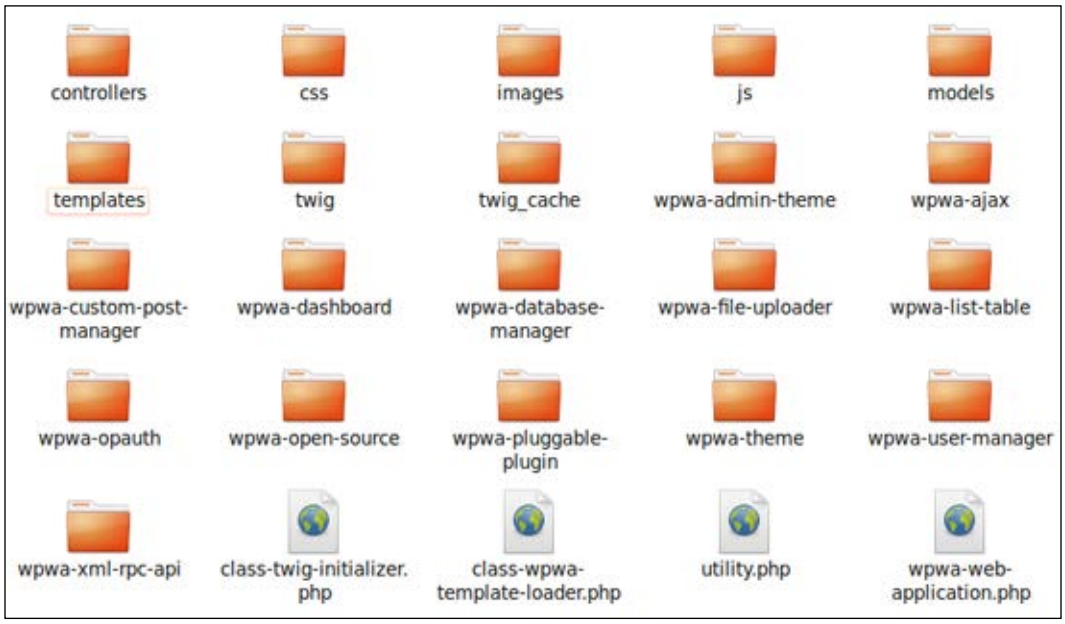
Step 15 – initializing application controllers

In the preceding section, we removed all the initialization of objects and created a function called `initialize` to contain the remaining actions. Now we need to invoke the initialize functions to execute the remaining actions. Therefore, we use the `initialize_app_controllers` function of the `WPWA_Web_Application` class to invoke the initialize functions of all the existing components as illustrated in the following code:

```
public function initialize_app_controllers(){
  $xml_rpc = new WPWA_XML_RPC_API();
  $xml_rpc->initialize();
  $user_manager = new WPWA_User_Manager();
  $user_manager->initialize();
  $app_theme = new WPWA_Theme();
  $app_theme->initialize();
  $open_source  = new WPWA_Open_source();
  $open_source->initialize();
  $opauth = new WPWA_Opauth();
  $opauth->initialize();
  $file_uploader = new WPWA_File_Uploader();
  $file_uploader->initialize();
```

```
$dashboard = new WPWA_Dashboard();
$dashboard->initialize();
//$dashboard->set_frontend_toolbar(FALSE);
$ajax = new WPWA_AJAX();
$ajax->initialize();
$base_path = plugin_dir_path(__FILE__);
require_once $base_path . 'class-twig-initializer.php';
$custom_posts = new WPWA_Custom_Post_Manager();
$custom_posts->initialize();
}
```

The object of each component is instantiated to invoke the `initialize` function. The custom post manager function uses `Twig_Initializer` for loading Twig templates. So, we have to move the `class-twig-initializer.php` file into the root folder of the plugin and include it inside the `initialize_app_controllers` function before invoking the `initialize` function. The following screenshot shows the folder structure after the completion of the preceding 15 steps:



Throughout this 15-step process, we have completed the main tasks of restructuring the process to separate the concerns and building a maintainable code structure.

Let's try to find the potential impacts caused by restructuring and how to solve them.

Restructuring the custom post manager

The custom post manager contains the Models and Twig templates library. First, we have to copy all the models into the `models` folder of the root plugin. The autoloader functionality is implemented in the main plugin file, and hence we have to move both the `twig` and `twig_cache` folders into the root folder as well. In step 15, we initialized the `Twig_Initializer` class from the main file. So, the `initialize` function of `wpwa-custom-post-manager` should be updated to the following code:

```
public function initialize(){
  $this->template_parser =
  Twig_Initializer::initialize_templates();
  $this->services = new WPWA_Model_Service($this-
  >template_parser);
  $this->projects = new WPWA_Model_Project($this-
  >template_parser);
  $this->books = new WPWA_Model_Book($this->template_parser);
  $this->articles = new WPWA_Model_Article($this-
  >template_parser);
}
```

The main plugin is responsible for loading the `twig_initializer.php` file, and hence we can directly invoke the `initialize_templates` function of the `$this->template_parser` class. The other sections of code remain the same as the original code.

Integrating a template loader into the user manager

In *Chapter 2*, *Implementing Membership Roles, Permissions, and Features*, we used direct file inclusions to load the necessary templates. In *Chapter 4*, *The Building Blocks of Web Applications*, we improved the loading of templates by introducing a common templates loader. Now we can integrate the template loader into the user manager component to keep the code consistent. First, we'll look at the template-loading code used in the `class-wpwa-user-manager.php` file:

```
include dirname(__FILE__) . '/templates/info.php';
include dirname(__FILE__) . '/templates/login.php';
include dirname(__FILE__) . '/templates/register.php';
```

Now let's look at the modified code for template loading in the user manager component:

```
$tmp = new WPWA_Template_Loader();
$tmp->render("info", array("message"=>$message));
$tmp = new WPWA_Template_Loader();
$tmp->render("login", array("errors"=>$errors));
$tmp = new WPWA_Template_Loader();
$tmp->render("register", array("errors"=>$errors));
```

With the implementation of the new process, we can pass the template data instead of relying on the global data. So, the template variables need to be changed from `$message` to `$data['message']` and `$errors` to `$data['errors']`.

We have completed the restructuring process and fixed the potential impacts to the existing components. There can also be several other impacts, which we haven't discussed here. Feel free to find them and discuss on the book's website.

WordPress has emerged as a web development framework in recent years. But still there are a very limited number of WordPress applications compared to generic blogs or websites. So, the best practices and design patterns have not been discussed or implemented for web application development. Here, we have discussed a possible technique for structuring applications. Still, there is a lot of scope for improving the current design. On the other hand, you might have a better structuring process for developing web applications with WordPress. So, I invite you to discuss your preferred structuring process on the book's website and help improve WordPress as a web application development framework.

Working with a restructured application

Having completed the restructuring process, we now have to understand the process of creating new functionalities from scratch. So, in this section, we are going to build the **Developer List** page with an autocomplete search using AJAX. Let's get started with the requirements planning.

There can be many lists within the portfolio management application. So, we need a new rewrite rule for implementing list-based pages. Then, we need a separate template for displaying the data for the developer list. All the existing developers will be displayed in the initial page load. Users can then use the autocomplete textbox to run a search on the developers. The list will be updated on the jQuery `keyup` event of the textbox to filter the list of developers using the search string.

We have to start the process by adding a new rewriting rule to WordPress. Remember that we merged all the rewriting rules into the `manage_routing_rules` function of the `Template_Router` class. So, let's look at the updated code with the inclusion of a new rule for lists:

```
public function manage_routing_rules() {
  add_rewrite_rule('user/([^/]+)/([^/]+)/?',
  'index.php?control_action=$matches[1]&record_id=$matches[2]',
  'top');
  add_rewrite_rule('user/([^/]+)/?',
  'index.php?control_action=$matches[1]', 'top');
  add_rewrite_rule('list/([^/]+)/?',
  'index.php?control_action=$matches[1]', 'top');
}
```

Once the rule is defined, we can match all the list-based pages with the `control_action` parameter. In this scenario, our URL will be `/list/developers`, and hence we need to match the `control_action` parameter to the value of the developers. Now we have to update the `front_controller` function with the new control action as shown in the following code:

```
case 'developers':
  $developer = new WPWA_Model_Developer();
  $result = $developer->list_developers();
  $tmp = new WPWA_Template_Loader();
  $tmp->render("developer_list", array("developers" => $result));
  exit;
  break;
```

Inside the matching case, we have the code for retrieving the default developer list and assigning the result to the template. So, let's start by creating the developer model.

Building the developer model

Models are mainly used for handling application logic and working with the database. We created four models in the custom post manager for projects, books, articles, and services. The developer role is also one of the major roles in the application, and hence we need a separate model to handle developer-specific functionalities. So, let's create a new file named `class-wpwa-model-developer.php` inside the `models` folder.

The autoloader created in the earlier section uses a prefix named `WPWA_Model_` to load the models, and hence we are going to name the new model class `WPWA_Model_Developer`. Let's take a look at the initial implementation of the developer model with the default data retrieval function:

```
class WPWA_Model_Developer {
  public function list_developers() {
    $user_query = new WP_User_Query(array('role' => 'developer',
    'number' => 25));
    return $user_query->results;
  }
}
```

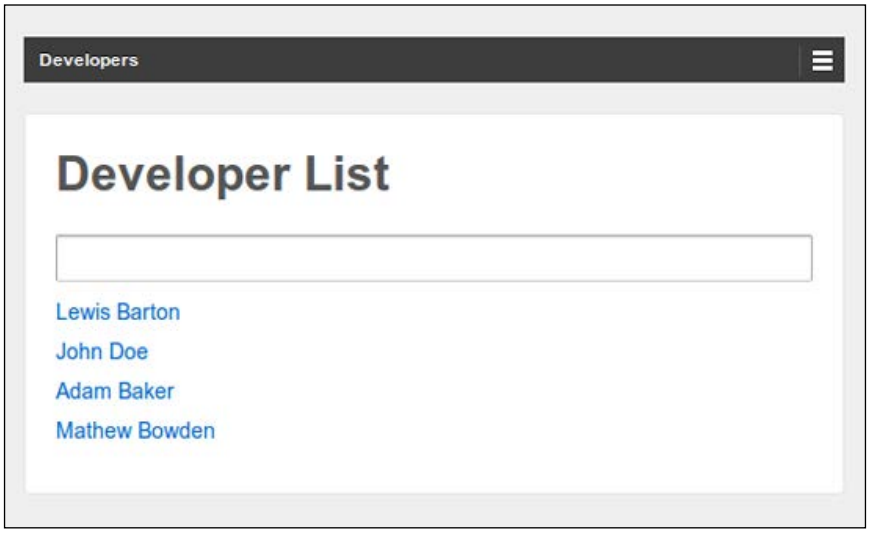
Here, we use the `WP_User_Query` class to retrieve 25 developers to be displayed on the initial page load. Next, we have to create a template to display the data retrieved from the `list_developers` function.

Designing the developer list template

With the new structure, all the template files are located inside the `templates` folder inside the root plugin folder. The developer list template should contain an autocomplete textbox and a dynamic panel for displaying the developer list. So, let's create a new template file named `developer_list.php` as shown in the following code:

```
<?php get_header(); ?>
<div class='main_panel'>
  <div class='developer_profile_panel'>
    <h2>Developer List</h2>
    <div class='field_label'><input type="text"
    id="autocomplete_dev_list"
    name="autocomplete_dev_list" /></div>
  </div>
  <div id='developer_list'>
    <?php foreach($data['developers'] as $developer){ ?>
    <div class="developer_row">
    <a href="<?php echo site_url();?>user/profile/<?php echo
    $developer->data->ID; ?>">
    <?php echo esc_html($developer->data->display_name);?>
    </a>
    </div>
    <?php } ?>
  </div>
</div>
<?php get_footer(); ?>
```

Once the preceding code is implemented, you can use `http://www.yoursite.com/list/developers` to access the **Developer List** page. The following screenshot previews the **Developers** page with the initial dataset:



Enabling AJAX-based filtering

AJAX-based filtering is becoming a trend in web development, and it is used by many popular sites such as Facebook and Google. Here, we are going to use the reusable plugin created in *Chapter 5*, *Developing Pluggable Modules*, to implement filtering for the developer list. Once the user enters a key inside the textbox, we use the value of the textbox as the search string to retrieve the list of developer records instantly without refreshing the page. We already have a specific JavaScript file for developers. Let's update the `developer.js` file with the following code to enable AJAX filtering:

```
$jq(document).ready(function(){
  $jq("#autocomplete_dev_list").keyup(function(){
    ajaxInitializer({
      "success":"ajax_developer_list",
      "dataType":"json",
      "data":{
        "search":$jq(this).val(),
        "action" : conf.ajaxActions.developer_list.action
      }
    });
  });
});
```

We invoke the `ajaxInitializer` function on the `keyup` event of the `#autocomplete_dev_list` textbox. This function encapsulates the complex and recurring tasks of making AJAX requests, allowing us to focus on handling the result. Here, we have used JSON as the data type and `ajax_developer_list` as the success handler function. The data attribute contains the search string and the corresponding server-side action. Notice that we are using the `WPWA_AJAX` class to configure the server-side actions. So, let's see the updated code for the server-side actions:

```
public function configure_actions() {
  $this->ajax_actions = array(
  "sample_key" => array("action" => "sample_action", "function" =>
  "sample_function_name"),
  "sample_key1" => array("action" => "sample_action1", "function"
  => "sample_function_name1"),
  "developer_list" => array("action" => "developer_list",
  "function" => "ajax_developer_list"),
  );
  // Remaining code
}
```

We have configured a new action named `developer_list` to invoke the function `ajax_developer_list`. Since we haven't specified a parameter for logged-in users, this action will be executed for both guest users as well as logged-in users.

Here, we are going to implement the `ajax_developer_list` function inside the `WPWA_AJAX` class. With the current structure, it's difficult to call the AJAX function from an external class. Try to restructure the AJAX component to enable AJAX function invocations for external classes.

Let's go through the implementation of the `ajax_developer_list` function for completing the server-side process of filtering developers:

```
public function ajax_developer_list() {
  global $wpdb;
  $search_val = isset($_POST['search']) ? $_POST['search'] : "";
  $sql = "SELECT u.ID, u.user_login, u.display_name, u.user_email
  FROM $wpdb->users u
  INNER JOIN $wpdb->usermeta m ON m.user_id = u.ID
  WHERE m.meta_key = 'wp_capabilities'
  AND m.meta_value LIKE '%developer%'
  AND u.display_name LIKE '%$search_val%'
  ";
  $userresults = $wpdb->get_results($sql);
  $result = array();
  foreach ($userresults as $val) {
```

```
        array_push($result, array("id" => $val->ID, "name" => $val-
        >display_name));
      }
      echo json_encode($result);
      exit;
    }
```

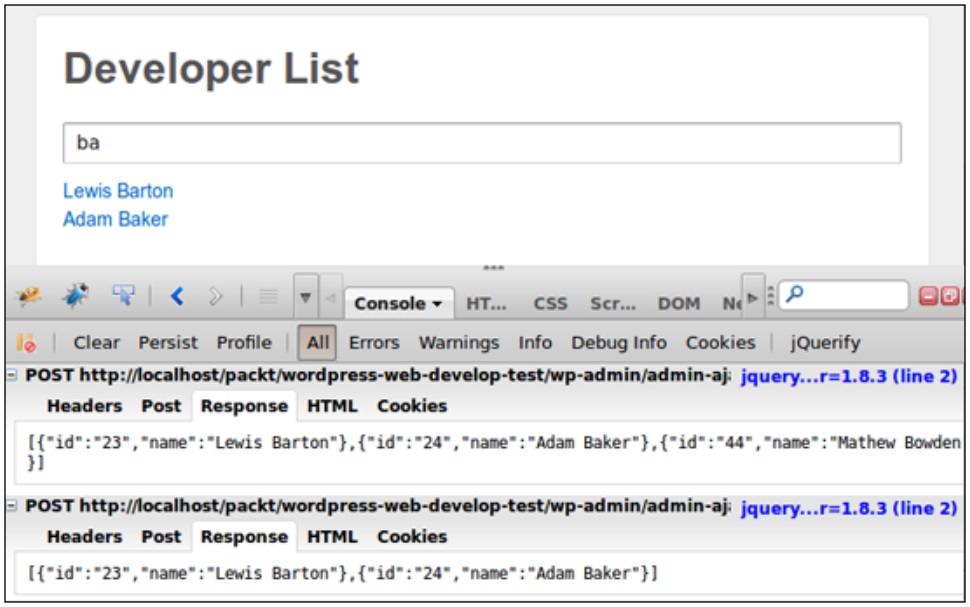
First, we retrieve the search string from the `$_POST` data for the request. Then, we use the custom query by joining the `users` table and the `usermeta` table to search by the display name. We can check for the necessary user roles using a `LIKE` statement on the `wp_capabilities` key of the `usermeta` table. Here, we had to use a custom query as `WP_User_Query` does not offer built-in support for searching the `display_name` column.

Finally, we send the result back to the browser in a simplified manner using a JSON-encoded string. Now, we can move back to the JavaScript code to implement the success handler for completing the request. So, have a look at the following code:

```
var ajax_developer_list = function(result){
  $jq("#developer_list").html("");
  $jq.each(result, function(i, val) {
    //console.log(val);
    $jq("#developer_list").append('<div class="developer_row">\n\
    <a href="'+conf.siteURL+'/user/profile/'+val.id+'">\n\
    '+val.name+'\n\
    </a>\n\
    </div>');
  });
};
```

Once the result is retrieved, we empty the existing list of developer records using the jQuery `html` function. Next, we use the `append` function to append the filtered records back into the list while traversing through the dataset using the jQuery `each` statement.

Now we have completed the filtering process for the developer list. Users can enter the search text inside the textbox to filter values. Consider the following screenshot for the filtered developer list on entering two keys:



As shown in the preceding screenshot, an AJAX request is made whenever you enter a new character into the textbox. This can really reduce the performance in situations where you have a large amount of database records. So, it's important to use this technique wisely when optimizing a database with indexing.

Up until this point, we have restructured the application and looked at the possible ways of working with the restructured version for new requirements. In the previous chapters, we left some tasks incomplete. So, here we are going to complete the foundation of a portfolio application by completing those remaining tasks. In the next two sections, we are going to complete the developer profile page with additional fields and move subscriber notifications to WordPress scheduling instead of creating notifications on post publishing. So let's get started.

Updating a user profile with additional fields

A developer profile page was created in *Chapter 8*, *Enhancing the Power of Open Source Libraries and Plugins*, with the use of `Backbone.js` and `Underscore.js`. The profile section of this page was limited to the name of the user as we had very limited information for users. Here, we are going to capture more information by using additional fields on the profile page of the WordPress dashboard. So, let's update the `initialize` function of the `WPWA_User_Manager` class to add the necessary actions for editing the profile:

```
public function initialize() {

  // Other actions
  $user = new WPWA_Model_User();
  add_action('show_user_profile', array($user,
  "add_profile_fields"));
  add_action('edit_user_profile', array($user,
  "add_profile_fields"));
}
```

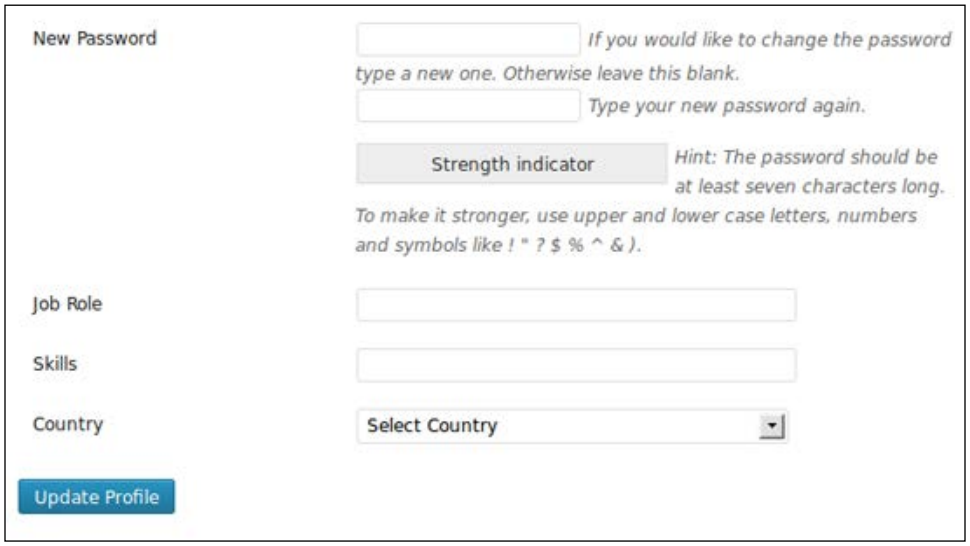
The profile editing page is common for all the user roles in the portfolio application. So, we need a new model class to work with user-related functions. We have created a new class named `WPWA_Model_User` inside the `models` folder. Then, we have defined two actions to be executed on the user profile page. Both the `show_user_profile` and `edit_user_profile` actions are used to add new fields to the end of the user edit form. According to the preceding code, the addition of new fields will be implemented in the `add_profile_fields` function of the `WPWA_Model_User` class. Let's look at the implementation of the `add_profile_fields` function:

```
class WPWA_Model_User {
  public function add_profile_fields() {
  global $user_ID;
  $tmp = new WPWA_Template_Loader();
  $tmp->render("profile_fields", array());
  }
}
```

Inside the `add_profile_fields` function, we can load a new template using our template loader to contain the HTML code for the new fields. The following code contains the additional fields inside the `profile_fields` template:

```
<table class="form-table">
  <tr>
    <th><label for="Job Role">Job Role</label></th>
    <td><input type="text" class="regular-text" value="<?php echo
    $data['job_role']; ?>" id="job_role" name="job_role"></td>
  </tr>
  <tr>
    <th><label for="Skills">Skills</label></th>
    <td><input type="text" class="regular-text" value="<?php echo
    $data['skills']; ?>" id="skills" name="skills"></td>
  </tr>
  <?php
  $countries = array(
    'AF' => 'Afghanistan',
    'AL' => 'Albania',
  );
  ?>
  <tr>
    <th><label for="country">Country</label></th>
    <td>
      <select name="country" id="country">
        <option value="" >Select Country</option>
        <?php foreach($countries as $country){ ?>
        <option <?php echo ($country == $data['country'])?
        "selected": "";?> value="<?php echo $country;?>"><?php
        echo $country;?></option>
        <?php } ?>
      </select>
    </td>
  </tr>
</table>
```

Basically, we have three fields for storing the job role, skills, and country of the developers. At this stage, the `$data` array is empty, and hence these fields won't have default values. It's important to use the CSS class `form-table` to maintain the consistency of designs with the existing fields. Now, your profile page should look something similar to the following screenshot:



Updating values of profile fields

Once the user clicks on the **Update Profile** button, all the custom fields need to be saved automatically into the database. So, we have to define two other actions named `edit_user_profile_update` and `personal_options_update` as shown in the following code:

```
add_action('edit_user_profile_update', array($user,
"save_profile_fields"));
add_action('personal_options_update', array($user,
"save_profile_fields"));
```

The action hooks defined in the preceding code are generally used to update additional profile fields. So, we invoke the `save_profile_fields` function of the user model to cater to the persisting tasks. Consider the implementation of the `save_profile_fields` function as shown in the following code:

```
public function save_profile_fields() {
  global $user_ID;
  $job_role = isset($_POST['job_role']) ?
  esc_html(trim($_POST['job_role'])) : "";
  $skills = isset($_POST['skills']) ?
  esc_html(trim($_POST['skills'])) : "";
  $country = isset($_POST['country']) ?
  esc_html(trim($_POST['country'])) : "";
  update_user_meta($user_ID, "_wpwa_job_role", $job_role);
  update_user_meta($user_ID, "_wpwa_skills", $skills);
  update_user_meta($user_ID, "_wpwa_country", $country);
}
```

The custom profile fields will be stored inside the `wp_usermeta` table, and hence we use the `update_user_meta` function to save the values grabbed from the `$_POST` array. Once the custom profile field values are updated, we need to display the existing values on the profile page. Earlier, we used an empty array to load the `profile_fields` template. Now, we can look at the updated version of the function to pass the necessary data to the `profile_fields` template to be displayed on the profile page:

```
public function add_profile_fields() {
  global $user_ID;
  $job_role = esc_html(get_user_meta($user_ID, "_wpwa_job_role",
  TRUE));
  $skills = esc_html(get_user_meta($user_ID, "_wpwa_skills",
  TRUE));
  $country = esc_html(get_user_meta($user_ID, "_wpwa_country",
  TRUE));
  $tmp = new WPWA_Template_Loader();
  $tmp->render("profile_fields",
  array("job_role"=>$job_role,"skills"=>$skills,
  "country"=>$country));
}
```

With the new implementation, we can access the template variables inside the template using the `$data` array. Having completed the profile field creation, we can now move on to our main goal of displaying the profile details on the developer profile page at the frontend. We can easily use the `get_user_meta` function to retrieve the necessary profile details. Let's look at the updated version of the `create_developer_profile` function of the `WPWA_Open_source` class:

```
public function create_developer_profile($developer_id) {
  $user_query = new WP_User_Query(array('include' =>
  array($developer_id)));
  $data = array();
  foreach ($user_query->results as $developer) {
    $data['display_name'] = $developer->data->display_name;
    $data['job_role'] = get_user_meta($developer->data->ID,
    "job_role", TRUE);
    $data['skills'] = get_user_meta($developer->data->ID,
    "skills", TRUE);
    $data['country'] = get_user_meta($developer->data->ID,
    "country", TRUE);
  }
  $current_user = wp_get_current_user();

  $data['developer_status'] = ($current_user->ID ==
  $developer_id);
  $data['developer_id'] = $developer_id;
  $tmp = new WPWA_Template_Loader();
  $tmp->render("developer", $data);
  exit;
  }
```

Here, we have retrieved all the custom profile fields to be passed as template variables. Finally, the process will be completed by updating the `developer.php` template to include the profile field data as shown in the following code:

```
<div class='developer_profile_panel'>
  <h2>Personal Information</h2>
  <div class='field_label'>Full Name</div>
  <div class='field_value'><?php echo
  esc_html($data['display_name']); ?></div>
  <div class='field_label'>Country</div>
  <div class='field_value'><?php echo esc_html($data['country']);
  ?></div>
  <div class='field_label'>Job Role</div>
```

```
    <div class='field_value'><?php echo esc_html($data['job_role']);
    ?></div>
    <div class='field_label'>Skills</div>
    <div class='field_value'><?php echo esc_html($data['skills']);
    ?></div>
</div>
```

Now, go to the browser and access `/user/profile/{user id}` and you will get a screen similar to the following screenshot:

So, we have completed the first of the two tasks for finalizing the basic foundation of the portfolio application. In the next section, we will complete the implementation of the application discussed in this book by developing the subscriber notification scheduling.

Scheduling subscriber notifications

Sending notifications is a common task in any web application. In this scenario, we have subscribers who want to receive e-mail updates about developer activities. In *Chapter 8*, *Enhancing the Power of Open Source Libraries and Plugins*, we created a simple e-mail notification system on post publish. Notifying subscribers on post publish can become impossible in a situation where you have a large number of subscribers. Therefore, we are going to take a look at the scheduling features of WordPress for automating the notification-sending process.

As a developer, you might be familiar with cron, which executes certain tasks in a time-based manner. WordPress scheduling functions offer the same functionalities with lesser flexibility. Let's see how to schedule subscriber notifications for predefined time intervals using the `wp_schedule_event` function of WordPress:

```
wp_schedule_event($timestamp, $recurrence, $hook, $args);
```

The preceding code illustrates the basic implementation of the `wp_schedule_event` function. The first parameter defines the starting time of the cron job. The next parameter defines the time interval between the executions of cron. WordPress provides built-in time intervals named `hourly`, `twice daily`, and `daily`. Also, we can add custom time intervals to the existing list of values. The third parameter defines the hook to be executed for providing the results of the cron. You should use a unique name for the hook. The last parameter defines the arguments to the hook, which we can keep blank in most occasions.

The `wp_schedule_event` function initializes the recurring function executions, and hence should be avoided inside a hook such as `init` that gets executed on every request. Ideally, scheduling events should be done inside the plugin activation handler.

Let's schedule subscriber notifications by updating the `Activation_Controller` class as shown in the following code:

```
public function execute_activation_hooks() {
  wp_schedule_event(time(), 'everytenminutes',
  'notification_sender');
// Remaining code
}
```

Inside the activation hook, we have initialized the scheduled event using the `wp_schedule_event` function. The activation time of the plugin is used as the starting time of the scheduled event. We have used a custom interval named `everytenminutes` to execute the task at ten-minute intervals. Since this is a custom interval, we have to add it to the existing schedules before using it. Finally, we have the hook named `notification_sender` for executing the custom functionality. Next, we need to add the custom time interval into the existing schedules list.

This notification-related functionality is common to all parts of an application, and hence we cannot specify a model for the implementation. Generally, we use these kind of functionalities in a utility class or a file. Here, we are going to create a separate file for handling utility functions. Create a file named `utility.php` inside the plugin root folder and include it inside the `wpwa-web-application.php` file. Let's begin with the implementation of the custom interval:

```
function everytenminutes($schedules) {
  $schedules['everytenminutes'] = array(
    'interval' => 60*10,
    'display' => __('Once Ten Minutes')
  );
return $schedules;
}
add_filter('cron_schedules', 'everytenminutes');
```

WordPress allows the customization of schedules using the `cron_schedules` filter with the preceding syntax. We have added a ten-minute schedule using 600 seconds as the time interval. Now, the schedules list will have four values including the ten-minute interval. Next, we have to implement the `notification_sender` hook for sending notifications to subscribers.

Notifying subscribers through an e-mail

This process is far more complex compared to the notification procedure used earlier with the publishing of posts. Here, we need to cater to the following list of tasks to automate the sending of notifications:

- Add a custom status on post publish to identify new posts
- Grab the new posts within the time interval using a custom status for posts
- Get the list of subscribers for the author of each post
- Send notifications to filtered subscribers

First, we need to find a way to track the posts that have to be notified and also the ones that have already been notified. Therefore, we are going to use a custom post meta value to track the notified status. You should remember the following code, which was used to send notifications on post publish:

```
add_action('new_to_publish', array($this,
'send_subscriber_notifications'));
add_action('draft_to_publish', array($this,
'send_subscriber_notifications'));
add_action('pending_to_publish',array($this,
'send_subscriber_notifications'));
```

Now, we are going to update the `send_subscriber_notifictions` function to suit the new process as shown in the following code:

```
public function send_subscriber_notifictions($post) {
  update_post_meta($post->ID, "notify_status", "0");
}
```

Here, we have removed the e-mail-sending functionality and updated a post meta value named `notify_status`, which contains the value `0`. It means the post is new and the subscribers haven't yet been notified.

Next, we are going to look at the implementation of the `notification_sender` hook inside the `utility.php` file. Let's have a look at the following code:

```
add_action("notification_sender", "notification_send");
function notification_send() {
  global $wpdb;
  require_once ABSPATH . WPINC . '/class-phpmailer.php';
  require_once ABSPATH . WPINC . '/class-smtp.php';
  $phpmailer = new PHPMailer(true);
  $phpmailer->From = "example@gmail.com";
  $phpmailer->FromName = "Portfolio Application";
  $phpmailer->SMTPAuth = true;
  $phpmailer->IsSMTP(); // telling the class to use SMTP
  $phpmailer->Host = "ssl://smtp.gmail.com"; // SMTP server
  $phpmailer->Username = "example@gmail.com";
  $phpmailer->Password = "password";
  $phpmailer->Port = 465;
  $phpmailer->IsHTML(true);
  $phpmailer->Subject = "New Schedule";
  // Remaining code
}
```

We have defined the `notification_sender` action with a function named `notification_send`. The first part of the function contains the necessary code for initializing the `PHPMailer` class to send e-mails. Afterwards, we have to grab the posts having `notify_status` meta value as `0` for sending e-mails to subscribers. Consider the next part of the code for retrieving posts based on `notify_status`:

```
function notification_send() {
  global $wpdb;
  // Initial code
  $args = array(
    'post_type' => array('wpwa_service', 'wpwa_book',
    'wpwa_project', 'wpwa_article'),
    'post_status' => 'publish',
    'meta_query' => array(
      array(
        'key' => 'notify_status',
        'value' => '0'
      )
    )
  );
  $post_query = null;
  $post_query = new WP_Query($args);
  $message = "";
}
```

The `WP_Query` class is used to retrieve all the published books, articles, projects, and services with a `notify_status` value of `0`. Finally, we have to use the following code inside this function for sending e-mails to subscribers:

```
function notification_send() {
  global $wpdb;
  // Initial code
  if ($post_query->have_posts()) : while ($post_query-
  >have_posts()) : $post_query->the_post();
  $author = get_the_author_ID();
  $sql = "SELECT user_nicename,user_email
  FROM $wpdb->users
  INNER JOIN " . $wpdb->prefix . "subscribed_developers
  ON " . $wpdb->users . ".ID = " . $wpdb->prefix .
  "subscribed_developers.follower_id
  WHERE " . $wpdb->prefix . "subscribed_developers.developer_id =
  '$author'";
  $subscribers = $wpdb->get_results($sql);
  $message.= "<a href='" . get_permalink() . "'>" .
  get_the_title() . "</a>";
  foreach ($subscribers as $subscriber) {
    $phpmailer->AddBcc($subscriber->user_email, $subscriber-
    >user_nicename);
  }
  $phpmailer->Body = "New Updates from your favorite
  developers<br/><br/><br/>." . $message;
  $phpmailer->Send();
  update_post_meta(get_the_ID(), "notify_status", "1");
  endwhile;
  endif;
}
```

While looping through the posts list, we get the subscribers using the author (developer) of the post. Then, all the subscribers are added to the e-mail using the `AddBcc` function. The e-mail message contains the name of the post with a direct link to access inside the browser. Afterwards, we send an e-mail with the new updates. This process will be continued for each and every post that has a `notify_status` value of `0`. Once the e-mail is sent, we update the `notify_status` value to `1` to prevent duplicate notifications in the next schedule.

WordPress scheduling works similar to the cron jobs in Linux-based systems. But we have a limitation compared to normal cron jobs. WordPress scheduling is initialized based on user activities. Once a user accesses the application, WordPress will check for the available schedules. If the next scheduled time is already passed, WordPress will execute the hook. If there are no user actions within the application, schedules will not be executed until someone interacts with the application.

Finally, we have completed the process of building the basic foundation of the portfolio application. We looked at various different techniques in building a web-application-specific functionality. The process of developing this application will be continued on the book's website, and I hope you will follow the rest of the development.

In the next section, we are going to talk about a few features of WordPress that we have left out so far and yet are important in web application development.

Lesser-known WordPress features

Throughout this book, we have looked at the major components related to web application development. WordPress also offers some additional features that are rarely noticed by the developers' community. Let's get a brief introduction on the following lesser known features of WordPress:

- Caching
- Transients
- Testing
- Security

Caching

In complex web applications, performance becomes a critical task. There are various ways of improving the performance from the application level as well as the database level. Caching is one of the major features of the performance-improving process, where you keep the results of complex logic or larger files in the memory or database for quick retrievals. WordPress offers a set of functions for managing caching within applications. Caching is provided through a class named `WP_Object_Cache`, which can be used effectively to manage nonpersistent cache.

We can cache the data using the built-in `wp_cache_add` function as defined in the following code:

```
wp_cache_add( $key, $data, $group, $expire );
```

The cached data is added using a specific key as the first parameter. The `$group` parameter defines the group name of the cached data. It's somewhat similar to namespacing, where we are allowed to create the same class inside multiple namespaces. By defining the `$group` parameter, we allow the possibility of creating duplicate cache keys in different groups. The fourth and final parameter defines the expiry time for the cached data.

The cached data can be accessed using the `wp_cache_get` function as shown in the following code:

```
wp_cache_get( $key, $group );
```

The existing functions allow you to manage the caching functionality for simple use cases. However, this might not be the best solution for larger web applications.

The nonpersistent nature of the WordPress cache is a limitation where you lose all the cached data on a page refresh.

More information about WordPress caching can be accessed from the codex at `http://codex.wordpress.org/Class_Reference/WP_Object_Cache`.

Generally, developers prefer the automation of caching tasks using existing plugins. So, let's look at some of the most popular plugins to provide caching inside WordPress:

- The W3 Total Cache plugin available at `http://wordpress.org/plugins/w3-total-cache/`
- The WP Super Cache plugin available at `http://wordpress.org/plugins/wp-super-cache/`

These are the most popular caching plugins, exceeding over 7 million downloads combined. Developers have to get used to these plugins to cater to the performance of complex applications.

Transients

A WordPress transient API caters to the limitations of caching functions by providing a database-level cache for temporary time intervals. Compared to caching, transients are used by many developers to work with large web applications. Transient functions work in a manner similar to caching functions, where we have functions for setting and getting transient values. An example usage of transients is illustrated in the following code:

```
set_transient( $transient, $value, $expiration );
get_transient( $transient );
```

The syntax of transient functions is similar to caching functions with the exception of group parameters. Each and every transient value will be stored in the `wp_options` table as a single row. The following screenshot shows a typical database result set with transient values:

option_id	option_name	option_value	autoload
1572	_transient_timeout_project_error_message_95	1370667651	no
1597	_transient_timeout_project_error_message_119	1370675035	no
1593	_transient_timeout_project_error_message_116	1370674749	no
1591	_transient_timeout_project_error_message_115	1370674729	no
1587	_transient_timeout_project_error_message_114	1370674466	no
1589	_transient_timeout_project_error_message_113	1370674509	no
1585	_transient_timeout_project_error_message_110	1370674260	no
1583	_transient_timeout_project_error_message_109	1370674061	no
2768	_transient_timeout_plugin_slugs	1375832527	no
1098	_transient_timeout_gform_update_info	1368776863	no

If you are using external plugins, you will see a large number of existing transient values within your database. In situations where you need persistent cache, make sure you use transients instead of caching functions.

Testing

Application testing is another critical task that is used to identify potential defects before releasing the application to a live environment. Testing is mainly separated into two areas named unit testing and integration testing. Unit testing is used to test each small component independently from others, while integration testing is used to test the application with the combination of all the modules.

Compared to other popular frameworks, WordPress code is not the easiest to test. However, we can use PHPUnit for testing themes as well as plugins in WordPress. You can find a guide for working with PHPUnit at `http://make.wordpress.org/core/handbook/automated-testing/`.

WordPress provides a set of test cases for testing major features. Many developers have a limited knowledge of the existing test cases as they are not available inside the core. You can access the complete list of test cases at `http://unit-tests.svn.wordpress.org/trunk/tests/`. Make sure you improve your knowledge of testing WordPress by going through these test cases. Then, you can write test cases for your own plugins and themes for unit-testing purposes.

Security

In WordPress web applications, security is considered to be one of the major threats. Most people believe that WordPress is insecure as a large number of WordPress websites are hacked every day. But not many people know that the reason behind the hacking of WordPress sites is the lack of knowledge of the site administrators. Once the necessary security policies are implemented, we can use WordPress applications without major issues.

The WordPress codex provides a separate section named *Hardening WordPress* for defining the necessary security constraints. You can read this guide at `http://codex.wordpress.org/Hardening_WordPress`. The following are some of the common and most basic guidelines for securing WordPress applications:

- Update the core, plugins, and themes to the latest version and remove unused plugins and themes
- Check third-party plugins for malicious code before usage
- Move the `wp-config.php` file from the default directory
- Restrict access to WordPress core folders using the necessary permission levels (the BulletProof Security plugin can be used to restrict permissions)
- Use unique and strong usernames and passwords
- Limit admin access via SSH (Secure Shell) and/or whitelisted IPs

There can be unlimited ways of hacking web applications, and it's hard to imagine and plan for every possibility. Apart from the basic guidelines, we can also use popular and stable WordPress plugins to secure our application. Here is a list of the most popular security plugins provided in the WordPress plugin directory:

- The Better WP Security plugin available at `http://wordpress.org/plugins/better-wp-security/`

- The WP Security Scan plugin available at `http://wordpress.org/plugins/wp-security-scan/`
- The BulletProof Security plugin available at `http://wordpress.org/plugins/bulletproof-security/`
- The Secure WordPress plugin available at `http://wordpress.org/plugins/secure-wordpress/`

Time for action

Throughout this book, we have developed various practical scenarios to learn the art of web application development. We now have the final set of actions before we complete this book. By now, you should have all the knowledge to get started with WordPress web development. After reading this chapter, you need to try the following set of actions to gain experience with the process:

- Find out different ways of structuring WordPress for web applications
- Improve the AJAX-based list to contain more filtering options
- Figure out practical scenarios to implement caches and transients

Final thoughts

WordPress is slowly but surely becoming a trend in web application development. Developers are getting started on building larger applications by customizing existing modules and features. But there are a lot of limitations and a lack of resources for web development-related tasks. So, the best practices and design patterns are yet to be defined for building applications with WordPress.

In this book, we developed an application structure considering the best practices of general web application development. WordPress architecture is different from typical PHP frameworks, and hence this structure might not be the best solution. As developers, we want to drive WordPress into a fully-featured web application framework. So feel free to discuss your own application structures and techniques which can be used for WordPress applications on the website for this book at `http://www.innovativephp.com/wordpress-web-applications`.

The website for this book is designed to provide additional resources on top of the theories and techniques discussed in this book. Make sure you follow the website content as it will be updated regularly with resources related to WordPress web application development. Also, we are going to improve the basic portfolio application developed in this book into a large-scale application by discussing every possible scenario. Please provide your contribution to improve the functionality of the portfolio application.

Summary

We began this chapter with a bunch of plugins developed throughout the first nine chapters of the book. Our portfolio application lacked proper structure with the usage of these plugins. So, we discussed various techniques for restructuring and building common architecture to build web applications.

Once the restructuring process was completed, we implemented a few new requirements such as subscriber notifications, AJAX-based developer-list filtering, and additional user-profile fields to understand how to work with the restructured architecture with a standalone plugin.

Finally, we talked about the lesser-known features of WordPress, which are important for building successful web applications. Here, we have completed the book with the basic foundation of a portfolio application. Make sure you follow the guides on the website of this book for understanding more complex theories and techniques of developing web applications while completing the portfolio management system.

Next, you can take a look at the *Appendix*, *Configurations, Tools, and Resources*, to set up WordPress for this book, and other related resources to go through the contents of this book.

Configurations, Tools, and Resources

Configure and set up WordPress

WordPress is a CMS that can be installed in a few minutes with the help of an easy-to-follow tutorial. Throughout this book, we have implemented a personal portfolio management application with advanced users. This compact tutorial is intended to help you set up your WordPress installation with the necessary configurations to be compatible with the features of our application. Let's get started.

Step 1 – downloading WordPress

We are using WordPress 3.6 as it's the latest version available at the time of writing this book. So, we have to download Version 3.6 from the official website at:

`http://wordpress.org/download/`

Step 2 – creating the application folder

First, we need to create a folder for our application inside the web root directory. Then, we will extract the contents of the downloaded ZIP file into the application folder. Finally, we have to provide the necessary permissions to create files inside the application folder. Make sure you provide the write permission for the `wp-config.php` file before starting the installation. Generally, we can set the permission for directories as `755` and for files as `644`. You can learn more about WordPress file permissions at:

`http://codex.wordpress.org/Hardening_WordPress#File_Permissions`

Step 3 – configuring the application URL

Initially, our application will be running on a local machine with a local web server. There are ways of working in the local environment:

- Create a virtual host to run the application
- Use the localhost to run the application

Creating a virtual host

Virtual hosts, often referred to as *vhosts*, allow us to configure multiple websites inside a single web server. Also, to refer to our application, we can match a custom URL. This method should be preferred in web application development as the migration from a local to a real server becomes less complex.

Let's say we want to run the portfolio application as `www.developerportfolio.com`. All we have to do is configure a virtual host to point the application folder to `www.developerportfolio.com`. Once set up, this will call the local application folder instead of the actual online website.

By using the actual server URL for the virtual host, we can directly export the local database into the server without changes.

The following resources will help you to set up a virtual host on different operating systems:

- **Windows** (**Wamp**): `http://www.kristengrote.com/blog/articles/how-to-set-up-virtual-hosts-using-wamp`
- **Mac**: `http://goo.gl/mZfVCi`
- **Fedora**: `http://www.techchorus.net/setting-apache-virtual-hosts-fedora`
- **Ubuntu**: `https://www.digitalocean.com/community/articles/how-to-set-up-apache-virtual-hosts-on-ubuntu-12-04-lts`

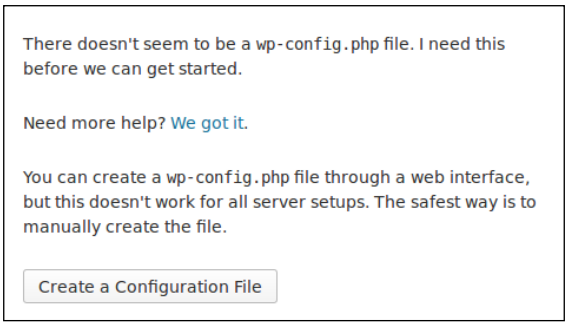
Using a localhost

A second and commonly used method is to use a localhost as the URL to access the web application. Once the application folder is created inside the web root, we can use `http://localhost/application_folder_name` to access the application.

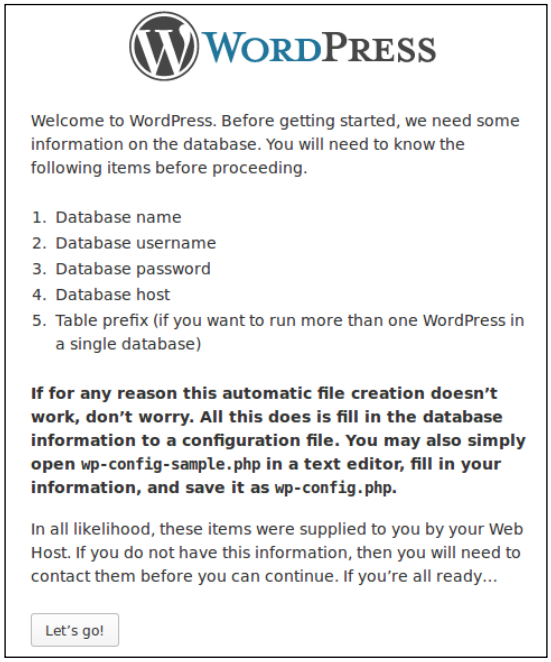
Step 4 – installing WordPress

Open a web browser and enter your application URL mentioned previously to get the initial screen of the WordPress installation process, as illustrated in the following screenshot.

We have to manually create the database before starting this installation process. So, create a new database from your favorite database editor and create a database user with the necessary permission to access the database.



Then click on the **Create a Configuration File** button which will load the screen shown in the following screenshot:

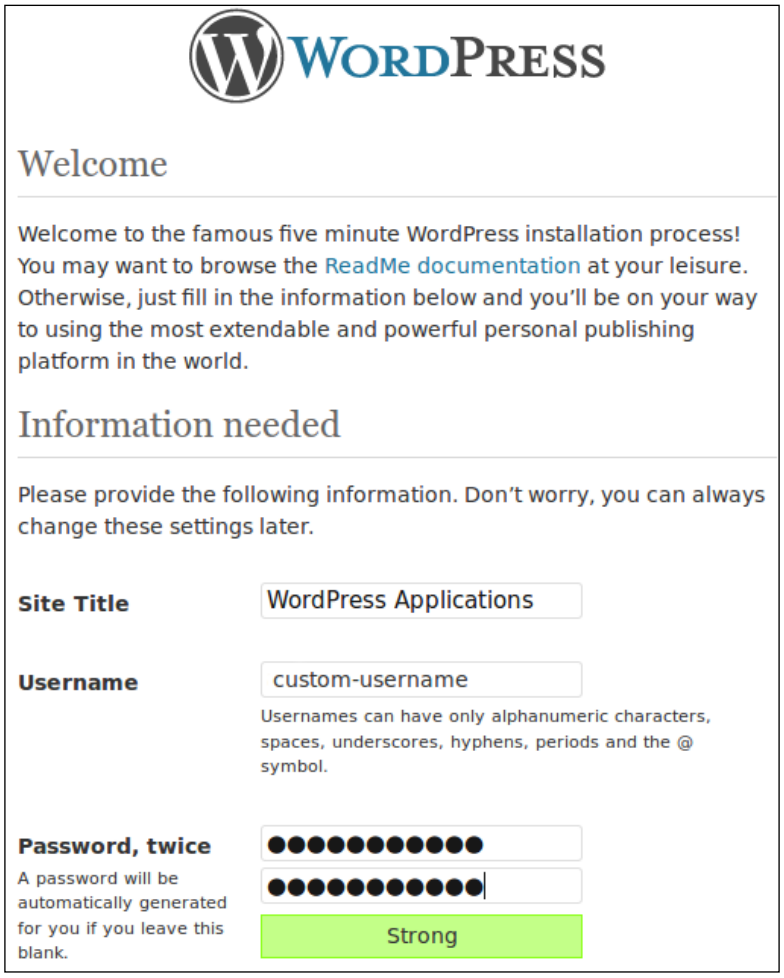


The preceding screenshot displays all the information needed to continue with the installation. After reading the contents, click on the **Let's go!** button to get the next screen as shown in the following screenshot:

Here, we have to enter the details to connect to the database. Use the details in the database-creation process to define the database name, user, password, and database host. Finally, we have to enter the table prefix. By default, WordPress uses `wp_` as the prefix. It's ideal to set a custom prefix like a random string for your tables to improve the security of your application. Once all the details are entered, click on the **Submit** button to get the next screen as shown in the following screenshot:

Also, we can use the WP Better Security plugin to generate a random prefix and update the database.

Click on the **Run the install** button to get the next screen as shown in the following screenshot:

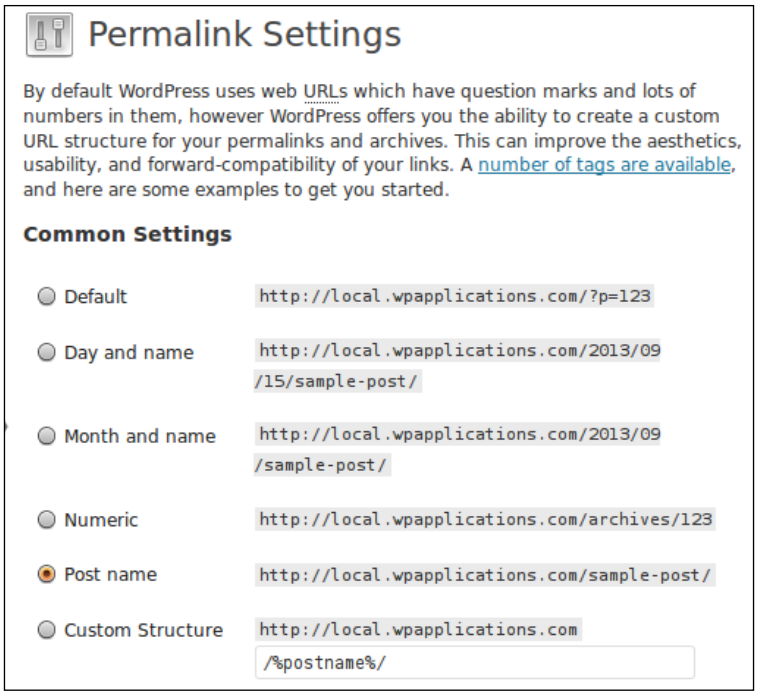


Fill in the form with the requested details. By default, WordPress uses `admin` as the username. Ideally, you should be using a custom username as the admin role to improve the security of the application. Once all the details are filled in, submit the form to complete the installation and you will get the following screenshot:

The details of your admin account will be displayed on this screen. Click on the **Log In** button to get the login form and log in to the admin area. Now we are ready to go.

Step 5 – setting up permalinks

Permalinks allow you to define the custom URL structure for your posts, pages, and custom URLs using `mod_rewrite`. By default, WordPress uses query parameters to load posts and pages through IDs. Usually, we change the existing URL structure to provide a neat URL. So, navigate to **Settings** | **Permalink** on the admin menu and you will find different URL structures. Select the **Post name** option for the URL and click on the **Save** button. Your screen should look something similar to the following screenshot:



Step 6 – downloading the Responsive theme

We have used a free theme called Responsive for the developer portfolio management application of this book. We can download the Responsive theme from the official WordPress themes directory at `http://wordpress.org/themes/responsive`. Then, we have to copy the extracted theme folder into the `/wp-content/themes` directory of our application.

Step 7 – activating the Responsive theme

Now we have to activate the theme from the WordPress admin panel. Choose **Apperence** | **Themes** from the left menu and click on the **Activate** link under the Responsive theme.

Step 8 – activating the plugin

Now copy the `wpwa-web-application` plugin into the `/wp-content/plugins` folder. Use the **Plugins** section on the admin menu to activate the plugin for this book.

Step 9 – using the application

Now we have completed the process of configuring WordPress for our portfolio management application. Open the web browser and enter the URL as `http://www.yoursite.com/user/register` or `http://localhost/application_folder/user/register` based on your URL structure to load the registration page of the application. You can use the menu and forms to navigate through the site and check all the features built throughout this book.

Open source libraries and plugins

We used a number of open source libraries and plugins throughout the book. The following list illustrates all the libraries and plugins used with the respective URLs to get more information:

- Responsive theme by CyberChimps: `http://goo.gl/Uf9Mp1`
- Members plugin by Justin Tadlock: `http://goo.gl/HuhDax`
- Rewrite Rules Inspector plugin by Daniel Bachhuber and Automattic: `http://goo.gl/oBVJmL`
- Twig template engine by SensioLabs: `http://goo.gl/geKkRY`
- Posts 2 Posts plugin by Alex Ciobica and Scribu: `http://goo.gl/8pQGmT`
- Pods – Custom Content Types and Fields: `http://goo.gl/ixMspf`
- Options-Framework by Syamil MJ: `http://goo.gl/GhBL4s`
- Custom List Table Example plugin by Matt Van Andel: `http://goo.gl/3tnfmf`
- Backbone.js library: `http://goo.gl/VyhEDl`
- Underscore.js library: `http://goo.gl/aZ42YD`
- PHPMailer library: `http://goo.gl/VX90ym`
- Opauth library: `http://goo.gl/SMeFJb`

Online resources and tutorials

The following is a list of online resources and tutorials that you can refer to:

- But Seriously, WordPress as an Application Platform? by Tom McFarlin: `http://goo.gl/gONP3i`
- WordPress For Application Development by Tom McFarlin: `http://goo.gl/ubDasf`
- My Thoughts on Building Web Applications with WordPress by Tom McFarlin: `http://goo.gl/fTUqQf`
- Why WordPress Isn't Viewed as an Application Framework by Tom McFarlin: `http://goo.gl/Ophmak`
- Using WordPress as a Web Application Framework by Harish Chouhan: `http://goo.gl/BFHqVB`
- Picklist – Build Powerful Websites and Web Applications with WordPress: `http://goo.gl/WYqRNh`
- Build an App With WordPress – The compulsory todo list by Harley Alexander: `http://goo.gl/rwMB6c`

Index

Symbols

$request_data variable 251

A

action hooks
planning, for layouts 232, 233
used, for extending home page template 229
add_action function 32
add_option function 86, 178
add_rewrite_rule function 48
add_theme_page function 175
admin role 41
admin dashboard
about 14, 164
admin toolbar, customizing 164, 165
admin toolbar items, managing 166-168
admin toolbar, removing 165
appearance 15
pages 14
posts 14
responsive design 195, 196
settings 15
users 15
visual presentation 190
admin list table
available custom columns, listing 185
bulk actions list, creating 186
checkbox, displaying 184, 185
column default handlers, implementing 184
custom class, defining 182
custom column handlers, implementing 183, 184
custom list, adding as menu page 187
generated list, displaying 187-190
initial configurations, creating 182, 183
instance variables, defining 182
list data, retrieving 186
sortable columns, defining 185
using 182
Advanced Content Type component
enabling 132
AJAX
using, in WordPress 138
AJAX-based filtering
about 316
enabling, in restructured application 316-319
ajax_developer_list function 317
ajaxInitializer function 146, 317
AJAX plugin
planning 140
AJAX requests
creating, jQuery used 138, 139
defining 139
drawbacks 140
Amazon Product Advertising API
URL 280
API access tokens
integrating 289-293
API client
building 281-284
API documentation
providing 293
api_request function 284
APIs
about 280
advantages 280
api_token parameter 294

API user authentication
integrating 287, 288
application data tables 82
application folder
creating 337
application layouts
widgetizing 212
application layout, WordPress theme
customizing 13
application options panel
building 176, 178
application URL
configuring 338
application users
registering 45
Archive Page 203
authentication parameter 294
author 41
auto saving 91

B

Backbone.js
about 240
code structuring 241, 242
developer profile page, creating 243-246
integrating, with Underscore.js 242
Backbone.js library
URL 344
Better WP Security plugin
URL 333
built-in insert functions
add_option 86
wp_insert_comment 86
wp_insert_post 86
built-in update functions
update_post_meta 86
update_user_meta 86
wp_update_term 86
BulletProof Security plugin
URL 334

C

caching 11, 330
CMS 8
comment-related tables
wp_commentmeta 76
wp_comments 76
wp_links 77
wp_options 77
comments_popup_link function 34
components, WordPress
admin dashboard 14
identifying 12
plugins 15
widgets 16
WordPress theme 12
Content Management System. *See* **CMS**
contributor 41
create_custom_tables function 83
create_developer_profile function 243, 245, 324
custom API
about 285
creating 285, 286
custom content types
about 94
Pods framework 128
custom e-mail sending functionality, with PHPMailer
creating 260
custom functions, creating 261-263
pluggable wp_mail function custom version, creating 260
custom field data
persisting 119-122
custom fields
enabling 110
Custom List Table Example plugin
URL 344
custom menu pages 171
custom post manager
restructuring 312
Custom Posts Manager plugin 149
custom post types
about 94
custom fields, enabling 110-112
custom taxonomies, creating for technologies 105-108
implementing, for portfolio application 97-100
messages, customizing 123, 124
planning 94
projects class, creating 102, 104

relationships 126
requisites, for portfolio application 94, 95
settings, implementing 100, 101
using, in web applications 94
custom post types, portfolio application
articles 97
books 97
projects 95
services 96
custom tables
creating 83, 84
used, for extending database 81
custom taxonomies
creating 105-108
permissions, adding to project type 108, 110
custom template implementation 47
custom template loader
creating 217-219
custom templates
creating 54
custom templates, with custom routing
direct template inclusion 207, 208
pure PHP templates, using 206
template engines 209, 210
templates, reusing 207
theme, versus plugin templates 208

D

database management 11
data selecting functions
get_option 86
get_posts 86
get_users 86
dbDelta function 83
default capabilities 44
default user roles
admin 41
author 41
contributor 41
editor 41
subscriber 41
superadmin 41
delete_option function 178
developer profile page, Backbone.js used
creating 243
events, integrating to Backbone.js views 254
models, creating in server 256-259
projects, creating from frontend 253
projects list, displaying on page load 249-252
structuring, with Backbone.js and Underscore.js 247, 249
validate function, adding 255
developers_list function 286
developer_subscriptions function 287, 292
do_action hook
about 52
advantages 52, 53
dynamic_sidebar function 220

E

editor 41
edit_user_profile_update function 322
execute_activation_hooks function 306
existing tables, querying
records, deleting 86
records, inserting 86
records, selecting 86
records, updating 86
extendable templates
about 226
using, in web applications 226, 227
extensible plugins
about 148
extensible file uploader plugin, creating 149, 150
file fields, converting with jQuery 151
file uploader, planning for portfolio application 148
file uploader plugin, extending 155
media uploader, integrating to buttons 152, 154
project screens, loading 158, 159
project screens, saving 157

F

Facebook Graph API
URL 280
feature-packed admin list tables
extended lists, building 181

using 180-182
file management 10
file uploader plugin
creating 149
extending 155
images, customizing 155, 156
flush_application_rewrite_rules function 50
front_controller function 314
frontend login
creating 63, 65
login form, displaying 65-68
frontend menu, web application
generating 221
navigation menu, creating 222, 223
user-specific menus, displaying 224, 225
frontend registration implementation
custom template implementation 47
custom templates, creating 54
functions access, controlling 51, 52
page template implementation 47
performing 46
registration form, designing 54, 55
registration form submission, handling 56-58, 61
registration process, planning 56
router, building for user modules 47
shortcode implementation 46
system users, activating 61-63
frontend, WordPress application
template execution hierarchy 201
theme file structure 200
functions, portfolio management application
developer profile management 20
frontend login and registration 20
notification service 20
responsive design 21
settings panel 20
third-party libraries 21
XML API 20
functions, WordPress Options API
add_option 178
delete_option 178
get_option 179
update_option 179

G

generate_project_messages function 123
generate_random_hash function 291
get_comment_meta function 27
getData function 250
getDevelopers function 286
get_footer function 220
get_header function 220
get_nodes function 166
get_option function 86, 179
get_posts function 86
get_template_part function 208
get_transient function 124
get_user_meta function 324
get_users function 86
Google Maps API
URL 280

H

home page template
designing 219, 220
extending, action hooks used 229
widgets, customizing 230, 232

I

initialize_app_controllers function 310
initialize_templates function 115
Initial Request 202
is_home function 220

L

LinkedIn app
building 266
list_developers function 315
list_projects function 251
load_opauth function 271
login_user function 65

M

main navigation menu
customizing 169, 170

menu items, creating 171
manage_routing_rules function 314
manage_user_routes function 50
master tables 82
Members plugin
URL 344
metatables
about 92
using 92
MVC architecture
versus, event-driven architecture 9, 10

O

Opauth
about 264
URL 264, 344
using 264
Open Close Principle 136
open source JavaScript libraries, WordPress core
about 239
Backbone.js 240
Backbone.js code structuring 241
Backbone.js, integrating with Underscore.js 242
developer profile page, creating with Backbone.js 243-247
open source libraries
advantages 238
in WordPress core 238
option pages
about 171
application options panel, building 176, 178
automating, with SMOF 173, 174
building 172
customizing 175
Options-Framework
URL 344

P

page layout, WordPress theme
footer 13
header 13
main content 13
structure 13
page template
about 47
cons 47
pros 47
permalinks
about 342
setting up 342
personal_options_update function 322
PHPMailer
about 259
used, for custom e-mail sending 259, 260
URL 344
used, within WordPress core 260
phpMyAdmin database browser 136
pluggable plugins
about 159
creating 159-161
pluggable templates 226
pluggable templates, WordPress
about 227
using 228
plugin_dir_path function 101
plugins
about 11, 15
using 276
Pods framework
about 128
admin menu 129
Advanced Content Type component, enabling 132
advantages 132
configurations 131
content type creation screen 130
custom field, creating 130
features 128
Product creation screen 131
URL 129, 131
portfolio application, structuring
activation controller, building 306
admin menu controller, building 309
application controllers, initializing 310, 311
autoloader, reusing 303
class initializations, creating 310
common folders, creating 300, 301
components, loading to main plugin 302
main plugin functions, defining 303

plugin definitions, removing 300
plugins, deactivating 298
plugins, moving into wpwa-web-application 299
script controller, building 307, 308
standalone plugin, creating 299
template loader, creating 302
template router, building 304, 305
portfolio application tables
application data tables 82
custom tables, creating 83, 84
master tables 82
planning 82
transaction tables 82
portfolio management application
development plan 18
functions 20, 21
goals 18
home page, building 210
integrating 298
planning 19
structuring 298
target audience 18
user roles 19
widget 211
post-related tables
about 74
wp_postmeta 75
wp_posts 74
post revisions
about 90
disabling 91
enabling 91
Posts 2 Posts plugin
about 126, 127
p2p_register_connection_type function 127
URL 344
process_projects function 250
projects class
creating 102
permissions, adding to projects 104, 105

Q

question-answer interface
answer status, changing 25-30
answer status, saving 30, 31
building 22
prerequisites 23
question list, generating 33-35
questions, creating 23, 25

R

redirect_templates function 305
register_activation_hook function 306
register_setting function 179
register_user function 57, 61
register_widget function 213
registration form
designing 54, 55
registration form submission
handling 56-60
registration process
planning 56
remove_menu function 167
render function 125
requisites, web application registration process
detailed information, requesting 45
user accounts, activating 45
user-friendly interface 45
Responsive theme
activating 343
downloading 343
URL 344
RESTful architecture
URL 240
RESTful JSON interface 240
restructured application
AJAX-based filtering, enabling 316-319
developer list template, designing 315, 316
developer model, building 314
working with 313, 314
result parameter 294
reusable AJAX requests
creating 146
reusable libraries
AJAX plugin, planning 140
AJAX request, creating using jQuery 138, 139
AJAX requests, defining 139
AJAX, using in WordPress 138
creating 138

drwabacks, AJAX requests 140
plugin, creating 141-143
plugin scripts, including for AJAX 143, 145
reusable AJAX requests, creating 146, 147
Rewrite Rules Inspector plugin
URL 344
router
building 48
query variables, adding 49
rewriting rules, flushing 49, 50
rules, creating 48
routing 11

S

save_post action 119
save_profile_fields function 323
scheduling 11
Secure WordPress plugin
URL 334
security
about 333
guidelines 333
SensioLabs 113
set_frontend_toolbar function 166
set_transient function 122
shortcodes
about 46
cons 46
pros 46
show_admin_bar function 166
Single Post Page
Attachment Post 203
Blog Post 203
Custom Post 203
Singular Page
about 202
Custom page 203
Default page 203
Single Post Page 203
Static Page 203
SMOF (Slightly Modded Options Framework) 173
spl_autoload_register function 99
subscriber 41
subscriber notifications
e-mail, sending 327, 329
scheduling 325, 326
superadmin 41
system users
activating 61, 62

T

template engine
about 113
Twig template, creating 116-118
Twig templates, configuring 114, 115
template execution hierarchy
about 201
Initial Request 202
Singular Page 202
template execution process 203, 204
template loader
integrating, into user manager 312, 313
template management 11
Template_Router class 307
templates
creating 225
extendable templates 226
pluggable templates 226
term-related tables
about 75
wp_term_relationships 75
wp_terms 75
wp_term_taxonomy 75
testing 332, 333
third-party libraries
using 276
transaction support 90
transaction tables 82
transients 332
twentytwelve_entry_meta function 33
Twig template
configuring 114, 115
creating 116, 117
data, parsing to 124, 125
URL 344
Twitter REST API
URL 280

U

Underscore.js library
URL 344

update_comment_meta function 32
update_option function 179
update_post_meta function 86, 122
update_user_meta function 86, 291, 323
user_api_settings function 290
user authentication implementation, Opauth used
LinkedIn app, building 266-269
login strategies, configuring 265, 266
Opauth library, initializing 270-272
performing 263-265
strategies, requesting 270
users, authenticating 273-275
user capabilities
about 43
creating 43
default capabilities 44, 45
user management
about 38
application users, registering 45
frontend login, creating 63
frontend registration, implementing 46
plugin, preparing 38, 39
user capabilities 43
user roles 39
user module 10
user profile
field values, updating 322-324
updating, with additional fields 320, 322
user-related tables
about 73
wp_usermeta 73
wp_users 73
user roles
about 39
adding 40
creating 40
default roles 41
removing 42, 43
selecting 42
user roles, portfolio management application
admin 19
developer 19
members 20
subscribers 20

V

visual presentation, admin dashboard 190-194

W

W3 Total Cache plugin
URL 331
ways of adapting, tables into web application
common-related tables 80
post-related tables 78
post-related tables, scenarios 79
term-related tables 79
term-related tables, scenarios 79
user-related tables 78
web application frameworks
frontend menu, generating 221
template execution process 203, 204
web application layout creation techniques
about 205
custom templates, with custom routing 206
shortcodes and page templates 205, 206
web applications
WordPress XML-RPC API 281
widgets
about 11, 16, 211
creating 213-217
sidebar 16
using 17
widget template
modifying, with extendable hooks 230-232
WordPress
about 7
application, using 344
as CMS 8
as web development framework 9
components, identifying 12
custom content types 94
guidelines 21, 22
installing 339-342
limitations 21, 22
online resources 345
open source libraries 344
permalinks, setting up 342

plugins 344
plugins, activating 343
question-answer interface, building 22
reference links, for tutorials 345
Responsive theme, activating 343
Responsive theme, downloading 343
WordPress API
components 281
WordPress application
frontend 200
WordPress configuration
3.6 version, downloading 337
application folder, creating 337
application URL, configuring 338
application URL, configuring using localhost 338
performing 337
virtual host, creating 338
WordPress core
open source JavaScript libraries 239
open source libraries 238
WordPress core modules
caching 11
database management module 11
file management module 10
plugins 11
routing 11
scheduling 11
template management module 11
user module 10
widgets 11
XMR-RPC API 11
WordPress database
about 72
common-related tables 76
custom tables, querying 87
existing tables, querying 85
extending, with custom tables 81
features 90
limitationss 90
portfolio application tables, planning 82
post-related tables 74
posts, working with 88
querying 85
roles, exploring 72
term-related tables 75
user-related tables 73
WP_Query, extending 88, 89
WordPress features
caching 330
security 333
testing 332
transients 332
WordPress Options API
about 178
functions 178
using 178-180
WordPress plugins
about 136
architecture 136, 137
WordPress plugins, for web development
about 137
extensible plugins 148
pluggable plugins 159
reusable libraries, creating with plugins 138
WordPress posts
working with 88
WordPress-specific features
auto saving 91
metatables 92
post revisions 90
transaction support 90
WordPress templates
comparing, with Twig templates 229
WordPress theme
about 12
application layout, customizing 13, 14
page layout structure 13
wp_cache_add function 331
wp_cache_get function 331
wpdb class 87
wp_enqueue_media function 151
wp_enqueue_script function 30
wp_generate_password function 60
wp_get_current_user function 27
wp_insert_comment function 86
wp_insert_post function 86
wp_insert_user function 60
wp_list_comments function 26
wp_localize_script function 30, 146, 147, 247
wp_mail_charset hook 260
wp_mail_content_type hook 260
wp_mail_from hook 260
wp_mail_from_name hook 260

wp_mail function 260
wp.media.editor.open function 153
wp.media.editor.send.attachment function 153
wp_new_user_notification function 60
WP_Query
extending, for applications 88, 89
wp_redirect function 275
wp_register_script function 30
wp_schedule_event function 325
WP Security Scan plugin
URL 334
wp_set_auth_cookie function 276
wp_signon function 67
WP Super Cache plugin
URL 331
wp_update_term function 86
wp_verify_nonce function 119
wpwa_comment_list function 26
wpwa_customize_admin_toolbar function 166, 167
wpwa_get_correct_answers function 34
wpwa_mark_answer_status function
implementing 32
WPWA_Services_Query class 89
wpwa-user-manager plugin 264
wpwa_xml_rpc_api function 285

X

XML-RPC API 281
xml_rpc_api function 287, 293
xmlrpc_decode function 283
xmlrpc_encode_request function 283
XMR-RPC API 11

Y

Youtube API
URL 280

Thank you for buying

WordPress Web Application Development

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

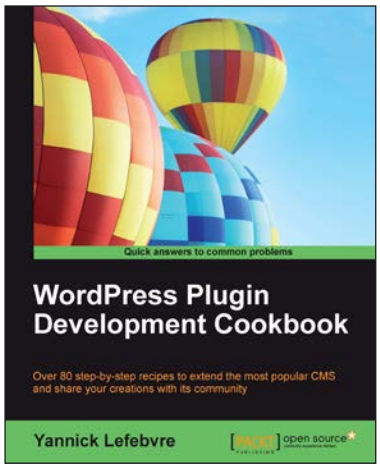
WordPress 3 Complete

ISBN: 978-1-84951-410-1 Paperback: 344 pages

Create your own complete website or blog from scratch with WordPress

1. Learn everything you need for creating your own feature-rich website or blog from scratch
2. Clear and practical explanations of all aspects of WordPress
3. In-depth coverage of installation, themes, plugins, and syndication
4. Explore WordPress as a fully functional content management system
5. Clear, easy-to-follow, concise; rich with examples and screenshots

---



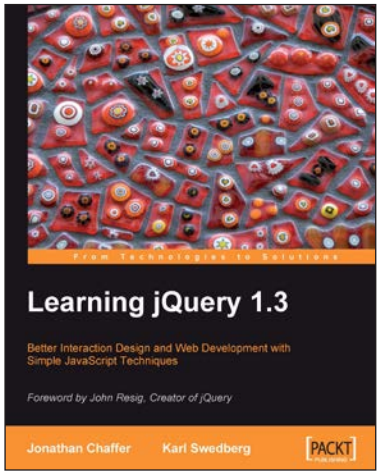
WordPress Plugin Development Cookbook

ISBN: 978-1-84951-768-3 Paperback: 318 pages

Over 80 step-by-step recipes to extend the most popular CMS and share your creations with its community

1. Learn to create plugins and configuration panels in order to bring new capabilities to WordPress Tailor WordPress to your needs with new content types, custom widgets, and fancy jQuery elements, without breaching security needs Detailed instructions on how to achieve each task, followed by clear explanations of concepts featured in each recipe

[PACKT] open source
community experience distilled
PUBLISHING

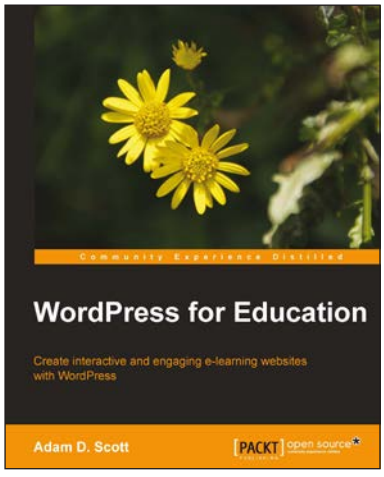


Learning jQuery 1.3

ISBN: 978-1-84719-670-5 Paperback: 444 pages

Better Interaction Design and Web Development with Simple JavaScript Techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. For web designers to create interactive elements for their designs
4. For developers to create the best user interface for their web applications
5. Packed with great examples, code, and clear explanations



WordPress for Education

ISBN: 978-1-84951-820-8 Paperback: 144 pages

Create interactive and engaging e-learning websites with WordPress

1. Develop effective e-learning websites that will engage your students
2. Extend the potential of a classroom website with WordPress plugins
3. Create an interactive social network and course management system to enhance student and instructor communication