



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# WordPress Web Application Development

## *Second Edition*

Build rapid web applications with cutting-edge technologies using WordPress

Rakhitha Nimesh Ratnayake

**[PACKT]** open source\*  
PUBLISHING community experience distilled

# WordPress Web Application Development

## *Second Edition*

Build rapid web applications with cutting-edge technologies using WordPress

**Rakhitha Nimesh Ratnayake**



BIRMINGHAM - MUMBAI

# WordPress Web Application Development

*Second Edition*

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2013

Second edition: May 2015

Production reference: 1250515

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

ISBN 978-1-78217-439-4

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Author**

Rakhitha Nimesh Ratnayake

**Project Coordinator**

Mary Alex

**Reviewers**

Alex Bachuk

Baljeet Singh

Doug Sparling

**Proofreaders**

Stephen Copestake

Safis Editing

**Indexer**

Hemangini Bari

**Commissioning Editor**

Deepika Gaonkar

**Production Coordinator**

Komal Ramchandani

**Acquisition Editor**

Reshma Raman

**Cover Work**

Komal Ramchandani

**Content Development Editor**

Rohit Singh

**Technical Editor**

Dhiraj Chandanshive

**Copy Editor**

Janbal Dharmaraj

# About the Author

**Rakhitha Nimesh Ratnayake** is a freelance web developer, writer, and an open source enthusiast. He develops premium WordPress plugins for individual clients and the CodeCanyon marketplace. Rakhitha is the creator of [www.innovativephp.com](http://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, which was published by Packt Publishing. In his spare time, he likes to watch cricket and spend time with his family. You can visit him online at [www.innovativephp.com](http://www.innovativephp.com) and follow him on ODesk at <http://goo.gl/ykDLnk>.

# About the Reviewers

**Alex Bachuk** is a web developer with over 7 years of experience, specializing in custom JavaScript and WordPress web applications. Alex has been working with WordPress since Version 2.5 and has worked on projects ranging from a single-page website to interactive web applications and social platforms.

These days, Alex mostly works on single-page web applications powered by Angular.js and full stack Javascript applications using Meteor. His current projects include <http://classmate.io>, a web application for education, and [www.timebooklet.com](http://www.timebooklet.com), a timesheet-focused reporting application.

Alex also organizes and talks at WordPress meetups throughout New England. He writes about technology on his blog, [www.alexbachuk.com](http://www.alexbachuk.com).

In his free time, Alex likes to travel the world with his wife Oksana, and when there is even more time, he practices Judo and Brazilian Jiu Jitsu.

**Baljeet Singh** is a web programmer, mobile application developer, consultant, and trainer. He is the creator of Cinnabar WordPress Framework (a WordPress theme framework based on Bootstrap 3). A github repo link for this is available at <http://goo.gl/7z2Zom>. He is very passionate about web technologies. In his free time, he likes to write about WordPress and various emerging technologies at <http://baljeetsingh.in/blog/>.

His objective is to make a positive impact on clients, co-workers, and the Internet, using his skills and experience to design and develop compelling and attractive websites, web applications, and mobile applications. He enjoys working on projects that involve a mix of web design, web development, and mobile application development.

**Doug Sparling** works as a technical architect and software developer for Andrews McMeel Universal, a publishing and syndication company in Kansas City, MO. At AMU, he uses Go for web services and backend processing, Ruby on Rails for web development, and Objective-C, Swift, and Java for iOS and Android development. The sites include [www.gocomics.com](http://www.gocomics.com), [www.uexpress.com](http://www.uexpress.com), [www.puzzlesociety.com](http://www.puzzlesociety.com), and [www.dilbert.com](http://www.dilbert.com).

He is also the director of technology for a small web development firm called New Age Graphics ([www.newage-graphics.com](http://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.

He is a passionate advocate for WordPress and has written several WordPress plugins. He can occasionally be found on the WordPress (<https://wordpress.org>) forums answering questions (and writing sample code) under the username *scriptrunner*.

He was also the co-author of a Perl book, *Instant Perl Modules*, for McGraw-Hill and is a reviewer for other Packt Publishing books, including *jQuery 2.0 Animation Techniques: Beginner's Guide*, and its first edition. He was also a reviewer for *The Well-Grounded Rubyist, Second Edition*, Manning Publications, and a technical proofer for *Programming for Musicians and Digital Artists*.

He is currently the technical directing editor for *Programming in Haskell* and reviewer for *Go In Action*, among others.



# www.PacktPub.com

## Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

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](http://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](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a web browser

## Free access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.





# Table of Contents

<b>Preface</b>	<b>xi</b>
<b>Chapter 1: WordPress as a Web Application Framework</b>	<b>1</b>
<b>WordPress as a CMS</b>	<b>2</b>
<b>WordPress as a web application framework</b>	<b>3</b>
The MVC versus event-driven architecture	4
<b>Simplifying development with built-in features</b>	<b>4</b>
User management	5
Media management	5
Template management	5
Database management	5
Routing	5
XMR-RPC API	6
Caching	6
Scheduling	6
Plugins and widgets	6
Themes	6
Actions and filters	7
The admin dashboard	7
<b>Identifying the components of WordPress</b>	<b>7</b>
The role of WordPress themes	8
Structure of a WordPress page layout	8
Customizing the application layout	9
The role of the admin dashboard	9
The admin dashboard	9
Posts and pages	10
Users	10
Appearance	10
Settings	10
The role of plugins	10

The role of widgets	11
<b>A development plan for the portfolio management application</b>	<b>13</b>
Application goals and target audience	13
Planning the application	14
User roles of the application	15
Planning application features and functions	15
<b>Understanding limitations and sticking to guidelines</b>	<b>17</b>
<b>Building a question-answer interface</b>	<b>18</b>
Prerequisites for building a question-answer interface	18
Creating questions	19
Customizing the comments template	22
Changing the status of answers	23
Saving the status of answers	28
Generating a question list	30
<b>Enhancing features of the questions plugin</b>	<b>32</b>
Customizing the design of questions	32
Categorizing questions	33
Approving and rejecting questions	33
Adding star rating to answers	33
<b>Summary</b>	<b>34</b>
<b>Chapter 2: Implementing Membership Roles, Permissions, and Features</b>	<b>35</b>
<b>Introduction to user management</b>	<b>36</b>
Preparing the plugin	36
<b>Getting started with user roles</b>	<b>37</b>
Creating application user roles	38
The best action for adding user roles	38
Knowing the default roles of Wordpress	40
Choosing among default and custom roles	40
Removing existing user roles	41
<b>Understanding user capabilities</b>	<b>41</b>
Creating your first capability	42
Understanding default capabilities	42
<b>Registering application users</b>	<b>43</b>
<b>Implementing frontend registration</b>	<b>44</b>
Shortcode implementation	44
Pros and cons of using shortcodes	45
Page template implementation	45
Pros and cons of page templates	45
Custom template implementation	46
Building a simple router for a user module	46

Creating the routing rules	47
Adding query variables	47
Flushing the rewriting rules	48
Controlling access to your functions	50
The advantages of using the do_action function	51
Creating custom templates	52
Designing the registration form	52
Planning the registration process	54
Handling registration form submission	55
Exploring the registration success path	58
Automatically log in the user after registration	60
Activating system users	61
<b>Creating a login form in the frontend</b>	<b>62</b>
Displaying the login form	64
Checking whether we implemented the process properly	66
<b>Time to practice</b>	<b>67</b>
<b>Summary</b>	<b>68</b>
<b>Chapter 3: Planning and Customizing the Core Database</b>	<b>69</b>
<b>Understanding the WordPress database</b>	<b>70</b>
<b>Exploring the role of existing tables</b>	<b>70</b>
User-related tables	71
Post-related tables	72
Term-related tables	73
Other tables	74
<b>Adapting existing tables into web applications</b>	<b>75</b>
User-related tables	76
Post-related tables	76
Scenario 1 – An online shopping cart	77
Scenario 2 – A hotel reservation system	77
Scenario 3 – The project management application	77
Term-related tables	77
Other tables	78
<b>Extending the database with custom tables</b>	<b>79</b>
<b>Planning the portfolio application tables</b>	<b>80</b>
Types of tables in web applications	80
Creating custom tables	81
<b>Querying the database</b>	<b>84</b>
Querying the existing tables	84
Inserting records	84
Updating records	84
Deleting records	85
Selecting records	85
Querying the custom tables	85

Working with posts	86
Extending the WP_Query class for applications	87
Introduction to WordPress query classes	88
The WP_User_Query class	89
The WP_Comment_Query class	89
Other query classes	90
<b>Limitations and considerations</b>	<b>90</b>
Transaction support	91
Post revisions	91
How to know whether to enable or disable revisions?	91
Auto saving	92
Using meta tables	92
<b>Summary</b>	<b>92</b>
<b>Chapter 4: Building Blocks of Web Applications</b>	<b>95</b>
<b>Introduction to custom content types</b>	<b>96</b>
The role of custom post types in web applications	96
<b>Planning custom post types for application</b>	<b>97</b>
Projects	97
Services	98
Articles	99
Books	99
<b>Implementing custom post types for a portfolio application</b>	<b>100</b>
Implementing the custom post type settings	103
Creating the project class	104
Assigning permissions to projects	107
Creating custom taxonomies for technologies and project types	108
Assigning permissions to the project type	111
Introduction to custom fields with meta boxes	113
<b>What is a template engine?</b>	<b>115</b>
Building a simple custom template loader	116
Creating your first template	118
Comparing the template loader and template engine	122
<b>Persisting custom field data</b>	<b>123</b>
Customizing custom post type messages	127
<b>Introducing custom post type relationships</b>	<b>129</b>
<b>Pods framework for custom content types</b>	<b>132</b>
Should you choose Pods for web development?	135
<b>Time to practice</b>	<b>136</b>
<b>Summary</b>	<b>137</b>

---

<b>Chapter 5: Developing Pluggable Modules</b>	<b>139</b>
<b>A brief introduction to WordPress plugins</b>	<b>140</b>
Understanding the WordPress plugin architecture	140
<b>WordPress plugins for web development</b>	<b>141</b>
Creating reusable libraries with plugins	142
Planning the template loader plugin	142
Using the template loader plugin	144
Handling plugin dependencies	145
Extensible plugins	148
Extend plugins with WordPress core actions and filters	149
Extend plugins with custom actions and filters	159
Pluggable plugins	161
Tips for using pluggable functions	164
<b>Time to practice</b>	<b>165</b>
<b>Summary</b>	<b>165</b>
<b>Chapter 6: Customizing the Dashboard for Powerful Backends</b>	<b>167</b>
<b>Understanding the admin dashboard</b>	<b>168</b>
<b>Customizing the admin toolbar</b>	<b>168</b>
Removing the admin toolbar	169
Managing the admin toolbar items	170
<b>Customizing the main navigation menu</b>	<b>173</b>
Creating new menu items	174
<b>Adding features with custom pages</b>	<b>175</b>
<b>Building options pages</b>	<b>175</b>
Creating a custom layout for options pages	176
Building an application options panel	178
Using the WordPress options API	181
<b>Using feature-packed admin list tables</b>	<b>183</b>
Working with default admin list tables	183
The post list	184
The user list	191
The comments list	192
Building extended lists	193
Using the admin list table for the following developers	194
Step 1 – defining the custom class	194
Step 2 – defining the instance variables	194
Step 3 – creating the initial configurations	195
Step 4 – implementing the custom column handlers	195
Step 5 – implementing the column default handlers	196
Step 6 – displaying the checkbox for records	197
Step 7 – listing the available custom columns	197
Step 8 – defining the sortable columns of list	198

Step 9 – creating a list of bulk actions	198
Step 10 – retrieving list data	198
Step 11 – adding a custom list as a menu page	199
Step 12 – displaying the generated list	199
<b>An awesome visual presentation for admin screens</b>	<b>202</b>
Using existing themes	203
Using plugin-based third-party admin themes	203
Creating your own admin theme	205
<b>The responsive nature of the admin dashboard</b>	<b>209</b>
<b>Time for action</b>	<b>210</b>
<b>Summary</b>	<b>211</b>
<b>Chapter 7: Adjusting Theme for Amazing Frontends</b>	<b>213</b>
<b>An introduction to the WordPress application frontend</b>	<b>214</b>
A basic file structure of the WordPress theme	214
Understanding the template execution hierarchy	215
The template execution process of web application frameworks	217
<b>Web application layout creation techniques</b>	<b>218</b>
Shortcodes and page templates	219
Custom templates with custom routing	219
Using pure PHP templates	220
The WordPress way of using templates	220
Direct template inclusion	221
Theme versus plugin-based templates	222
<b>Building the portfolio application home page</b>	<b>223</b>
What is a widget?	223
<b>Widgetizing application layouts</b>	<b>224</b>
Creating widgets	225
<b>Designing a home page template</b>	<b>231</b>
<b>Generating the application frontend menu</b>	<b>233</b>
Creating a navigation menu	233
Displaying user-specific menus on the frontend	236
<b>Managing options and widgets with customizer</b>	<b>237</b>
Adding custom options to the theme customizer	238
Handling widgets in the theme customizer	240
<b>Creating pluggable templates</b>	<b>242</b>
Pluggable templates in WordPress	242
<b>Extending the home page template with action hooks</b>	<b>244</b>
Customize widgets to enable extendable locations	245
<b>Planning action hooks for layouts</b>	<b>247</b>
<b>Time for action</b>	<b>249</b>
<b>Summary</b>	<b>249</b>



<b>Chapter 8: Enhancing the Power of Open Source Libraries and Plugins</b>	<b>251</b>
<b>Why choose open source libraries?</b>	<b>252</b>
<b>Open source libraries inside the WordPress core</b>	<b>252</b>
<b>Open source JavaScript libraries in the WordPress core</b>	<b>253</b>
What is Backbone.js?	254
Understanding the importance of code structuring	255
Integrating Backbone.js and Underscore.js	256
Creating a developer profile page with Backbone.js	257
Structuring with Backbone.js and Underscore.js	260
Displaying the projects list on page load	262
Creating new projects from the frontend	266
Integrating events to Backbone.js views	268
Validating and creating new models for the server	269
Creating new models in the server	270
<b>Using PHPMailer for custom e-mail sending</b>	<b>273</b>
Usage of PHPMailer within the WordPress core	273
Creating a custom version of a pluggable wp_mail function	274
Loading PHPMailer inside plugins and creating custom functions	274
<b>Implementing user authentication with OAuth</b>	<b>277</b>
Configuring login strategies	279
Implementing LinkedIn account authentication	281
Verifying LinkedIn account and generating response	283
Building a LinkedIn app	285
The process of requesting the strategies	287
Initializing the library	287
Authenticating users to our application	289
<b>Using third- party libraries and plugins</b>	<b>292</b>
<b>Time for action</b>	<b>293</b>
<b>Summary</b>	<b>293</b>
<b>Chapter 9: Listening to Third-party Applications</b>	<b>295</b>
<b>Introduction to APIs</b>	<b>296</b>
The advantages of having an API	296
<b>The WordPress XML-RPC API for web applications</b>	<b>297</b>
<b>Building the API client</b>	<b>298</b>
<b>Creating a custom API</b>	<b>302</b>
<b>Integrating API user authentication</b>	<b>303</b>
<b>Integrating API access tokens</b>	<b>306</b>
<b>Providing the API documentation</b>	<b>311</b>
<b>Time for action</b>	<b>312</b>
<b>Summary</b>	<b>313</b>

<b>Chapter 10: Integrating and Finalizing the Portfolio Management Application</b>	<b>315</b>
Integrating and structuring the portfolio application	316
Adding the template loader dependencies	317
Integrating the template loader into a user manager	318
Working with a restructured application	320
Building the developer model	321
Designing the developer list template	322
Enabling AJAX-based filtering	323
Updating a user profile with additional fields	326
Updating the values of the profile fields	328
Scheduling subscriber notifications	331
Notifying subscribers through e-mails	333
Time for action	336
Final thoughts	337
Summary	337
<b>Chapter 11: Supplementary Modules for Web Development</b>	<b>339</b>
<b>Internationalization</b>	<b>340</b>
Introduction to WordPress translation support	340
The translation functions in WordPress	341
Creating plugin translations	341
Creating the POT file using PoEdit	342
Loading language files	345
Changing the WordPress language	345
Working with media grid and image editor	346
Introduction to the post editor	347
Using the WordPress editor	348
Video embedding	349
Lesser-known WordPress features	349
Caching	350
Transients	351
Testing	351
Security	352
Introduction to multisite	353
Time for action	354
Summary	355

---

<b>Appendix: Configurations, Tools, and Resources</b>	<b>357</b>
<b>Configure and set up WordPress</b>	<b>357</b>
Step 1 – downloading WordPress	357
Step 2 – creating the application folder	358
Step 3 – configuring the application URL	358
Creating a virtual host	358
Using a localhost	359
Step 4 – installing WordPress	359
Step 5 – setting up permalinks	362
Step 6 – downloading the Responsive theme	363
Step 7 – activating the Responsive theme	363
Step 8 – activating the plugin	363
Step 9 – using the application	364
<b>Open source libraries and plugins</b>	<b>364</b>
<b>Online resources and tutorials</b>	<b>364</b>
<b>Index</b>	<b>367</b>

---



# 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 use this framework for advanced web development. 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 onto 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.js, jQuery, and server-side technologies and techniques, such as template engines and OAuth integration.

This book differentiates from the norm by creating a website that is dedicated to providing tutorials, articles, and source code 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. The identification of the 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 using WordPress query classes and coverage of the planning portfolio management application database are the highlights of this chapter.

*Chapter 4, 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 managing template loaders 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 with the use of WordPress action and filter hooks 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 with various different techniques are the highlights of this chapter.

*Chapter 7, Adjusting Theme for Amazing Frontends*, dives into the techniques of designing amazing layouts, thereby opening them up 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 into your web application and structuring the application on the client side are the highlights of this chapter.

*Chapter 9, Listening to Third-party Applications*, demonstrates how to use the 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 and refactoring the code developed throughout this book. Improving the consistency of the application's code and completing the features developed throughout the previous chapters are the highlights of this chapter.

*Chapter 11, Supplementary Modules for Web Development*, introduces you to the supplementary WordPress features, such as Internationalization, video embedding, media grid, and multisite. An introduction to important concepts in application development, such as caching, security, and testing are the highlight of this chapter.

*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 4.0 or higher.
- 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 or designers, who know how to create a basic CMS site and are looking for ways to learn the complex web application development in a reusable, maintainable, and modular way. Basic knowledge of WordPress themes and plugin development is expected, although this is not a must for experienced PHP developers to go through this book.

## Conventions

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

Code words in text 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:

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

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Once the **Publish** button is clicked, we validate the form and save the error messages as transients."

[  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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in 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 at [www.packtpub.com/authors](http://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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

## Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: [http://www.packtpub.com/sites/default/files/downloads/40820S\\_ColorImages.pdf](http://www.packtpub.com/sites/default/files/downloads/40820S_ColorImages.pdf).

## 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted 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](mailto: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

If you have a problem with any aspect of this book, you can contact us at [questions@packtpub.com](mailto:questions@packtpub.com), and we will do our best to address the problem.

# 1

## WordPress as a Web Application Framework

In recent years, WordPress has matured from the most popular blogging platform 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 are interested in using WordPress, there are discussions and arguments about 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 features, which can be easily adapted to web application development using slight modifications. Since you are already reading this book, you have to be someone who is really excited to see how WordPress fits into web application development. Throughout this book, we will learn how we can inject the best practices of web development into WordPress framework to build web applications in rapid process.

Basically, this book will be important for developers from two different perspectives. On one hand, beginner- to intermediate-level WordPress developers can get 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 will cover the following topics:

- WordPress as a CMS
- WordPress as a web application framework
- Simplifying development with built-in features

- Identifying the components of WordPress
- Making a development plan for portfolio management application
- Understanding limitations and sticking with guidelines
- Building a question-answer interface

In order to work with 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 to choose 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. Later, it continued to improve as a CMS and now has a reputation 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?

Recent versions of WordPress have included popular web development libraries such as Backbone.js and Underscore.js and developers are building different types of applications with WordPress. So, we can assume that it's moving in 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 ideal 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 features and the existence of over 20,000 open source plugins
- A super simple and easy-to-access administration interface
- A fast learning curve and comprehensive documentation for beginners
- A rapid development process involving themes and plugins
- An 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. However, 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, we'll 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 the future of WordPress for web development.

## **WordPress as a web application 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's a list of possible reasons why frameworks become a priority in web application development:

- Frameworks provide stable foundations for building custom functionalities
- Usually, stable frameworks have a large development community with an active support
- They have built-in features to address the common aspects of application development, such as routing, language support, form validation, user management, and more
- They have a large amount 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 becomes the framework of choice for most developers. However, we have to keep in mind that WordPress is an application where we built applications on top of existing features. On the other hand, traditional frameworks are foundations used for building applications such as WordPress. Now, let's take a look at how WordPress fits into the boots of web application framework.

## **The MVC versus event-driven architecture**

A vast majority of web development frameworks are built to work with MVC architecture, where an 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 why developers refuse to choose it as a development framework. Even though it is not MVC, we can create custom execution process to make it work like a MVC application. Also, we can find frameworks such as WP MVC, which can be used to take advantage of both WordPress's native functionality and a vast plugin library and all of the many advantages of an MVC framework. Unlike other frameworks, it won't have the full capabilities of MVC. However, unavailability of the MVC architecture doesn't mean that we cannot develop quality applications with WordPress. There are many other ways to separate concerns in WordPress applications.

WordPress on the other hand, relies on a procedural event-driven architecture with its action hooks and filters system. Once a 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](http://codex.wordpress.org/Plugin_API/Action_Reference).

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

## **Simplifying development with built-in features**

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

Let's get a brief introduction about the WordPress core features to see how it fits into web application development.



## **User management**

Built-in user management features are quite advanced in order to cater 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. We can separate users into multiple levels using roles and then use capabilities to define the permitted functionality for each user level. Most full stack frameworks don't have a built-in user management features, and hence, this can be considered as an advantage of using WordPress.

## **Media 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. A 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 template engine. However, it offers a wide range of capabilities in 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 its 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 permalinks. WordPress makes it simple to change the default 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 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 called persistent and nonpersistent cache. Nonpersistent caching is provided by WordPress cache object while persistent caching is provided through its Transient API. Caching techniques in WordPress is a simple compared to other frameworks, but it's powerful enough to cater to complex web applications.

## **Scheduling**

As developers, you might have worked with cron jobs for executing certain tasks at specified intervals. WordPress offers same scheduling functionality through built-in functions, similar to a cron job. However, WordPress cron execution is slightly different from normal cron jobs. In WordPress, cron won't be executed unless someone visits the site. Typically, it's used for scheduling future posts. However, it can be extended to cater complex scheduling functionality.

## **Plugins and widgets**

The power of WordPress comes from 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 further in this chapter.

## **Themes**

The design of a WordPress site comes through the theme. This site offers many built-in template files to cater to the default functionality. Themes can be easily extended for custom functionality. Also, the design of the site can be changed instantly by switching compatible theme.

## **Actions and filters**

Actions and filters are part of the WordPress hook system. Actions are events that occur during a request. We can use WordPress actions to execute certain functionalities after a specific event is completed. On the other hand, filters are functions that are used to filter, modify, and return the data. Flexibility is one of the key reasons for the higher popularity of WordPress, compared to other CMS. WordPress has its own way of extending functionality of custom features as well as core features through actions and filters. These actions and filters allow the developers to build advanced applications and plugins, which can be easily extended with minor code changes. As a WordPress developer, it's a must to know the perfect use of these actions and filters in order to build highly flexible systems.

## **The admin dashboard**

WordPress offers a fully featured backend for administrators as well as normal users. These interfaces can be easily customized to adapt to custom applications. All the application-related lists, settings, and data can be handled through the admin section.

The overall collection of 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 features 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 will 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 WordPress themes
- The role of admin dashboard
- The role of plugins
- The role of widgets

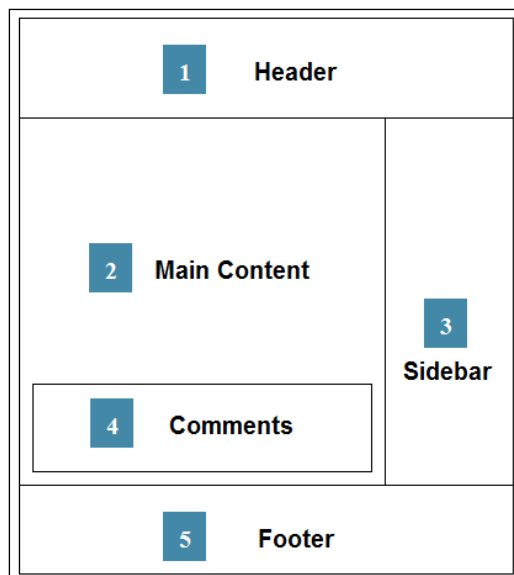
## The role of WordPress themes

Most 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. However, we can easily visualize themes 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 to create 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 views.

## 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 web application-specific structure. Let's visualize the common layout of WordPress using the following screen:



Having looked at the structure, it's obvious that header, footer, and the content area are mandatory even for web applications. However, the footer and comments section will play a less important role in web applications, compared to web pages. 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 targeting specific requirements of a client. On the other hand, a product is an application created based on the common set of requirements for wide range of users. Therefore, customizations will be required on layouts of your product based on different clients.

WordPress themes make it 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.

## **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 amongst other competitive frameworks. Most of you should be familiar with using admin area for CMS functionalities. However, we will have to understand how each component in the admin area suits the development of web applications.

## **The admin dashboard**

Dashboard is the location where all the users get redirected, once logged into admin area. Usually, it contains dynamic widget areas with the most important data of your application. Dashboard can play a major role in web applications, compared to blogging or CMS functionality. The dashboard contains a set of default widgets that are mainly focused on main WordPress features such as posts, pages, and comments. In web applications, we can remove the existing widgets related to CMS and add application-specific widgets to create a powerful dashboard. WordPress offers a well-defined API to create a custom admin dashboard widgets and hence we can create a very powerful dashboard using custom widgets for custom requirements in web applications.

## **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, Services, and so on.

## **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 the 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. Generally, widgets are used in sidebars of WordPress sites to display information such as recent members, comments, posts, and so on. However, in web applications, widgets can play a much bigger role as we can use widgets to split main template into multiple sections. Also, these types of widgetized areas become handy in applications where majority of features are implemented with AJAX.

The theme options panel can be used as the general settings panel of web applications where we define the settings related to templates and generic site-specific configurations.

## **Settings**

This section involves general application settings. Most of the prebuilt items in this section are suited for blogs and websites. We can customize this section to add new configuration areas related to our plugins, 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 to add new sections is one of the key reasons for its flexibility.

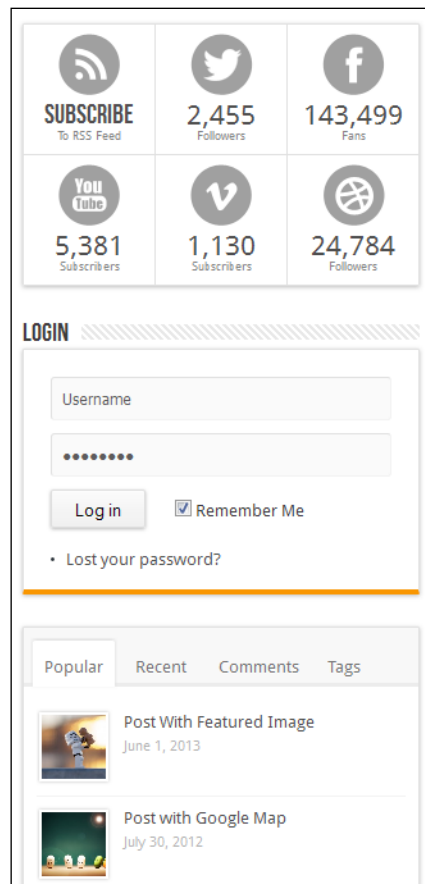
## **The role of plugins**

In normal circumstances, WordPress developers use functions that involve application logic scattered across theme files and plugins. Even some of the developers change the core files of WordPress, which is considered a very bad practice. In web applications, we need to be much more organized.

In the *Role of WordPress theme* section, 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. The plugins 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 this, plugins have the ability to extend other plugins.

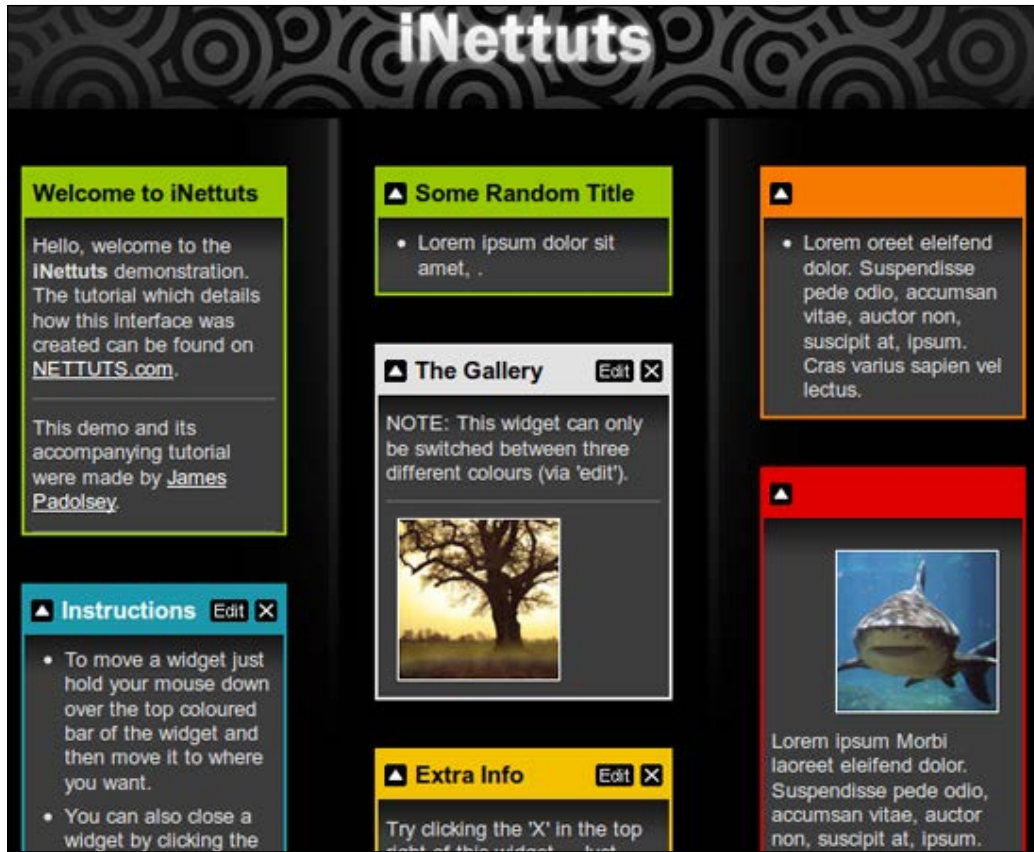
## The role of widgets

The official documentation of WordPress refers to widgets as a component that adds content and features to your sidebar. In a typical blogging or CMS user's perspective, it's a completely valid statement. Actually, the widgets offer more in web applications by going beyond the 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 source code. The following screenshot shows a sample dynamic widgetized area. We can use the same technique for developing 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 our application developed throughout this book.

## A development plan for the 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 will 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 about the goal of this application, features and functionalities, and the structure of components to see how it fits into WordPress.

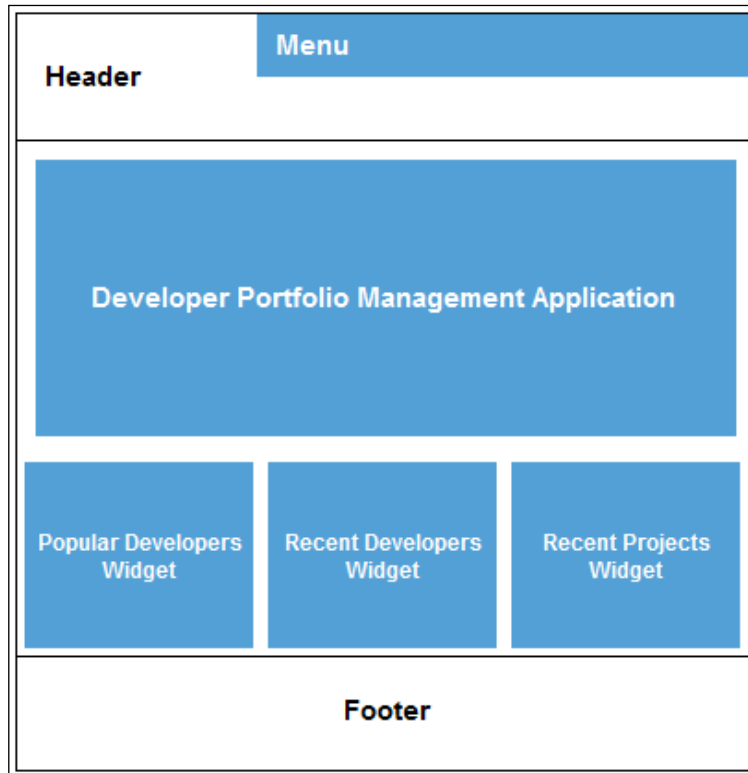
## Application goals and target audience

Developers and designers who work online as freelancers know the importance of a personal profile to show your skills for improved reputation. However, most people, including experts who work full-time jobs don't maintain such profiles, and hence get 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 will be targeted towards all the people who are involved in web development and design. I believe that both output of this application and the contents of the book will be ideal for the 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 diagram shows the structure of our application home page:



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

## User roles of the application

Application consists of four user roles, including the built-in admin role. User roles and their respective functionalities are explained in the following section:

- **Admin:** This manages the application configurations, settings, and 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 the things created by developers and designers. They will be able to access and download the work made public by developers. Basically, members will have more permission to directly interact with developers, compared to subscribers. Also, we can implement premium content section in future for paid members.
- **Subscribers:** These are also normal users who want to follow the activities of their preferred developers. These users will be notified whenever their preferred developers create a new activity within 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 profile by completing content divided into various sections such as services, portfolio, articles, and books.
- **Frontend login and registration:** Typically, web applications contain the login and registration in the frontend, whereas WordPress provides it on the admin area. Therefore, custom implementation of login and registration will be implemented in the application frontend.
- **Settings panel:** 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 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 mobile devices in Internet browsing, 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 the book, we will be creating functionalities such as OpenAuth 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 take a look at the various types of WordPress powered web applications at [http://www.innovativephp.com/demo/packt/wordpress\\_applications](http://www.innovativephp.com/demo/packt/wordpress_applications).

## Understanding limitations and sticking to guidelines

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

In this section, we will 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 suit 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 can be a limitation in WordPress as it's built in to work with 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 will 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 the 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.
- **Architecture:** 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 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 to update 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 now moving rapidly towards object-oriented architecture. So, there will be a mix of both procedural and object-oriented code. WordPress also uses a hook-based architecture to provide 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 the 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, can 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 practical approach to learn new things over the more conventional theoretical approach.

So, I will complete this chapter by converting default WordPress functionality into a simple question-answer interface such as Stack Overflow, to give you a glimpse of what we will develop throughout this book.

## Prerequisites for building a question-answer interface

We will be using version 4.2.2 as the latest stable version; this is available at the time of writing this book. I suggest that you set up a fresh WordPress installation for this book, if you haven't already done so.



Also, we will be using the Twenty Fourteen theme, which is available with default WordPress installation. Make sure that you activate the Twenty Fourteen theme in your WordPress installation.

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

1. Create questions using the admin section of WordPress.
2. Allow users to answer questions using comments.
3. Allow question creators to mark each answer as correct or incorrect.
4. Highlight the correct answers of each question.
5. Customize the question list to include a 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 off, 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, a post can be considered as the question and 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 plugin, which you can find at [http://codex.wordpress.org/Post\\_Types#Custom\\_Post\\_Types](http://codex.wordpress.org/Post_Types#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 will create a plugin to implement the necessary tasks for our application:

1. First off, create a folder called `wpwa-questions` inside the `/wp-content/plugins` folder and add a new file called `wpwa-questions.php`.
2. Next, we need to add the block comment to define our file as a plugin:

```
/*
Plugin Name: WPWA 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
Text Domain: wpwa-questions
*/
```

3. Having created the main plugin file, we can move into creating a custom post type called wpwa-question using the following code snippet.
4. Include this code snippet in your wpwa-questions.php file of the plugin:

```
add_action('init', 'register_wp_questions');
function register_wp_questions() {
    $labels = array(
        'name' => __( 'Questions', 'wpwa_questions' ),
        'singular_name' => __( 'Question',
'wpwa_questions' ),
        'add_new' => __( 'Add New', 'wpwa_questions' ),
        'add_new_item' => __( 'Add New Question',
'wpwa_questions' ),
        'edit_item' => __( 'Edit Questions',
'wpwa_questions' ),
        'new_item' => __( 'New Question',
'wpwa_questions' ),
        'view_item' => __( 'View Question',
'wpwa_questions' ),
        'search_items' => __( 'Search Questions',
'wpwa_questions' ),
        'not_found' => __( 'No Questions found',
'wpwa_questions' ),
        'not_found_in_trash' => __( 'No Questions found in
Trash', 'wpwa_questions' ),
        'parent_item_colon' => __( 'Parent Question:',
'wpwa_questions' ),
        'menu_name' => __( 'Questions', 'wpwa_questions' ),
    );
    $args = array(
        'labels' => $labels,
        'hierarchical' => true,
        'description' => __( 'Questions and Answers',
'wpwa_questions' ),
```

```

        '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( 'wpwa_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 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 to have a look at documentation on custom post creation at [http://codex.wordpress.org/Function\\_Reference/register\\_post\\_type](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. This is considered a bad practice as it becomes extremely difficult to maintain because your application becomes larger. So, we will be using plugins to add functionality throughout this book and drawbacks of the `functions.php` technique will be discussed in later chapters.

5. Once the code is included, you will get a new section on the admin area for creating questions. This section will be similar to the posts section inside the WordPress admin. Add few questions and insert some comments using different users before we move into the next stage.

Before we go into the development of questions and answers, we need to make some configurations so that our plugin works without any issues. Let's look at the configuration process:

1. First, we have to look at the comment-related settings inside **Discussion Settings** the WordPress **Settings** section. Here, you can find a setting called **Before a comment appears**.
2. Disable both checkboxes so that users can answer and get their answers displayed without approval process. Depending on the complexity of application, you can decide whether to enable these checkboxes and change the implementation.
3. The second setting we have to change is the **Permalinks**. Once we create a new custom post type and view it on browser, it will redirect you to a 404 page not found page. Therefore, we have to go to the **Permalinks** section of WordPress **Settings** and update the **Permalinks** using the **Save Changes** button. This won't change the **Permalinks**. However, this will flush the rewrite rules so that we can use the new custom post type without 404 errors.

Now, we can start working with the answer-related features.

## Customizing the comments template

Usually, the comments section is designed to show comments of a normal post.

While using comments for custom features such as answers, we need to customize the existing template and use our own designs.

1. So, open the `comments.php` file inside the Twenty Fourteen theme.
2. Navigate through the code and you will find a code section similar to the following one:

```
wp_list_comments( array(
    'style' => 'ol',
    'short_ping' => true,
    'avatar_size' => 34,
) );
```

3. 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 will change the previous implementation as following:

```
if(get_post_type( $post ) == "wpwa_question"){
    wp_list_comments('avatar_size=60&type=comment&callback=wpwa_comment_list&style=ol');
}else{
    wp_list_comments( array(
        'style' => 'ol',
        'short_ping' => true,
        'avatar_size' => 34,
    ) );
}
```

4. Here, we will include a conditional check for the post type in order to choose the correct answer list generation function. When the post type is `wpwa_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 answers list.



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

In the next section, we will be completing the customization of comments template by adding answer statuses and allowing users to change the statuses.

## Changing the status of answers

Once the users provide their answers, the creator of the question should be able to mark them as correct or incorrect answers. So, we will implement a button for each answer to mark the 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.

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. Hence, I'll be explaining the important sections of the code. It's ideal to work with the full code for the `wpwa_comment_list` function from the source code folder:

```
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
}
```

The `wpwa_comment_list` function 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 be only visible to the creator of the question.

In order to change the status of answers, follow these steps:

1. First, we will 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.
2. Next, we will check whether the logged-in user created the question. If so, we will set `show_answer_status` to true. Also, we have to retrieve the status of the current answer by passing the `comment_id` and `_wpwa_answer_status` keys to the `get_comment_meta` function.
3. Then, we will have to include the common code for generating comments list with necessary condition checks.

4. Open the `wpwa-questions.php` file of the plugin and go through the rest of `wpwa_comment_list` function to get an idea of how comments loop works.
5. Next, we have to highlight correct answers of each question and I'll be using an image as the highlighter. In the source code, we use 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>";
    }
?>
```

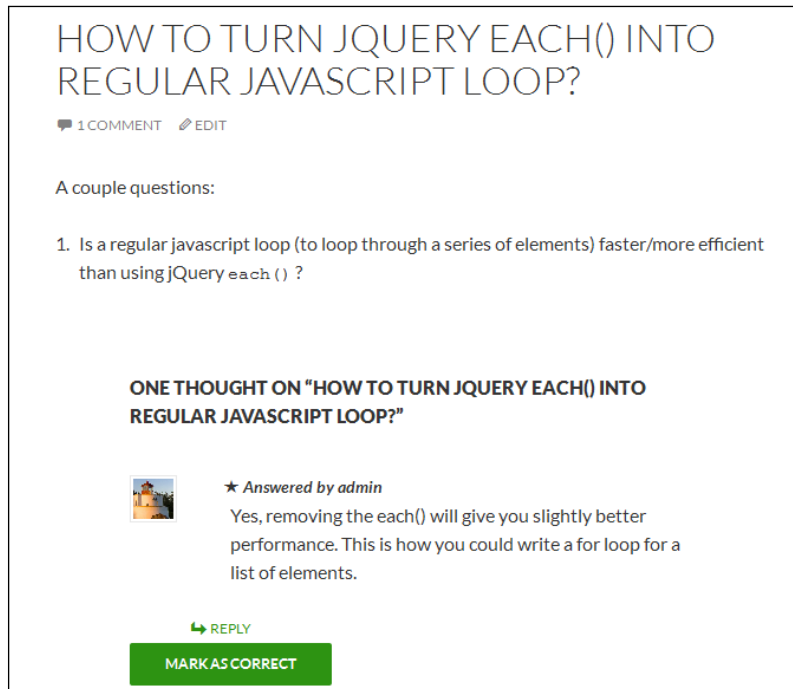
6. In the source code, you will see a `<div>` element with the class `reply` for creating the comment reply link. We will need to insert our answer button status code right after this, as shown in the following code:

```
<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', 'wpwa_questions');
            } else {
                $question_status = 'valid';
                $question_status_text = __('Mark as
Correct', 'wpwa_questions');
            }
        }
    ?>

    <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>
```

7. 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 will get the status of answer as `true` or `false` using the `_wpwa_answer_status` key from the `wp_commentmeta` table.
8. Based on the returned value, we will define buttons for either **Mark as Incorrect** or **Mark as Correct**. Also, we will specify some CSS classes and HTML5 data attribute to be used later with jQuery.
9. Finally, we keep the `comment_id` in a hidden variable called `hcomment`.
10. Once you include the code, the button will be displayed for the author of the question, as shown in the following screen:



11. Next, we need to implement the AJAX request for marking the status of the answer as `true` or `false`.

Before this, 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( 'wpwa-questions', plugins_url(
'js/questions.js', __FILE__ ), array('jquery'), '1.0', TRUE );
    wp_enqueue_script( 'wpwa-questions' );
    wp_register_style( 'wpwa-questions-css', plugins_url(
'css/questions.css', __FILE__ ) );
    wp_enqueue_style( 'wpwa-questions-css' );
    $config_array = array(
        'ajaxURL' => admin_url( 'admin-ajax.php' ),
        'ajaxNonce' => wp_create_nonce( 'ques-nonce' )
    );
    wp_localize_script( 'wpwa-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. The `wp_enqueue_script` action is used to include script files into the page while the `wp_register_script` action is used to add custom files. Since jQuery is built-in to WordPress, we can just use the `wp_enqueue_script` action 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 parameters 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, we discussed previously. Now, everything is set up properly to create the AJAX request.

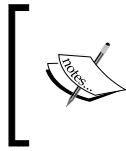


## Saving the status of answers

Once the author clicks 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:

```
jQuery(document).ready(function($) {
    $(".answer-status").click( function() {
        $(body).on("click", ".answer-status" , function() {
            // Get the button object and current status of the answer
            var answer_button = $(this);
            var answer_status = $(this).attr("data-ques-status");
            // Get the ID of the clicked answer using hidden field
            var comment_id = $(this).parent().find(".hcomment").val();
            var data = {
                "comment_id":comment_id,
                "status": answer_status
            };
            // Create the AJAX request to save the status to database
            $.post( wpwacnf.ajaxURL, {
                action:"mark_answer_status",
                nonce:wpwacnf.ajaxNonce,
                data : data,
            }, function( data ) {
                if("success" == data.status){
                    if("valid" == answer_status){
                        answer_buttonval("Mark as Incorrect");
                        answer_button.attr("
data-ques-status","invalid");
                    }else{
                        answer_button.val("Mark as Correct");
                        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 `wpwacnf` variable. Once a server returns the response with success status, the button will be updated to contain new status and display text.

The next step of this process is to implement the server-side code for handling AJAX request. 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 to mark the status of the answer using the `update_comment_meta` function. This example contains the most basic implementation of data saving process. In real applications, we need to implement necessary validations and error handling.

Now, the author who asked 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 questions list.

## Generating a 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:

1. Here, we will create a customized layout for our questions.
2. Make a copy of the existing `archive.php` file of the TwentyFourteen theme and rename it as `archive-wpwa_question.php`. Here, you will find the following code section:

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

3. The TwentyFourteen 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 a custom template called `content-questions.php` by duplicating the `content.php` file and change the preceding code to the following:

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

4. Finally, we need to consider the implementation of `content-wpwa_question.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 `the_excerpt` and the `the_content` functions of the template. We can comment the following line within this template:

```
the_content( __( 'Continue reading <span  
class="meta-nav">&rarr;</span>', 'twentyfourteen' ) );
```

5. Then, we will create our own metadata by adding the following code to the `<div>` element with the `entry-content` class:

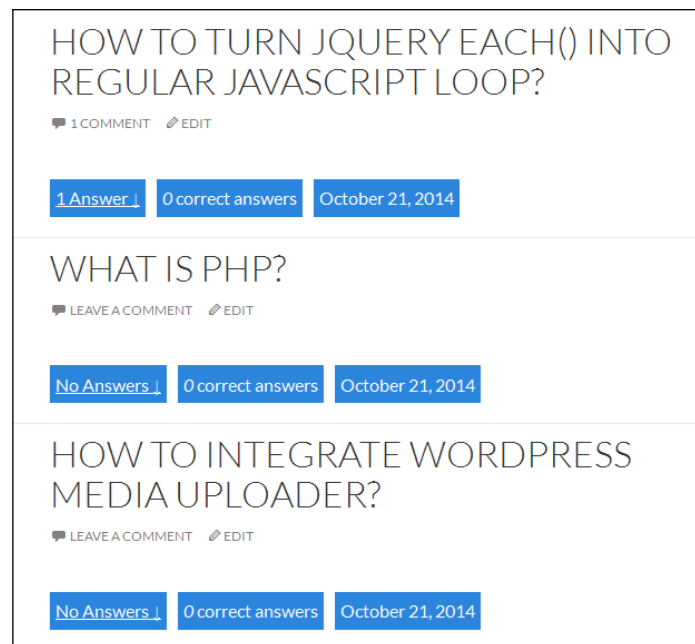
```
<div class="answer_controls"><?php  
comments_popup_link(__( 'No Answers &darr;', 'responsive'),  
__( '1 Answer &darr;', 'responsive'), __( '% Answers &darr;',  
'responsive' )); ?>  
</div>  
<div class="answer_controls">  
<?php wpwa_get_correct_answers(get_the_ID()); ?>  
</div>  
<div class="answer_controls">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

6. Then, we need to display the number of correct answers of 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 to retrieve the approved answers of each post, which 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 following image:



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 HTML and PHP code inside both, themes and plugins.



I suggest that you go through the Chapter01 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 develop a web application from scratch using detailed explanation in the following chapters.

## Enhancing features of the questions plugin

In the previous sections, we illustrated how to quickly adapt WordPress into different kind of implementations by customizing its core features. However, I had to limit the functionality to the most basic level in order to keep this chapter short and interesting for the beginners. If you are willing to improve it, you can try other new features of such an application. Here, we will be looking at some of the enhancements to this plugin and how we can use core WordPress features to implement them.

## Customizing the design of questions

This is one of the major requirements in such an application. Here, we have a very basic layout and it's difficult to know whether this is a question or just a normal post. Consider the following screenshot for a well-designed question interface:



Remember that we customized the comments list for adding new options. Similarly, we can customize the design of questions to create an interface such as the previous screenshot by using a separate template file for the questions custom post type. We have to create a file called `single-wpwa_question.php` and change the design and functionality as we want.

## Categorizing questions

Categories allow us to filter the results and limit it to a certain extent. It's an essential feature in the question-answer application so that users can directly view questions related to their topic instead of browsing all the questions. WordPress provides categories by default. However, these categories are mainly intended for posts. Therefore, we have to use custom taxonomies to create categories for custom post types such as questions. More details about taxonomies will be discussed in the following chapters.

## Approving and rejecting questions

Currently, our application can post questions without any approval process. However, this is not the ideal implementation as it can create a lot of spam questions. In a well-built application, there should be a feature to approve/reject questions when needed. This feature can be easily built with WordPress admin lists. We have a list of questions in the backend, and we can use the **Bulk Actions** dropdown on top to add custom actions and implement this feature.

## Adding star rating to answers

The author who asked the question can mark the answers as correct or incorrect. However, there can be scenarios where answers are not marked as correct are more suitable than answers marked as correct. Therefore, we can introduce star rating features to answers so that public can rate the answers. The person who looks for the answer can consider the rating before choosing answers. Implementation of this requirement is similar to the functionality of the **Mark as Correct** button. We have to get a `js` plugin for star rating and integrate to the interface through the `wpwa_comment_list` function. Then we can use AJAX to mark the status of answers.

We have looked at some of the possible enhancements to such an application. You can think of many more such functionalities and implement them in your own version.

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

## Summary

Our main goal was to find 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 features of WordPress and how each of those 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 existing functionalities.

By now, you should be able to decide whether to choose WordPress for your web application, visualize how your requirements fits into components of WordPress, and identify and minimize the limitations.

In the next chapter, we will start developing the portfolio management application with the user management. Before we go there, I suggest that you research user management features 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 a simple and interactive user management process, 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 will execute some tasks outside the box to bring user management features from WordPress core to WordPress themes.

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

In this chapter, we will cover the following topics:

- Introducing user management
- Understanding user roles and capabilities
- Creating a simple MVC-like process
- Implementing registration on the frontend
- Implementing login on the frontend
- Time to practice with exercises



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

## Introduction to user management

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

- Managing user roles and capabilities
- A built-in user registration functionality
- A built-in user login functionality
- A built-in forgot password functionality

Developers are likely to encounter these tasks in almost all web applications. In most cases, these features and functions can be effectively used without significant changes in 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 in order to be compatible with advanced application requirements. In the following sections, we will learn how to extend these common functionalities to suite various scenarios.

## Preparing the plugin

As developers, we have the option to build a complete application with standalone plugins or use various independent plugins to cater to specific modules. Generally, many developers tend to use a bunch of existing plugins without developing their own. The main reason behind using other plugins is that developers want to save time and cost when developing WordPress websites. However, in most scenarios, this process is not ideal for complex web application development. I recommend that you select fewer plugins and improve them with additional functionality or develop everything from scratch.

Throughout this book, we will be integrating all the independent modules into a standalone plugin. We will create a specific plugin for our portfolio management application. So, let's get started by creating a new folder called `wpwa-portfolio-manager`, inside the `/wp-content/plugins` folder.

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

```
<?php
/*
    Plugin Name: WPWA Portfolio Manager
    Plugin URI:
    Description: User management functionality 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 will take this 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-portfolio-manager.php` file:

```
class WPWA_Portfolio_Manager{
    public function __construct() {
        // Initialization code
    }
}

$wpwa_portfolio_manager = new WPWA_Portfolio_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 with 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 will 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. You can find out more about existing user roles at [http://codex.wordpress.org/Roles\\_and\\_Capabilities](http://codex.wordpress.org/Roles_and_Capabilities). In this scenario, `read`, `edit_posts`, and `delete_posts` will be the capabilities while `true` and `false` is used to enable and disable status.

## Creating application user roles

As planned earlier, we will need three types of user roles for our application to handle subscribers, developers, and members. So, we can 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 ));
}
```

Application user roles are created with the default capability of `read`, used by all the user roles in WordPress. Initialization of this function should be done inside the constructor of our plugin.

## The best action for adding user roles


The user roles discussed in the previous section will be saved in the database as settings. Therefore, only a single call to this function is required throughout the life cycle of an application. We have two options for implementing this functionality. Application installation and plugin activation are the most suitable places to call these kinds of functions to eliminate duplicate executions.

- **Application installation:** WordPress provides a well-defined step-by-step installation process. Similarly, every application or plugin needs an installation process. Once installation is completed, these files are not accessible again. Therefore, plugin installation is the ideal place to create user roles.
- **Plugin activation:** WordPress plugin activation hooks let us execute certain functionality on plugin activation. This is also a one-time process in a plugin. However, we can deactivate and reactivate the plugin multiple times. So, this functionality gets executed multiple times and hence, we have to check whether it's already been executed. If this is not checked, all the changes made after the activation will be reset to default values. So, plugin activation is the second best option for this type of functionality.

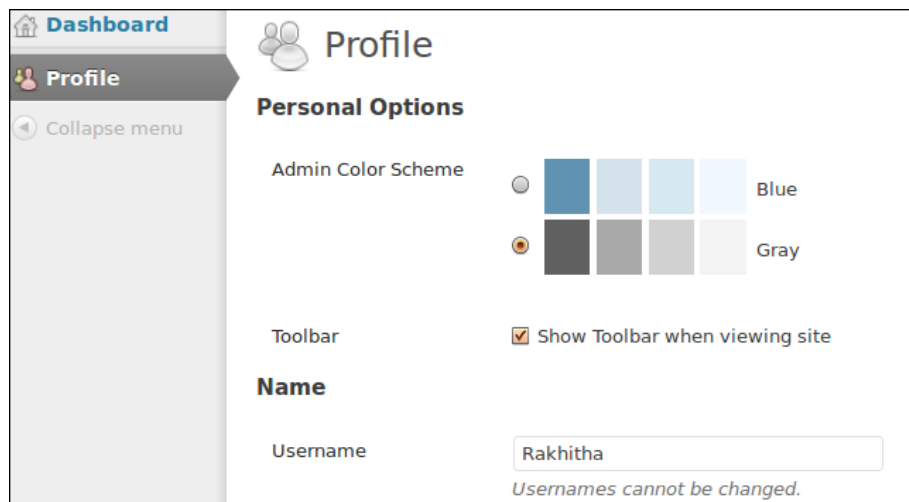
Here, we will be using plugin activation to add and remove application user roles. Later, we will be moving this functionality into installation, after defining the necessary functionality for setup. So, we need to include the call to the `add_application_users` function inside the constructor, as shown 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 duplicate calls to 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 to create 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 a dashboard and profile information, as shown in the following screenshot:



## Knowing the default roles of WordPress

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

- **Superadmin:** A superadmin has administration permission in WordPress multisite implementation
- **Admin:** An admin has permission to all administration activities inside a single site
- **Editor:** An editor can create, publish, and manage posts, including the posts of other users
- **Author:** An author can create and publish their own posts
- **Contributor:** A contributor can create posts, but cannot publish on their own
- **Subscriber:** A subscriber can read posts and manage their profile

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.

## Choosing among default and custom roles

This is an interesting question that 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 admin. So, they are similar to contributors. Companies can create and publish their own job posts, similar to authors. An admin can play the role of an editor or admin in the default system.

Even though we can match certain aspects of our portfolio application roles with the existing roles, we will 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 the 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 the unnecessary user roles from the system, as illustrated 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, which 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 custom capability with a follower user role, created in the earlier section, *Creating application user roles*:

```
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 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 in 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_options where
option_name='wp_user_roles'
```

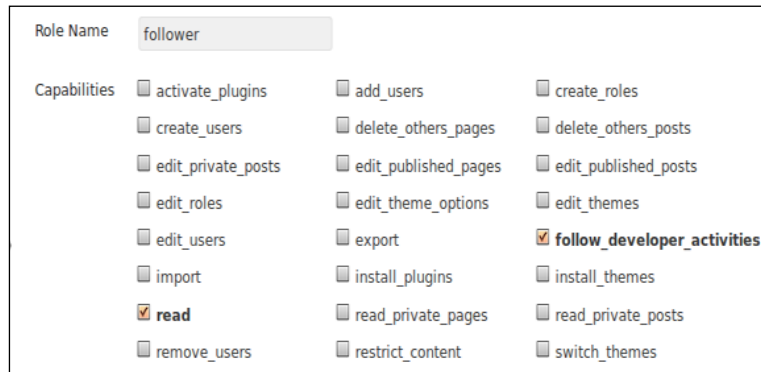
You should see a serialized array like the following one:

```
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_theme";b:1;s:16:"activate_plugins";b:1;s:12:"edit_plugins";b:1;s:10:"edit_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, which are assigned to a specific user role, 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 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 following chapters. Next, we will see how user registration works in WordPress.

## Registering application users

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

- **User-friendly interface:** An application can have different types of user roles. Until registration is completed, 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 such as with WordPress. Therefore, we need to explore the possibilities of adding 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 the user. 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 registration process in web applications. Complex applications may come with more requirements in this process. Therefore, we need to extend the WordPress registration process, in order to cater to various requirements. In the next section, we will address the issues mentioned here by creating a WordPress registration from the frontend.

## Implementing frontend registration

Fortunately, we can make use of the existing functionalities to implement registration from the frontend. We can use a regular HTTP request or AJAX-based technique to implement this feature. In this book, I will focus on a normal process instead of using AJAX. Our first task is to create the registration form in the frontend.

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

- 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 to add 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

Following are the pros and cons of using shortcodes:

- Shortcodes are easy to implement in any part of your application
- Its hard to manage the template code assigned using the PHP variables
- There is a possibility of the shortcode getting 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

Following are the pros and cons of page templates:

- A page template is more stable than shortcode.
- Generally, page templates are associated with the look of the website rather than providing dynamic forms. The full width page, two-column page, and left sidebar page are some common implementations of page templates.
- A template is managed separately from logic, without using PHP variables.
- The page templates depend on the theme and need to be updated on theme switching.

## Custom template implementation

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

## Building a simple router for a user module

Routing is one of the important aspects in advanced application development. We need to figure out ways of building custom routes for specific functionalities. In this scenario, we will 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, `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 used to create custom rewrite rules. We will 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 will define the regular expression for matching all the routes in the `user` section. Redirection is made to the `index.php` file with a query variable called `control_action`. This variable will contain the URL 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 them. The value of `top` will give a higher precedence, while the value of `bottom` will give a lower precedence.

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

1. Add query variables to the WordPress `query_vars`
2. 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 we want to use a new query variable, make sure to add it to the existing list. First, we 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 list of existing variables by adding or removing entries from an array. Now, consider the implementation to add 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, the existing `query_vars` variable will be passed as an array. We will modify the array by adding a new query variable called `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 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 that you perform such tasks in plugin activation or installation as we did earlier in 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 URL <http://www.example.com/user/login> and check whether 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 persist successfully. However, we will 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 flushing is executed. Now, follow the previous process once again; you won't get a 404 page since all the rules have taken effect.



You can get 404 errors due to the modification in rewriting rules and not flushing it 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 the previous section.

So, I recommend that you use the free plugin called 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 screen:

Rule	Rewrite	Source
<code>user/([ ^/]+)/?</code>	<code>index.php?control_action=\$matches[1]</code>	other
<code>category/(.+?) /feed/((feed rdf rss rss2 atom))/?\$</code>	<code>index.php?category_name=\$matches[1]&amp;feed=\$matches[2]</code>	category
<code>category/(.+?) /(feed rdf rss rss2 atom)/?\$</code>	<code>index.php?category_name=\$matches[1]&amp;feed=\$matches[2]</code>	category

## Controlling access to your functions

We have a custom router, which 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 called `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 consider will be to register as we are in the registration process. Once the `control_action` query variable is matched with registration, we will 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 the typical scenario where we call a function within the class or in an external class to implement the registration. In the previous code, we called a function within the class, but with the `do_action` hook instead of the usual function call.

## The advantages of using the `do_action` function

WordPress action hooks define specific points in the execution process, where we can develop custom functions to modify 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 a bad practice by looking at the definition of the open/closed principle on Wikipedia.

*"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 implemented a simple controller, which can be quite effective in developing web application functionalities. In the following sections, we will 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 possible reasons:

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

As a solution to these concerns, we will implement the custom templates inside the plugin. First, create a folder inside the current plugin folder and name it as `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 whole content area will be used for the registration and the default sidebar will be omitted for this screen. Create a PHP file called `register-template.php` inside the `templates` folder with the following code:

```
<?php get_header(); ?>
<div id="wpwa_custom_panel">
```

---

```

<?php
if( isset($errors) && count( $errors ) > 0) {
    foreach( $errors as $error ){
        echo '<p class="wpwa_frm_error">'. $error .'';
    }
}
?>
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 will 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='wpwa_frm_label'><?php echo
__('Username','wpwa'); ?></label>
            <input class='wpwa_frm_field' type='text'
id='wpwa_user' name='wpwa_user' value='' />
        </li>
        <li>
            <label class='wpwa_frm_label'><?php echo __('E-
mail','wpwa'); ?></label>
            <input class='wpwa_frm_field' type='text'
id='wpwa_email' name='wpwa_email' value='' />
        </li>
        <li>
            <label class='wpwa_frm_label'><?php echo __('User
Type','wpwa'); ?></label>
            <select class='wpwa_frm_field' name='wpwa_user_type'>
                <option <?php echo __('Follower','wpwa');
?></option>
                <option <?php echo __('Developer','wpwa');
?></option>
                <option <?php echo __('Member','wpwa');
?></option>
            </select>
        </li>
        <li>
            <label class='wpwa_frm_label' for=''>&nbsp;</label>

```

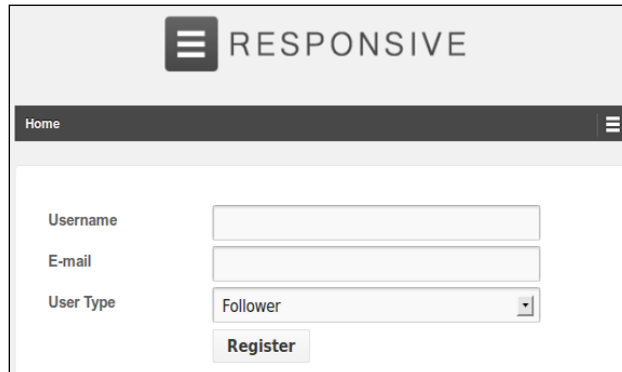
```
        <input type='submit' value='<?php echo  
__('Register','wpwa'); ?>' />  
    </li>  
</ul>  
</form>
```

As you can see, the form action is set to a custom route called `user/register` to be handled through the front controller. Also, we have added an extra field called `user type` to choose the preferred user type on registration.



You might have noticed that we used `wpwa` as the prefix for form element names, element IDs, as well as CSS classes. Even though it's not a must to use a prefix, it can be highly effective when working with multiple third-party plugins. A unique plugin-specific prefix avoids or limits conflicts with other plugins and themes.

We will get a screen similar to the following one, once we access the `/user/register` link 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 into 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
- The 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
- Users should activate their account by clicking 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-  
template.php';  
        exit;  
    }  
}
```

Once 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 a 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 the submit button, we have to execute quite a few tasks in order to register a user in WordPress database. Let's figure out the main tasks for registering a user:

- Validating form data
- Registering the user details
- Creating and saving activation code
- Sending e-mail notifications with an activate 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 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['wpwa_user'] ) ?
$_POST['wpwa_user'] : '' );
$user_email = ( isset ( $_POST['wpwa_email'] ) ?
$_POST['wpwa_email'] : '' );
$user_type = ( isset ( $_POST['wpwa_user_type'] ) ?
$_POST['wpwa_user_type'] : '' );
// Validating user data
if ( empty( $user_login ) )
    array_push($errors, __('Please enter a username.','wpwa') );
if ( empty( $user_email ) )
    array_push( $errors, __('Please enter e-mail.','wpwa') );
if ( empty( $user_type ) )
    array_push( $errors, __('Please enter user type.','wpwa') );
}
// Including the template
}
```

The following steps are to be performed:

1. First, we will check whether the request is made as POST.
2. Then, we get the form data from the POST array.
3. Finally, we will 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.','wpwa'));
elseif ( email_exists( $user_email ) )
    array_push( $errors, __('User with this email already
registered.','wpwa'));
if ( empty( $sanitized_user_login ) || !validate_username(
$user_login ) )
    array_push( $errors, __('Invalid username.','wpwa') );
elseif ( username_exists( $sanitized_user_login ) )
    array_push( $errors, __('Username already exists.','wpwa') );
```

The steps to perform are as follows:

1. First, we will use the existing `sanitize_user` function and remove unsafe characters from the username.
2. Then, we will make validations on the e-mail to check whether it's valid and its existence status in the system. Both the `email_exists` and `username_exists` functions checks for the existence of an 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 the registration form. You can add more advanced validation in your implementations in order to minimize 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 to the registration template. Here is a preview of our registration screen with generated error messages:

The screenshot shows a web browser window with a dark header bar containing the word "Home" and a hamburger menu icon. Below the header, a red rectangular box contains three lines of white text: "Please enter a username.", "Please enter e-mail.", and "Invalid username.". Below this box, the registration form is displayed. It has three input fields: "Username" (empty), "E-mail" (empty), and "User Type" (a dropdown menu with "Follower" selected). Below the input fields is a "Register" button.

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 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='wpwa_frm_label'><?php echo
__('Username','wpwa'); ?></label>
        <input class='wpwa_frm_field' type='text'
id='wpwa_user' name='wpwa_user' value='<?php echo isset(
$user_login ) ? $user_login : ''; ?>' />
    </li>
    <li>
        <label class='wpwa_frm_label'><?php echo __('E-
mail','wpwa'); ?></label>
        <input class='wpwa_frm_field' type='text'
id='wpwa_email' name='wpwa_email' value='<?php echo isset(
$user_email ) ? $user_email : ''; ?>' />
    </li>
    <li>
        <label class='wpwa_frm_label'><?php echo __('User
Type','wpwa'); ?></label>
        <select class='wpwa_frm_field' name='wpwa_user_type'>
            <option <?php echo (isset( $user_type ) &&
$user_type == 'follower') ? 'selected' : ''; ?> value='follower'><?php
echo __('Follower','wpwa'); ?></option>
            <option <?php echo (isset( $user_type ) &&
$user_type == 'developer') ? 'selected' : ''; ?>
value='developer'><?php echo __('Developer','wpwa'); ?></option>
            <option <?php echo (isset( $user_type ) && $user_type ==
'member') ? 'selected' : ''; ?> value='member'><?php
echo __('Member','wpwa'); ?></option>
        </select>
    </li>
    <li>
        <label class='wpwa_frm_label' for=''>&nbsp;</label>
        <input type='submit' value='<?php echo
__('Register','wpwa'); ?>' />
    </li>
</ul>
</form>
```

## Exploring the registration success path

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.','wpwa') );
    } else {
        $activation_code = $this->random_string();
        update_user_meta( $user_id, 'wpwa_activation_code',
$activation_code );
        update_user_meta( $user_id, 'wpwa_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.','wpwa');
    }
    if ( !is_user_logged_in() ) {
        include dirname(__FILE__) . '/templates/login-template.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 in the database.

 The `wp_insert_user` function will be used to update the current user or add new users 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, we can create a registration fail message and assign it to the `$errors` variable as we did earlier. Once the registration is successful, we will 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 activation code and set the activation status as inactive for the moment. Finally, we will 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 the 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 will implement it as a standalone function inside our main plugin file. The full 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:

```
$activate_link = site_url() .
"/user/activate/?wpwa_activation_code=$activate_code";
$message = __('Hi there,') . '\r\n\r\n';
$message .= sprintf(__('Welcome to %s! Please activate your
account using the link:', 'wpwa'), get_option('blogname')) .
'\r\n\r\n';
$message .= sprintf(__('<a href="%s">%s</a>', 'wpwa'),
$activate_link, $activate_link) . '\r\n';
$message .= sprintf(__('Username: %s', 'wpwa'), $user_login) .
'\r\n';
$message .= sprintf(__('Password: %s', 'wpwa'), $plaintext_pass) .
'\r\n\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 into 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 their inbox, they can use the activation link to activate the account.

## Automatically log in the user after registration

In general, most web applications uses e-mail confirmations before allowing users to log in to the system. However, there can be certain scenarios where we need to automatically authenticate the user into the application. A social network sign in is a great example for such a scenario. When using social network logins, the system checks whether the user is already registered. If not, the application automatically registers the user and authenticates them. We can easily modify our code to implement an automatic login after registration. Consider the following code:

```
if ( !is_user_logged_in() ) {
    wp_set_auth_cookie($user_id, false, is_ssl());
    include dirname(__FILE__) . '/templates/login-template.php';
    exit;
}
```

The registration code is updated to use the `wp_set_auth_cookie` function. Once it's used, the user authentication cookie will be created and hence the user will be considered as automatically signed in. Then, we will redirect to the login page as usual. Since the user is already logged in using the authentication cookie, they will be redirected back to the home page with access to the backend. This is an easy way of automatically authenticating users into WordPress.

## Activating system users

Once the user clicks on the activate link, redirection will be made 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['wpwa_activation_code'] ) ?
    $_GET['wpwa_activation_code'] : '';
    $message = '';
    // Get activation record for the user
    $user_query = new WP_User_Query(
        array(
            'meta_key' => 'wpwa_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, 'wpwa_activation_status',
        'active' );
        $message = __( 'Account activated successfully.', 'wpwa' );
    } else {
        $message = __( 'Invalid Activation Code', 'wpwa' );
    }
}
```

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

We will get the activation code from the link and query the database for finding a matching entry. If no records are found, we set the message as activation failed or else, we can update the activation status of the matching user to activate the account. Upon activation, the user will be given a message using the `info-template.php` template, which consists of a very basic template like the following one:

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

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



We 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 form in the frontend

The 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 will implement a 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 called login-template.php. Here is the implementation of the login form design with the necessary error messages:

```

<?php get_header(); ?>
<div id=' wpwa_custom_panel'>
    <?php
    if (isset($errors) && count($errors) > 0) {
        foreach ($errors as $error) {
            echo '<p class="wpwa_frm_error">' . $error. '</p>';
        }
    }
    if( isset( $success_message ) && $success_message != "" ){
        echo '<p class="wpwa_frm_success">' . $success_message.
    '</p>';
    }
    ?>
    <form method='post' action='<?php echo site_url();
    ?>/user/login' id='wpwa_login_form' name='wpwa_login_form'>
        <ul>
            <li>
                <label class='wpwa_frm_label' for='username'><?php
                echo __('Username','wpwa'); ?></label>
                <input class='wpwa_frm_field' type='text'
                name='wpwa_username' value='<?php echo isset( $username ) ?
                $username : ''; ?>' />
            </li>
            <li>
                <label class='wpwa_frm_label' for='password'><?php
                echo __('Password','wpwa'); ?></label>
                <input class='wpwa_frm_field' type='password'
                name='wpwa_password' value="" />
            </li>
            <li>
                <label class='wpwa_frm_label' >&nbsp;</label>
                <input type='submit' name='submit' value='<?php echo
                __('Login','wpwa'); ?>' />
            </li>
        </ul>
    </form>
</div>
<?php get_footer(); ?>

```

Similar to the registration template, we have a header, error messages, the HTML form, and the 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 make detailed explanations. You can take a look at the preview of our login screen in the following screenshot:



Next, we need to implement the form submission handler for the login functionality. Before this, 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 in the frontend.

## Displaying the login form

We will 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-template.php';
    } else {
        wp_redirect(home_url());
    }
    exit;
}
```

First, we need to check whether the user has already logged in to the system. Based on the result, we will 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['wpwa_username'] ) ?
    $_POST['wpwa_username'] : '';
    $password = isset ( $_POST['wpwa_password'] ) ?
    $_POST['wpwa_password'] : '';
    if ( empty( $username ) )
        array_push( $errors, __( 'Please enter a username.', 'wpwa' ) );
    if ( empty( $password ) )
        array_push( $errors, __( 'Please enter password.', 'wpwa' ) );
    if (count($errors) > 0){
        include dirname(__FILE__) . '/templates/login-template.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 the necessary errors to be shown in the frontend. Once validations are successfully completed, we assign all the form data to an array after sanitizing the values. The username and password are contained in the credentials array with the `user_login` and `user_password` keys. The `remember` 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 the `WP_Error` class based on the result.

 Internally, this function sets an authentication cookie. Users will not be logged in if it is not set. If you are using any other process for authenticating users, you have to set this authentication cookie manually.

Once a user is successfully authenticated, a redirection will be made to the home page of the site. Now, we should have the ability to authenticate users from the login form in the frontend.

## Checking whether we implemented the process properly

Take a moment to think carefully about our requirements and try to figure out what we have missed.

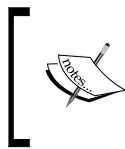
Actually, we didn't check the activation status on log in. Therefore, any user will be able to log in to the system without activating their account. Now, let's fix this issue by intercepting the authentication process with another built-in action called `authenticate`, as shown in the following code:

```
public function authenticate_user( $user, $username, $password ) {
    if(! empty($username) && !is_wp_error($user)){
        $user = get_user_by('login', $username );
        if (!in_array( 'administrator', (array) $user->roles ) ) {
            $active_status = '';
            $active_status = get_user_meta( $user->data->ID, 'wpwa_
activation_status', true );
            if ( 'inactive' == $active_status ) {
                $user = new WP_Error( 'denied', __( '<strong>ERROR</strong>:
Please activate your account.', 'wpwa'
) );
            }
        }
    }
    return $user;
}
```

This function will be called in the authentication action by passing the `user`, `username`, and `password` variables 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 they are admin. Then, we can check the activation status of other user types before authenticating. If an 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, to make it work as shown in the following code:

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



This filter is also executed when the user logs out of the application. Therefore, we need to consider the following validation to prevent any errors in the logout process:

```
if(! empty($username) && !is_wp_error($user))
```

Now, we have a simple and useful user registration and login system, ready to be implemented in 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. Before we have a complete user creation and authentication system, there are plenty of other tasks to be completed. So, I would recommend you to try out the following tasks in order to be comfortable with implementing such functionalities for web applications:

- Create a frontend functionality for the lost password
- Block the default WordPress login page and redirect it 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 on 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 frontend controller and custom template system.

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

By now, you should be capable of defining user roles and capabilities to match your application, create custom routers for common modules, implement custom controllers with custom template systems, and customize the existing user registration and authentication process.

In the next chapter, we will 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

Generally, a database acts as the primary location to keep your web application data to be 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 will focus on the basics of planning and accessing database for web applications. This chapter is important for the rest of the book and can appear theoretical compared to other chapters.

In this chapter, we will cover the following topics:

- Understanding the WordPress database
- Exploring the role of existing tables
- Adapting existing tables into web applications
- Extending the 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 predefined database structure. Instead, these frameworks focus on the core foundation of an application while allowing the developers to focus on the 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. The WordPress core database is designed to power the generic functionalities of a CMS. Therefore, it's our responsibility to use our skills to make it work as an application development framework.

The 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 eleven database tables to cater core functionality with the default MySQL table engine.



MyISAM was used as the default MySQL table engine prior to version 5.5.5 and this has been changed to InnoDB from version 5.5 onwards.

WordPress core features will always be limited to these eleven 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 about the existing tables in order to associate them in web applications.

## Exploring the role of existing tables

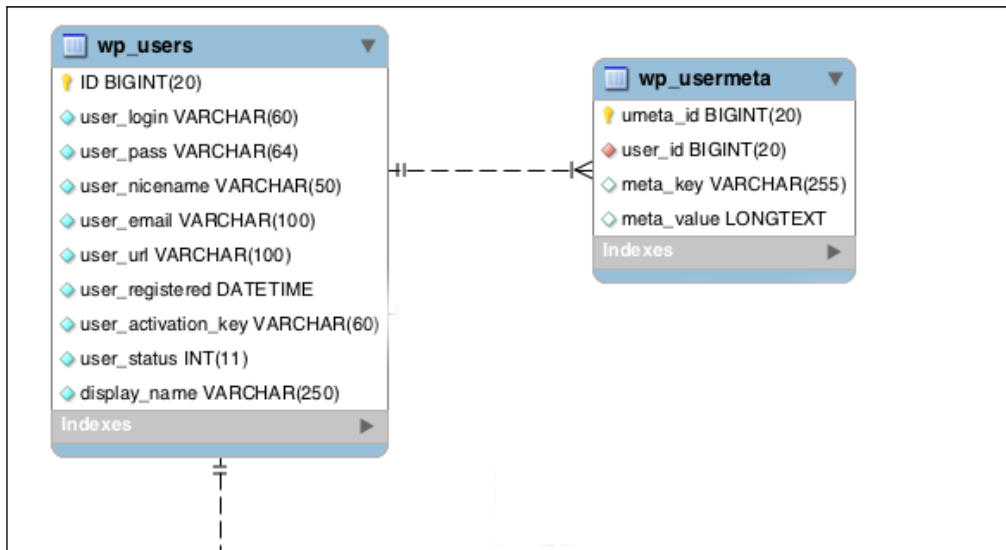
Assuming that most of you are existing WordPress developers, you will have a solid understanding of an existing database table structure. However, I suggest that you continue with this section as web applications can have a different perspective on using these tables. Based on the functionality, we will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

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

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


## User-related tables

This section consists of two tables for keeping the user-related information of your application. Let's take a look at the relationship between user-related tables before moving onto the explanations.



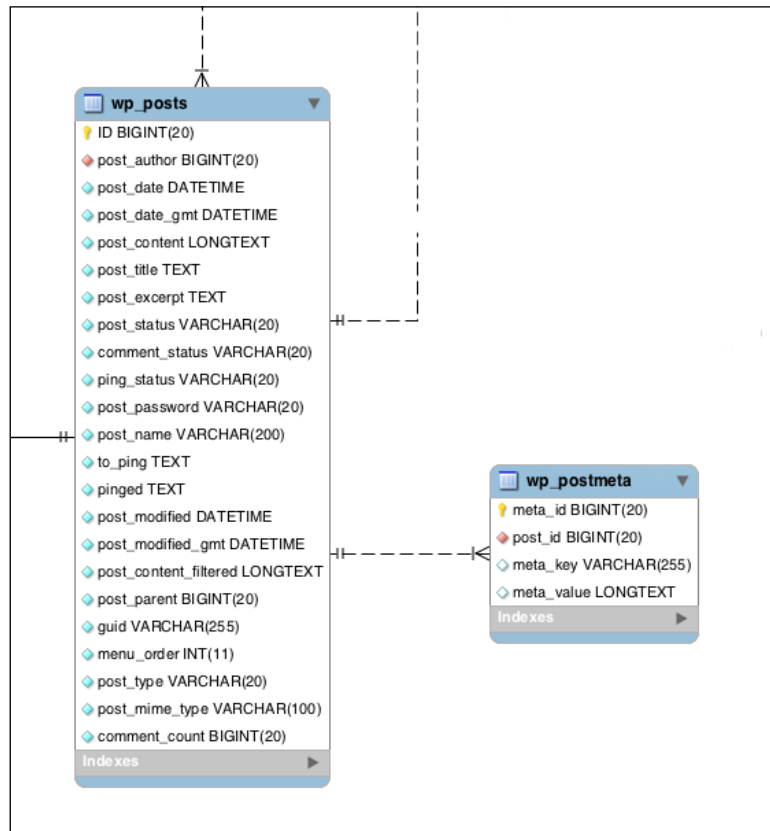
The two tables shown in the preceding diagram are as follows:

- **wp\_users:** All the registered users will be stored in this table with their basic details such as name, e-mail, username, password, and so on.
- **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 to add any user-related information as new key-value pairs.


 Throughout this chapter, we'll be referring to WordPress tables with the default prefix of `wp_`. You can change the prefix through the installation process or by manually changing the `wp-config.php` file in the root directory.

## Post-related tables

This section consists of two tables to keep website posts- and page-related information. Let's take a look at the relationship between post-related tables before moving onto the explanations.

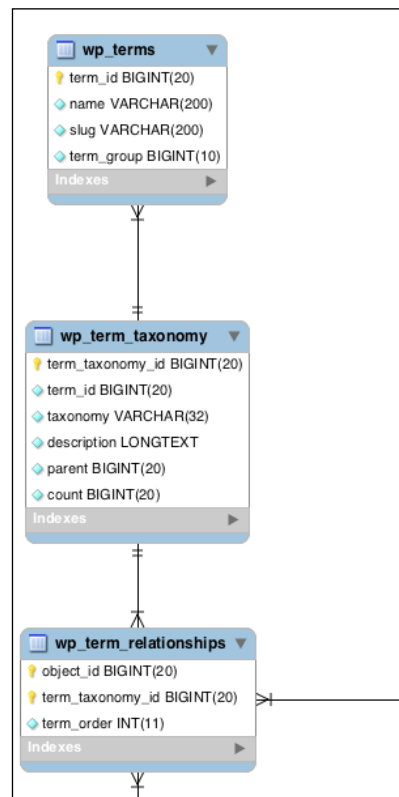


The tables shown in the diagram are as follows:

- **wp\_posts**: This table is used to keep all the posts and pages of your website with the details such as post name, author, content, status, post type, and so on.
- **wp\_postmeta**: This table is used to keep all the additional details for each post as key-value pairs. By default, it will contain details such as page template, attachments, edit locks, and so on. 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 between term-related tables:



The three tables shown in the diagram are as follows:

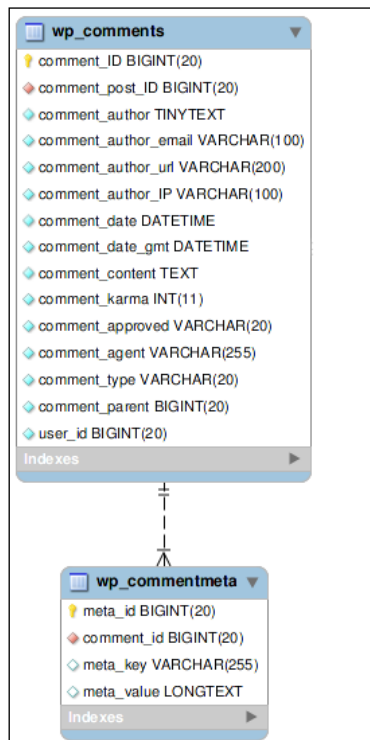
- **wp\_terms**: This table contains master data for all new categories and tags, including custom taxonomies.
- **wp\_term\_taxonomy**: This table is used to define the type of terms and the number of posts or pages available for each term. Basically, all the terms will be categorized as category, post-tags, 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 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 feedback for posts and pages. Comments-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 screen previews the relationship between comment-related tables:



The tables shown in the diagram are as follows:

- `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 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/9/97/WP3.8-ERD.png>.

Now, you should have a clear idea of the role of existing tables and the reasons for their existence from a 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 it becomes 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 we can build using existing tables and the features that need their own table structures.

We should be trying 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 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

In web applications, the user table plays the same role as in a 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 called `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 called `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 core functionality. There is every chance that other plugin developers are using these fields with a different meaning and thus creating the possibility of lost 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 the existing columns of core tables.

Therefore, we choose the `wp_usermeta` table to keep the activation status of all users. Other plugin developers can also implement similar functionalities with unique keys inside the `wp_usermeta` table. In short, the `wp_usermeta` table can be used effectively to create advanced user-related functionalities in web applications as long as it doesn't involve one-to-many relationship. It's possible to use multiple metafields 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 a 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 ability to match most of our application data in 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 `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 – A 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 and check out dates, and number of people, can be created using additional fields in the `wp_postmeta` table.

## Scenario 3 – The project management application

Let's consider a much more advanced scenario in creating a relationship between 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 tasks using taxonomies or a separate custom table.

Till now, we discussed three completely different scenarios in real-world applications, and we were able to match all the requirements with custom post types. Now, you should be able to understand the importance of these two tables in web application development. In the next chapter, we will be continuing our exploration of custom post types and the use of the `wp_posts` and `wp_postmeta` 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 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 ebooks, 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 a given task

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

## Other tables

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



The 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 their 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 currently supported in WordPress.

Let's recall the example in *Chapter 1, WordPress as a Web Application Framework*, where we created the question and answer interface using posts and comments. Answers were considered as a 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 called `comment_type` is not available, we have to use a meta key called `wpwa_comment_type` to filter different comments from each other.

Finally, we will 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 amount 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, which increases the performance due to a limited number of table records.

Up until this point, we explored the role of existing tables and how we can adapt them in real-world web applications. A complex web application 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 following section.

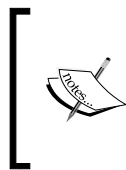
## Extending the database with custom tables

A default WordPress database 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 to existing tables:** In the previous section, we considered real application requirements and matched the data to existing tables. Unfortunately, it's not practical in all the 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 in a posts table. Let's assume that we are building a product catalog that creates millions of orders. Storing order details in the posts table as a custom post type is not the ideal implementation. In such circumstances, the posts table will 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

The portfolio management system developed throughout this book will make use of existing tables in every possible scenario. However, it's hard to imagine even an average web application without using custom tables, so here we will identify the possible custom tables for our system. You might need to revert back to the planning section in the *Development plan for portfolio management application* section of *Chapter 1, WordPress as a Web Application Framework*, in order to remind the system requirements. We planned to create a functionality to allow subscribers to follow developers in the system. Let's discuss the requirement 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 to be compatible with complex systems. Developers are stored as users of the system. Therefore, we only have a 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 called `user_activities` to implement this feature.

## Types of tables in web applications

Database tables of web applications can be roughly categorized into three sections as follows:

- **Master tables:** These tables contain predefined or configuration data for the application, which rarely gets changed. The options table can be considered as the perfect example of this type of table in the WordPress context.
- **Application data tables:** These tables contain the highly dynamic core application data. Posts and users can be considered as good examples of 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 an increasing speed. It's difficult to find good examples of these types of tables in the WordPress context.

Based on these 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 table called `subscribed_developers`. We can assume that most of the transaction tables will need their own custom tables. For now, we will stick with these two tables and additional custom tables will be added in later chapters when needed.

## Creating custom tables

In typical circumstances, 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 or installation. This process is similar to the user role creation process in *Chapter 2, Implementing Membership Roles, Permissions, and Features*.

We will be using activation-based table creation in this book. However, you can try the installation-based table creation to cater to advanced scenarios. We will add the database table creation functionality into the `wpwa-portfolio-manager` plugin we created in *Chapter 2, Implementing Membership Roles, Permissions, and Features*. So, let's get started by creating a new function called `create_custom_tables` inside the `class-wpwa-portfolio-manager.php` file:

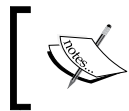
```
<?php
    public function create_custom_tables() {
        // Creating Database Tables
    }
?>
```

Now we can implement the `create_custom_tables` function to create 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 the two tables for our application using the `dbDelta` function:

```
public function create_custom_tables() {
    global $wpdb;
    require_once( ABSPATH . 'wp-admin/includes/upgrade.php' );
    $user_activities_table = $wpdb->prefix.'user_activities';
```

```
if($wpdb->get_var("show tables like '$user_activities_table'")
!= $user_activities_table) {
    $sql = "CREATE TABLE $user_activities_table (
        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
}
```

Firstly, 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 called `wp_` for all the tables. It's important to use the existing prefix to keep the consistency and avoid issues in multi-site scenarios. Next, we have to check the existence of a database table using the `show tables` query. 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](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 have the function for creating custom tables for our application. This function needs to be executed through the plugin activation hook. In *Chapter 2, Implementing Membership Roles, Permissions, and Features*, we executed multiple functions through multiple calls to `register_activation_hook`. However, it's more efficient to use a single call to `register_activation_hook` to handle multiple functions. Therefore, we will change the implementation of activation initialization function as follows:

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

Now we can execute all the activation-related functions inside the `activate_portfolio_manager` function as follows:

```
public function activate_portfolio_manager(){
    // Creates all the user types
    $this->add_application_user_roles();
}
```

---

```

// Remove unused user roles
$this->remove_application_user_roles();
// Create custom capabilities for user roles
$this->add_application_user_capabilities();
// Flush rewrite rules
$this->flush_application_rewrite_rules();
// Create custom tables
$this->create_custom_tables();
}

```


We created the custom tables using the `dbDelta` function inside the plugin activation. WordPress recommends the `dbDelta` function over direct SQL queries for table creation since it examines the current table structure, compares it to the desired table structure, and makes the necessary modifications without breaking the existing database tables. Apart from the 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 number of limitations, as listed here:

- Custom tables are hard to manage in WordPress upgrades.
- WordPress default backups will not include custom tables.
- No built-in functions for accessing database. All the queries, filtering, and validation need to be done from scratch, using the existing `$wpdb` variable.
- User interfaces for displaying these table data needs to be created from scratch.

Therefore, you should avoid creating custom tables in all possible circumstances, 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 look at the source code at <http://wordpress.org/plugins/pods/> for learning the use of custom tables.


]

A detailed exploration about the PODS framework will be provided in the next chapter, *Chapter 4, 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. A complete guide for using the `wpdb` class can be found at [http://codex.wordpress.org/Class\\_Reference/wpdb](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, 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 creates a new post or page in the `wp_posts` table. If this is used on an existing post, it will update the existing record
- `add_option`: This creates a new option on the `wp_options` table, if it doesn't already exist
- `wp_insert_comment`: This creates a new comment on 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 creates or updates additional details about posts in the `wp_postmeta` table
- `wp_update_term`: This updates existing terms in the `wp_terms` table
- `update_user_meta`: This updates user meta details in the `wp_usermeta` table based on the user ID

## Deleting records

We have similar methods for deleting records in each of the existing tables as we have for updating records. The following list illustrates a few of the built-in delete functions:

- `delete_post_meta`: This deletes custom fields using the specified key in the `wp_postmeta` table
- `wp_delete_post`: This removes existing posts, pages, or attachments from the `wp_posts` table
- `delete_user_meta`: This removes the metadata matching criteria from a user from the `wp_usermeta` table

## Selecting records

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

- `get_posts`: This 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 necessary arguments to get the post list from the OOP method.
- `get_option`: This retrieves the option value of the given key from the `wp_options` table.
- `get_users`: This retrieves a list of users as an array from the `wp_user` table.

Most of the database operations on exiting tables can be executed using these built-in functions. Therefore, use of the `$wpdb` class is not necessary in most occasions unless queries become complex and difficult 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:

- `$wpdb->get_results( "select query" )`: This can be used to select a set of records from any database table.
- `$wpdb->query( 'query' )`: This can be used to execute any custom query. This is typically used to update and delete statements instead of select statements, as it only provides the affected rows count as the result.

- `$wpdb->get_row('query')`: This can be used to retrieve a single row from the database as an object, an associative array, or as a numerically indexed array.

A complete list of the `wpdb` class functions can be accessed at [http://codex.wordpress.org/Class\\_Reference/wpdb](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 existing tables. For example, consider the following query for proper usage of these functions with 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 the functions such as `escape` and `escape_by_ref` to secure direct SQL queries.

Data validation is an important aspect of keeping the consistency of the database. WordPress offers the `prepare` function for formatting 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 higher enforces 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 up with a separate class called `WP_Query` for interacting with posts and pages. You can look at more details about the use of `WP_Query` at [http://codex.wordpress.org/Class\\_Reference/WP\\_Query](http://codex.wordpress.org/Class_Reference/WP_Query).

Up to now, we've looked 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 subclasses to cater to project-specific requirements. In the next section, we will learn how to extend the `WP_Query` class to create custom database access interfaces.

## Extending the `WP_Query` class for applications

The default `WP_Query` class works similarly for all the 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 those 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 when 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:

```
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 a 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 will play a vital part in our portfolio application development. In the following chapters, we will 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](http://codex.wordpress.org/Plugin_API/Filter_Reference#WP_Query_Filters).

## Introduction to WordPress query classes

WordPress provides a set of classes for querying the database in an object-oriented manner. These classes make it easier to access and understand the queries, compared to procedural functions. In the earlier section, we discussed more details about one of the query classes called `WP_Query`. This class is the most frequently used one among all the query classes. In this section, we will explore the functionality of the remaining query classes.

## The WP\_User\_Query class

The `WP_User_Query` class is used to query user-related data from WordPress database. Basically, this class uses the `wp_users` and the `wp_usermeta` tables for its queries. This is the second-most used class after `WP_Query`. Let's take a look at the basic usage of this class using the following code:

```
$user_query = new WP_User_Query( array( 'role' => 'Administrator'
    ) );
if ( ! empty( $user_query->results ) ) {
    foreach ( $user_query->results as $user ) {
        // display user details
    }
}
```

The `WP_User_Query` class takes an array of arguments for filtering users based on various criteria. In this scenario, we have filtered users with an administrator role. The following are some of the filtering methods that we can use on the `WP_User_Query` class:

- Get users by role
- Get users from a certain blog in multisite scenarios
- Get users based on a keyword search
- Get users with specific custom field and field value from the `wp_usermeta` table



More details about the use of the `WP_User_Query` class is provided at [http://codex.wordpress.org/Class\\_Reference/WP\\_User\\_Query](http://codex.wordpress.org/Class_Reference/WP_User_Query).

## The WP\_Comment\_Query class

The `WP_Comment_Query` class works with the `wp_comments` and `wp_commentmeta` tables for retrieving WordPress post comments-related data. This class is used in some of the themes for providing custom comments-related features. However, this is less frequently used compared to the `WP_Query` and the `WP_User_Query` classes. The following code shows the default usage of this class:

```
$comments_query = new WP_Comment_Query;
$comments = $comments_query->query( $args );
if ( $comments ) {
```

```
foreach ( $comments as $comment ) {  
    // display comments  
}  
}
```

You can use this class to retrieve comments of a specific user, specific post, comments with certain status, and many other parameters.



A complete guide to using this class can be accessed at [http://codex.wordpress.org/Class\\_Reference/WP\\_Comment\\_Query](http://codex.wordpress.org/Class_Reference/WP_Comment_Query).

## Other query classes

Apart from these main query classes, there are several other query classes in WordPress. Most of these classes are not needed or used frequently by developers. However, these classes are widely used within the WordPress core and work with the main query classes. The following are some of the other query classes available in WordPress:

- `WP_Meta_Query`: This class is used to generate the necessary SQL for meta-related queries
- `WP_Tax_Query`: This is a container class for multiple taxonomy queries
- `WP_Date_Query`: This class is used to generate the MySQL `WHERE` clause for the specified date-based parameters

We discussed the query classes in WordPress and their usage in brief. As a developer, you should be looking for opportunities to use these classes in custom plugin and theme development to understand the various parameters.

## Limitations and considerations

We have less flexibility with the WordPress built-in database compared to 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, which need to be executed inside a single process. We have to make sure that either all queries get executed successfully or none of them gets executed, to keep 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 upwards uses InnoDB as the table engine, and hence, we have the possibility of implementing transaction support. However, WordPress doesn't offer any library or functions for handling transactions, and hence, all 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, the importance of post revisions is restricted to the fields such as post title, content, author, and excerpt.

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 will span across multiple days, so you can create an event and use the post content to include the activities of the first day. Then, from the 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 in each day. Therefore, 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 );
```

## Auto saving

Auto saving is another feature that goes in combination with post revisions. Auto saving will create a different type of post revision at 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 auto saving without editing the core files. Therefore, we need to extend the interval of auto saving by defining a large value for `AUTOSAVE_INTERVAL` constant inside the `wp-config.php` file:

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

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

## Using meta tables

The WordPress table structure gives higher priority to meta tables for keeping additional data as key-value pairs. Although meta tables work well in most scenarios, this 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 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 that you follow the resource section for this chapter in the official book website at <http://www.innovativephp.com/wordpress-web-applications> for more resources and tutorials 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 with 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

## 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 beginner developers, who are used to working with simple websites every day. 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, and hence, named this chapter as *Building Blocks of Web Applications*.

While exploring the advanced use cases of custom post type implementations, we will 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 application
- Implementing the custom post type settings
- Validating post creation
- Building a simple template loader
- Introduction to custom post type relationships
- 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 type" misleads some people to think of them as different types of normal posts. In reality, these post types can model almost anything in real web applications. This is similar to collections in other web frameworks such as Ruby on Rails or Meteor.js. 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. The inability to manage different post types in their own lists and the inability to add different fields to different posts are some of the limitations with the old process. With the introduction of custom post types, we now have the ability to separate each different type of the post type to act as a model to cater to complex requirements. The demand for using these custom post types to build complex applications is increasing every day. 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 number 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 is preferable to make use of custom post types without developing everything from scratch.

Once a custom post type is registered, you will automatically have the ability to execute create, read, update, and 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 do need the ability 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. The rest of this chapter will mainly focus on handling these custom data in different ways to suit complex applications.

## Planning custom post types for application

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

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

### Projects

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

The screenshot shows a project creation form with the following fields and sections:

- Project Name:** A text input field.
- Description:** A large text area.
- Project Status:** A dropdown menu with "Selection..." as the current value.
- Project Duration:** A text input field.
- Download URL:** A text input field.
- URL:** A text input field.
- Screenshots:** A section with an "Add File" button.
- Technologies:** A list of three items: "Item1", "Item2", and "Item3".
- Project Type:** A list of three items: "Item1", "Item2", and "Item3".

- **Project Name:** This can be matched as the `title` field of a custom post type
- **Description:** This can be matched as the `editor` field of a custom post type
- **Technologies:** This can be matched as custom taxonomies
- **Project Type:** This can be matched as another custom taxonomy
- **URL:** This can be matched to a custom text field
- **Screenshots:** This can be matched as custom fields
- **Project Duration:** This can be matched as a custom text field
- **Download URL:** This can be matched as a custom text field
- **Project Status:** This can be matched as a custom dropdown field

## Services

Generally, people who work for someone or a company don't offer services. However, 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 will model the services using custom post types. Now, let's look at the following screenshot for necessary data requirements and their WordPress specific matches:

The screenshot shows a form for creating a service. It includes the following fields and components:

- Service Title:** A text input field.
- Description:** A large text area for the service description.
- Service Availability:** A dropdown menu with a "Selection..." placeholder.
- Service Price Type:** A dropdown menu with a "Selection..." placeholder.
- Price:** A text input field.
- Tasks:** A list box containing three items: "Item1", "Item2", and "Item3".

Let's have a look at the components in the screenshot:

- **Service Title:** This can be matched as the title field of a custom post type
- **Description:** This can be matched as the editor field of a custom post type

- **Tasks:** This can be matched as custom taxonomies
- **Service Price Type:** This can be matched to a custom dropdown field
- **Price:** This can be matched to a custom text field or dropdown field
- **Service Availability:** This can be matched to a custom dropdown field

## Articles

The **Articles** section contains articles, tutorials, and 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. Since every website will need a blog at some point in its life cycle, we will skip normal posts and create separate custom post types for articles.



We will not discuss articles in detail as it contain similar fields such as Title, Summary, URL, Categories, Screens, and so on.

## Books

Compared to other sections, **Books** will have less impact and fewer data as most developers and designers are not authors. However, writing books on your preferred technology is a great way to enhance your reputation and build a name online. Similar to **Articles**, **Books** will have a generic set of fields such as:

- Title
- Summary
- URL
- Download
- Categories
- Screens

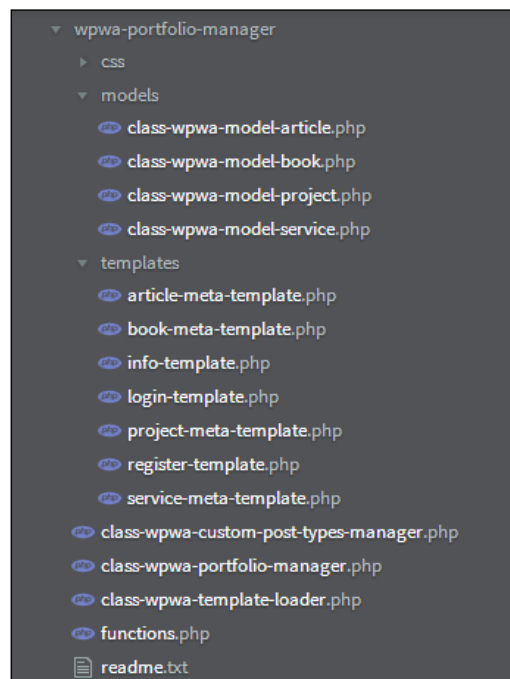
I hope you have a clear understanding about the things we will implement in this chapter. We will start by implementing these custom post types using a plugin. While implementing these models, we will also take a look at some advanced techniques for template management, validations, and post relationships.



## Implementing custom post types for a portfolio application

In this section, we will extend the plugin developed in *Chapter 3, Planning and Customizing Core Database*, and implement the custom post type-related functionality. First, we have to create a new file called `class-wpwa-custom-post-types-manager.php` inside the root directory of the `wpwa-portfolio-manager` plugin.

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 thing 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 updated folder structure of the plugin using the following screenshot:



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

- `class-wpwa-custom-post-manager.php`: This includes the main initialization and configuration code for models

- `class-wpwa-template-loader.php`: This includes a template file loading and initialization code
- `models`: This folder contains all the custom post type-specific classes
- `templates`: This folder contains all the HTML templates required for the plugin

All the custom post type-specific classes located inside the `models` folder need to be included in 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 bunch of the `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 the concept called autoloading files for such projects. So, we will implement an autoloader for our post type classes inside the `models` folder. Let me explain how 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 **class name**. However, there can be two major problems with the default technique:

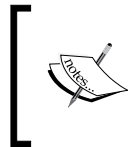
- We cannot have classes inside subfolders
- This function will load each and every class inside the application, even the classes we don't want to be included

Basically, this means we have to look for an alternative implementation of the autoloader to suit our plugins. We will use a predefined format for our model classes to solve this issue. All the model classes will be named as `WPWA_Model_{post_type_name}`. For example, the projects class will be named as `WPWA_Model_Project`, while the filename will be `class-wpwa-model-project.php` so that we can look for a project file inside the `models` folder. Having the solution in mind, we will implement our autoloader function inside the `class-wpwa-portfolio-manager.php` file, as 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 need to 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 the `$class_directory` variable to find out 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 will 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 file and 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 for custom post types. Let's create a class called `WPWA_Custom_Post_Manager` in the `class-wpwa-custom-post-manager.php` file 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 a 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 keeps the path to the plugin folder from your document root.
- `template_parser`: In this, we will handle templates using a template loader. So, this variable will hold an instance of the template loader.
- `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 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 into its own class. Therefore, all the post type-specific implementations will be inside those classes, meaning that the responsibility of constructor will be limited to the instantiation of those classes. Let's see how a main file constructor looks like:

```
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 a 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 will look at the detailed implementation of the `Project` class. Other custom post types are similar, and hence, you can find them inside the source code folder. Thus 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 implementations of those classes.

## Creating the project class

We will choose the `Project` class as it is the most complex one 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() {
        global $wpwa_template_loader;
        $this->template_parser = $wpwa_template_loader;
        // Initialization code goes here
    }
}
```

Here, we also have two instance variables for keeping the custom post type name and template loader 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 post type creation:

```
class WPWA_Model_Project {
    global $wpwa_template_loader;
    // Our existing instance variables
    public function __construct() {
        $this->template_parser = $wpwa_template_loader;
        $this->post_type = "wpwa_project";
        add_action('init', array($this, 'create_projects_post_type'));
    }
}
```

First, we will assign the name of the post type to the instance variable so that we can use it for registering the post type. With 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 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.

We use the WordPress `init` action to call the `create_projects_post_type` function for registering new custom post types. Let's look at the implementation of this function:

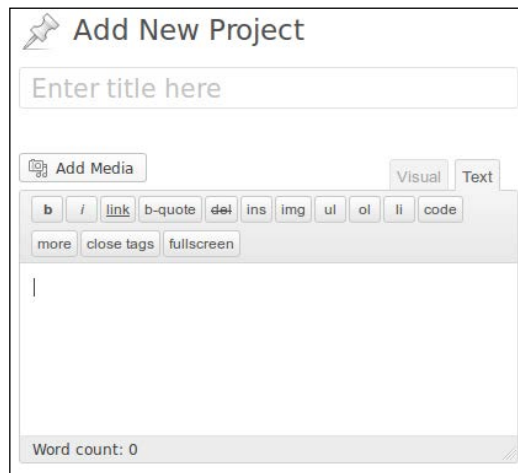
```
public function create_projects_post_type() {
    global $wpwa_custom_post_types_manager;
    $params = array();
    $params['post_type'] = $this->post_type;
    $params['singular_post_name'] = __('Project', 'wpwa');
    $params['plural_post_name'] = __('Projects', 'wpwa');
    $params['description'] = __('Projects', 'wpwa');
    $params['supported_fields'] = array('title', 'editor');
    $wpwa_custom_post_types_manager-
    >create_post_type($params);
}
```

You might have noticed that familiar custom post type creation code is missing here. We have to define the common settings and all the labels for each and every custom post type. Since we are planning to use multiple custom post types for our project, we have to prevent code duplication. Therefore, we will be implementing the custom post creation in a common function and passing the necessary parameters from the individual models. In this code, we have defined necessary labels and supported fields to be passed to the common `create_post_type` function inside the `WPWA_Custom_Post_Types_Manager` class. Now, we can look at the implementation of the `create_post_type` function for all our models:

```
public function create_post_type($params) {
    extract($params);
    $labels = array(
        'name' => sprintf( __( '%s', 'wpwa' ),
            $plural_post_name),
```

```
'singular_name' => sprintf( __( '%s', 'wpwa' ),
$singular_post_name),
'add_new' => __( 'Add New', 'wpwa' ),
'add_new_item' => sprintf( __( 'Add New %s ', 'wpwa'
), $singular_post_name),
'edit_item' => sprintf( __( 'Edit %s ', 'wpwa' ),
$singular_post_name),
'new_item' => sprintf( __( 'New %s ', 'wpwa' ),
$singular_post_name),
'all_items' => sprintf( __( 'All %s ', 'wpwa' ),
$plural_post_name),
'view_item' => sprintf( __( 'View %s ', 'wpwa' ),
$singular_post_name),
'search_items' => sprintf( __( 'Search %s ', 'wpwa'
), $plural_post_name),
'not_found' => sprintf( __( 'No %s
found', 'wpwa' ), $plural_post_name),
'not_found_in_trash' => sprintf( __( 'No %s found
in the Trash', 'wpwa' ), $plural_post_name),
'parent_item_colon' => '',
'menu_name' => sprintf( __( '%s', 'wpwa' ),
$plural_post_name),
);
$args = array(
'labels' => $labels,
'hierarchical' => true,
'description' => $description,
'supports' => $supported_fields,
'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( $post_type, $args );
}
```

There is nothing substantial to explain in the preceding code other than the use of dynamic variables instead of hardcoding the details. 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 projects creation screen, as shown in the following screenshot:



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

## Assigning permissions to projects

In general, a developer user role should be able to handle all the post types created in this application. Therefore, 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 `class-wpwa-portfolio-manager.php` file in the WPWA Portfolio 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. This function is executed within the plugin activation handler, and so you need to deactivate the plugin and activate it again to apply the new capabilities to the users. Once completed, you will be able to manage projects as developers.

Now, we have the necessary permissions and basic fields ready for creating a project title and description. The most important part of a web application 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 project type. So, let's get started on the implementation.

## Creating custom taxonomies for technologies and project 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 project types. Let's open the `class-wpwa-model-project.php` file and update the `Project` class constructor to implement the actions for taxonomy creation:

```
class WPWA_Model_Project {
    // Other Instance variables
    private $technology_taxonomy;
    private $project_type_taxonomy;
    public function __construct() {
        global $wpwa_template_loader;
        $this->template_parser = $wpwa_template_loader;
        $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 reused across all functions. Initialization of these variables is handled through the constructor. Next, we will 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, this function creates code duplication, and so we will be using a common function. First, we will be looking at the model-specific function for defining necessary data:

```

public function create_projects_custom_taxonomies() {
    global $wpwa_custom_post_types_manager;
    $params = array();
    $params['category_taxonomy'] = $this->technology_taxonomy;
    $params['post_type'] = $this->post_type;
    $params['singular_name'] = __('Technology', 'wpwa');
    $params['plural_name'] = __('Technology', 'wpwa');
    $wpwa_custom_post_types_manager->
create_custom_taxonomies($params);
    $params['category_taxonomy'] = $this->project_type_taxonomy;
    $params['post_type'] = $this->post_type;
    $params['singular_name'] = __('Project Type', 'wpwa');
    $params['plural_name'] = __('Project Type', 'wpwa');
    $params['capabilities'] = array(
        'manage_terms' => 'manage_project_type',
        'edit_terms'   => 'edit_project_type',
        'delete_terms' => 'delete_project_type',
        'assign_terms' => 'assign_project_type'
    );
    $wpwa_custom_post_types_manager->
create_custom_taxonomies($params);
}

```

We are creating two custom taxonomies for projects, and so we have called the common `create_custom_taxonomies` function twice with the necessary parameters. The following code previews the common `create_custom_taxonomies` function:

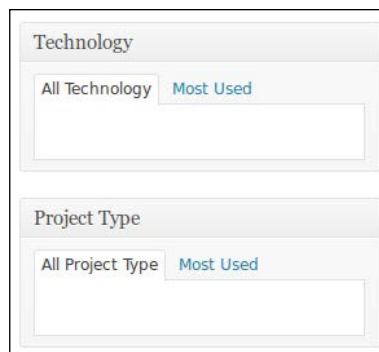
```

public function create_custom_taxonomies($params) {
    extract($params);
    $capabilities = isset($capabilities) ? $capabilities : array();
    register_taxonomy(
        $category_taxonomy,
        $post_type,
        $capabilities
    );
}

```

```
array(
    'labels' => array(
        'name' => sprintf( __( '%s Category',
'wpwa' ) , $singular_name),
        'singular_name' => sprintf( __( '%s Category',
'wpwa' ) , $singular_name),
        'search_items' => sprintf( __( 'Search %s
Category', 'wpwa' ) , $singular_name),
        'all_items' => sprintf( __( 'All
%s Category', 'wpwa' ) , $singular_name),
        'parent_item' => sprintf( __(
'Parent %s Category', 'wpwa' ) , $singular_name),
        'parent_item_colon' => sprintf( __(
'Parent %s Category:', 'wpwa' ) , $singular_name),
        'edit_item' => sprintf( __( 'Edit
%s Category', 'wpwa' ) , $singular_name),
        'update_item' => sprintf( __(
'Update %s Category', 'wpwa' ) , $singular_name),
        'add_new_item' => sprintf( __( 'Add
New %s Category', 'wpwa' ) , $singular_name),
        'new_item_name' => sprintf( __( 'New
%s Category Name', 'wpwa' ) , $singular_name),
        'menu_name' => sprintf( __( '%s
Category', 'wpwa' ) , $singular_name),
    ),
    'hierarchical' => true,
    'capabilities' => $capabilities ,
)
);
}
```

Before moving into code explanation, I would like you to refresh the projects creation area to see 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 custom taxonomy creation function with all the necessary options. There is nothing new in technology taxonomy other than the use of instance variables instead of hardcoding. Astute readers might notice the difference in project type implementation. We have added a section called `capabilities` to the project type. In today's world, web-based technologies change rapidly. So, we need to provide the ability for developers to define any new technology in our application. On the other hand, project types are fixed and won't get changed regularly. Therefore, we need to block the project type creation for user roles other than the admin.

By default, WordPress uses the `manage_categories` permission for all the taxonomies, including 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 the project type:

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

The following four permissions called are used to handle default permissions:

- `manage_terms`
- `edit_terms`
- `delete_terms`
- `assign_terms`

Here, we need to handle the permissions of the project type separately from others, and hence, we have assigned four custom permission types to respective keys. Now, take a look at the project creation menu on 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 permission until we assign them.

## Assigning permissions to the project type

We added custom capabilities in the project type creation process. However, WordPress has 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 go to **Users | Roles | Administrator** to see all the available capabilities. You won't see the new capabilities in this screen.

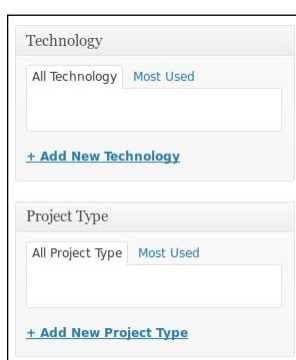
Now, open the `class-wpwa-portfolio-manager.php` file again in the WPWA Portfolio Manager plugin with the updated code in the preceding section on project permissions. Navigate to the `add_application_user_capabilities` function and change the existing code as follows:

```
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);
    }
}
```



You can get more details about roles and capabilities from the WordPress codex at [http://codex.wordpress.org/Roles\\_and\\_Capabilities](http://codex.wordpress.org/Roles_and_Capabilities).

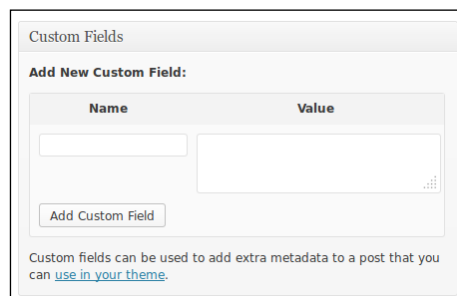
By now, you should know that capabilities cannot be defined without a user role. Therefore, we get the administrator role as an object. Throughout this application, we will specify all the custom capabilities into an array called `custom_admin_capabilities`. Finally, we add each of the custom capabilities 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 go to the **Users | Roles | Administrator** section to view the capabilities. Now, you should be able to see the custom capabilities assigned to admin role. Also, you will have access to the **Add New Project Type** link in the project creation screen, as shown in the following screenshot:



So far, we have created the default 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 is provided with default posts as well. We can enable custom fields on posts by clicking the **Screen Options** menu on the top of the post creation screen and ticking the checkbox of the **Custom Fields**. 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, so the default custom fields screen is not ideal for web applications.

Instead, we can use the same custom fields in different approaches 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 meta box creation for 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 metafields. They are as follows::

- The first parameter defines a unique key to the meta box (the HTML `id` attribute of the screen)
- The second and third parameters define meta box title and function respectively
- The fourth parameter defines the associated post type; in this case, the parameter will be `Projects`



Here, I have mentioned the required parameters. In case you want more advanced configurations with optional parameters, look at the documentation at [http://codex.wordpress.org/Function\\_Reference/add\\_meta\\_box](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 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 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:

- 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
- Difficulty in debugging the codes

In web applications, we need to separate layers based on their functionality. Therefore, we have to keep logic away from the presentation 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?

The 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 shouldn't have complex code inside template files other than simple if-else statements and loops.



There are plenty of open source template engines available for PHP. Smarty, Mustache, and Twig are some of the popular ones among them. However, integrating this type of template engine in WordPress is a complex task, compared to using it in other PHP frameworks. The architecture of WordPress is different from any other PHP framework as it drives on action hooks and filters. Therefore, the integration needs to be capable of handling WordPress-specific things such as actions, filters, widgets, template tags, and so on.



The Twig templates engine created by SensioLabs is one of my favorites as it offers many unique features compared to the other template engines. If you are not familiar with template engines, I suggest that you look at the Twig documentation at <http://twig.sensiolabs.org/documentation>. As I mentioned, the scope of integrating Twig with WordPress is beyond the scope of this book, and hence will be discussed in the official website for this book. We will be using the Timber plugin for WordPress at <https://wordpress.org/plugins/timber-library/>. Until then, you can try out the plugin, which offers the integration of Twig templates.

Integrating WordPress plugins with the afore mentioned template engines is not so popular among developers; even the most popular and well-coded plugins use a simple template loading techniques. Throughout this book, we will be using our own template loading technique. Keep in mind that this technique is nowhere close to the features offered by a template engine. Also, this technique partially separates the presentation layer from logic, compared to template engines, which completely separate these two concerns. Let's get started with creating a simple template loading technique used by many developers.

## Building a simple custom template loader

As developers, you might be familiar with the WordPress template loading technique using the `get_template_part` function. Basically, this function includes a file behind the scenes. This is similar to executing a `require` or `include` function on a file. However, WordPress recommends using this function instead of manual `require` or `include` function calls. We will use the same technique for building our template loader. This is the most commonly used technique among many popular plugins. Let's start by creating a new file called `class-wpwa-template-loader.php` inside our `plugins/wpwa-portfolio-manager` folder with the following class definition:

```
class WPWA_Template_Loader{
    // Template loading code
}
$wpwa_template_loader = new WPWA_Template_Loader();
```


Now, let's look at the implementation of our `get_template_part` function similar to WordPress core function:

```
public function get_template_part($slug,$name= null,$load= true) {
    do_action( 'wpwa_get_template_part_' . $slug, $slug, $name );
    // Setup possible parts
    $templates = array();
    if ( isset( $name ) )
        $templates[] = $slug . '-' . $name . '-template.php';
    $templates[] = $slug . '-template.php';
    // Allow template parts to be filtered
    $templates = apply_filters( 'wpwa_get_template_part',
    $templates, $slug, $name );
    // Return the part that is found
    return $this->locate_template( $templates, $load, false );
}
```

The WordPress `get_template_part` function has three parameters with `$slug` being the only required parameter. You can define the template name in two ways, as follows:

- **Templates without parts:** These are standalone templates and so only a slug will be available for these templates. For example, `projects.php` where `projects` will be the slug without any subtemplate.
- **Templates with parts:** These are partial templates and so both slug and name will be available in a template name. This is useful when having multiple subtemplates for a specific section, for example, `project-tasks.php` and `project-members.php`. The `project` will be the main slug and `tasks` and `members` will be the templates.

We will check the availability of sub or main template with the given filename and assign it to an array called `$templates`.

 Note that we have an action called `wpwa_get_template_part` and a filter called `wpwa_get_template_part`. These hooks can be used to change the templates array based on different conditions and execute custom code when specific templates are loaded.

Finally, we will call the `locate_template` function for loading the templates inside the `$templates` array. Let's take a look at the implementation of the `locate_template` function:

```
public function locate_template( $template_names, $load = false,
    $require_once = true ) {
```

```
// No file found yet
$located = false;
// Traverse through template files
foreach ( (array) $template_names as $template_name ) {
    // Continue if template is empty
    if ( empty( $template_name ) )
        continue;
    $template_name = ltrim( $template_name, '/' );
    // Check templates for frontend section
    if ( file_exists( trailingslashit( wpwa_path ) . 'templates/'
. $template_name ) ) {
        $located = trailingslashit( wpwa_path ) . 'templates/' .
$template_name;
        break;
        // Check templates for admin section
    }
}
if ( ( true == $load ) && ! empty( $located ) )
    load_template( $located, $require_once );
return $located;
}
```

We traverse all the templates in the `$templates` array and look for the existence of a template inside the specified template folders. Here, we are only searching the `templates` folder. As the project gets larger, we will have to define multiple sublevels inside the `templates` folder or multiple template folders. In such scenarios, we have to extend this code to include multiple locations for searching templates. Once a template is found, we break the process and return the template file.

Now, we have a basic template loader. In the next section, we will look at how to use this template loader to load the necessary templates.

## Creating your first template

Template engines allow us to use pure HTML files or any other file type for templates. However, we are only using a template loader and therefore we need some PHP code inside the templates. So, let's create the first template file called `project-meta-template.php` inside the `templates` folder for project metadata. Here, we have the complete template code for the project meta box:

```
<?php
    global $template_data;
    extract( $template_data );
?>
```

---

```

<input type="hidden" name="project_meta_nonce" value="<?php echo
$project_meta_nonce; ?>" />
<table class="form-table">
  <tr>
    <th style=''><label for='<?php echo $project_status_label;
?>'><?php echo $project_status_label; ?> *</label></th>
    <td>
      <select class='widefat' name="sel_project_status"
id="sel_project_status">
        <option <?php selected( $project_status, 0 ); ?>
value="0">Select</option>
        <option <?php selected( $project_status, 'planned'
); ?> value="planned">Planned</option>
        <option <?php selected( $project_status, 'pending'
); ?> value="pending">Pending</option>
        <option <?php selected( $project_status, 'failed'
); ?> value="failed">Failed</option>
        <option <?php selected( $project_status,
'completed' ); ?> value="completed">Completed</option>
      </select>
    </td>
  </tr>
  <tr>
    <th style=''><label for='<?php echo
$project_duration_label; ?>'><?php echo $project_duration_label;
?> *</label></th>
    <td><input class='widefat' name='txt_duration'
id='txt_duration' type='text' value='<?php echo $project_duration;
?>' /></td>
  </tr>
  <tr>
    <th style=''><label for='<?php echo $project_url_label;
?>'><?php echo $project_url_label; ?></label></th>
    <td>
      <input class='widefat' name='txt_url' id='txt_url'
type='text' value='<?php echo $project_url; ?>' /></td>
  </tr>
  <tr>
    <th style=''><label for='<?php echo
$project_download_url_label; ?>'><?php echo
$project_download_url_label; ?></label></th>
    <td><input class='widefat' name='txt_download_url'
id='txt_download_url' type='text' value='<?php echo
$project_download_url; ?>' /></td>
  </tr>
</table>

```

Let's get started!

1. First, we will use a global variable called `$template_data` to access the data for templates. We can start the template by defining the nonce value inside a hidden field for securing form submission. In this technique, we are using PHP variables to assign data to templates.
2. Then, we have a list of fields for **Project URL**, **Download URL**, **Project Status**, and **Project Duration**. Each of these fields uses a PHP variable from the global `$template_data` variable for displaying data.



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

3. The next task will be to assign the created template into a 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 our new template loader:

```
public function display_projects_meta_boxes() {
    global $post,$template_data;
    $data = array();
    // Get the existing values from database
    $template_data['project_meta_nonce'] =
wp_create_nonce('wpwa-project-meta');
    $template_data['project_url'] =
esc_url(get_post_meta( $post->ID, '_wpwa_project_url', true
));
    $template_data['project_duration'] =
esc_attr(get_post_meta( $post->ID,
'_wpwa_project_duration', true ));
    $template_data['project_download_url'] =
esc_attr(get_post_meta( $post->ID,
'_wpwa_project_download_url', true ));
    $template_data['project_status'] =
esc_attr(get_post_meta( $post->ID, '_wpwa_project_status',
true ));
    $template_data['project_status_label'] = __('Project
Status','wpwa');
    $template_data['project_duration_label'] = __('Project
Duration','wpwa');
```

```

$template_data['project_url_label'] = __('Project
URL', 'wpwa');
$template_data['project_download_url_label'] =
__('Download URL', 'wpwa');
ob_start();
$this->template_parser->get_template_part(
'project', 'meta');
$display = ob_get_clean();
echo $display;
}

```

First we have to get the existing data from the database and assign them to the global `$template_data` variable. The template will only have access to the data specified in this variable. Most of the variables contain data required for the project screen. However, `project_meta_nonce` might be new to some of you. WordPress uses nonce value generation for securing and validating form submissions. Therefore, we have assigned the nonce value to the data array with a key called `project_meta_nonce`.



Nonce is used for security purposes to protect against unexpected or duplicate requests that can cause undesired permanent or irreversible changes to the website and particularly to its database. Specifically, a nonce is a 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 sent along 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.

Many developers may not be familiar with this technique, and so let's look at a detailed explanation of the template loading process. As mentioned earlier, the template loader will load the templates using the PHP `require` or `include` statements. Therefore, the output of template cannot be assigned to a variable, and hence, we need to buffer the output. We will use the `ob_start` function for output buffering. The following is the definition of the `ob_start` function provided by the PHP site:

*"This function will turn output buffering on. While output buffering is active no output is sent from the script (other than headers), instead the output is stored in an internal buffer."*

4. Then, we will use the `$wpwa_template_loader` object assigned in the constructor to load the template using our new template loader. Therefore, the content of the template file will be stored in an internal buffer.

5. Finally, we will execute the `ob_get_clean` function to get the current buffer contents and delete the current output buffer. Now, all the contents of the template file will be assigned to the variable and we can easily output it with an echo statement.

By now, you will have the complete project creation screen with default fields, taxonomies, and custom fields as shown in the following screenshot:

We have managed to create a template loader to separate logic from presentation. However, our template loader is far from completely separating the logic from presentation. In the next section, we will look at the functionality of our template loader and how it differentiates from a perfect template engine.

## Comparing the template loader and template engine

We started building a template loader to solve the issue with a mixing template code inside PHP variables. We managed to partially solve this problem. Now, the templates are stored as separate files with HTML content and we can reuse these templates from multiple `Model` classes. The `Model` class contains the logic and passes the data to templates. So, there is no more HTML content inside PHP variables. However, we still have PHP variables inside the template files to output model data. Let's see how our template loader differentiates from a proper template engine:

- Many template engines compile templates down to plain optimized PHP code and provide cached versions if necessary
- Handles automatic output escaping

- Uses specific syntax for defining data inside templates so that templates can be used without PHP
- Templates can be reused as blocks using inheritance

Considering the preceding points, it is obvious that our template loader only provides basic features for separating logic from the presentation layer. We will discuss how to integrate a proper template engine into WordPress on the official book website.

## 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 metatables. As usual, we have to update the constructor to add the 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 action `save_post` inside the constructor will only be called once the post is saved to the database with 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 a custom field 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 autosaving process. Finally, we have to verify the post type and check whether the current user has 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 reused across multiple locations.

Once all the validations are successfully completed, we will implement the data saving process inside the `class-wpwa-model-project.php` file, 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/>') . $this->error_message;
        set_transient($this->post_type."_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 to now. So, let's break the code into three sections, to simplify the explanation process:

- **Section 1:** The initial code includes the retrieval of form values through the `$_POST` array to be stored in variables. Here, we have to validate and filter the POST data to avoid harmful data. Therefore, we have used `trim` and WordPress escape functions to filter the data. Finally, we will cast the data into proper data type in order to prevent invalid data submissions.
- **Section 2:** In this section, we will implement the form validations for each and every form field. Once 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.



If 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. Therefore, we have chosen an alternative and a poor way to handle 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 is preferable to work with published data unless you are implementing custom application-specific status. Hence, we will set the post to draft status upon validation failure. Then, we can update the post to the database and add the `save_post` action back. Even though the post is still saved, we won't see it in application frontend as we are only focusing on published data. Once the user submits the form without errors, it will revert back to the publish status.

Once a project is successfully updated, WordPress will display the error as **Post draft updated**. Definitely, we need much more user-friendly errors to suit our applications. Therefore, we have to change the existing error messages generated by WordPress. Before this, 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 hooks and actions based procedure, it's not possible to get the error message after submission. Therefore, 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 custom post saving function, we can revert back 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 the 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 alternating the existing messages to suit our requirements. This is another section with duplicate codes, and hence, we will be using a common function to display the messages for all post types. First, we have to look at the model-specific messages function for defining necessary labels and data, as shown in the following code:

```
public function generate_project_messages( $messages ) {
    global $wpwa_custom_post_types_manager;
    $params = array();
    $params['post_type'] = $this->post_type;
    $params['singular_name'] = __('Project', 'wpwa');
    $params['plural_name'] = __('Projects', 'wpwa');
    $messages = $wpwa_custom_post_types_manager-
>generate_messages($messages, $params);
    return $messages;
}
```

As usual, we will define the necessary data and execute the common function. Implementation of `generate_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_messages` function inside the `WPWA_Custom_Post_Types_Manager` class:

```
public function generate_messages( $messages, $params ) {
    global $post, $post_ID;
    extract($params);
    // Get the temporary error message from database and WordPress
    generated
    // error no
    $this->error_message = get_transient(
    $post_type."_error_message_$post->ID" );
    $message_no = isset($_GET['message']) ? (int) $_GET['message'] :
    '0';
    // Remove the temporary error message from database
    delete_transient( $post_type."_error_message_$post->ID" );
```

```
if ( !empty( $this->error_message ) ) {
    //Override the default WordPress generated message
    //with our own custom message
    $messages[$post_type] = array( "$message_no" => $this-
    >error_message );
} else {
    // Customize the messages list
    $messages[$post_type] = array(
        0 => '', // Unused. Messages start at index 1.
        1 => sprintf(__('%1$s updated. <a href="%2$s">View
        %3$s</a>', 'wpwa' ),$singular_name,
        esc_url(get_permalink($post_ID)),singular_name),
        2 => __('Custom field updated.', 'wpwa' ),
        3 => __('Custom field deleted.', 'wpwa' ),
        4 => sprintf( __('%1$s updated.', 'wpwa' ),
        $singular_name),
        5 => isset($_GET['revision']) ? sprintf(__('%1$s restored
        to revision from %2$s', 'wpwa' ),$singular_name,
        wp_post_revision_title((int) $_GET['revision'], false)) : false,
        6 => sprintf(__('%1$s published. <a href="%2$s">View
        %3$s</a>', 'wpwa' ),$singular_name,
        esc_url(get_permalink($post_ID)),singular_name),
        7 => sprintf(__('%1$s saved.', 'wpwa' ),$singular_name),
        8 => sprintf(__('%1$s submitted. <a target="_blank"
        href="%2$s">Preview %3$s</a>', 'wpwa' ), $singular_name,
        esc_url(add_query_arg('preview', 'true',
        get_permalink($post_ID))), $singular_name),
        9 => sprintf(__('%1$s scheduled for: <strong>%2$s</strong>.
        <a target="_blank" href="%3$s">Preview %4$s</a>', 'wpwa' ),
        $singular_name,
        date_i18n(__('M j, Y @
        G:i'),strtotime($post->post_date)),
        esc_url(get_permalink($post_ID)),
        $singular_name),
        10 => sprintf(__('%1$s draft updated. <a
        target="_blank" href="%2$s">Preview %3$s</a>', 'wpwa' ),
        $singular_name, esc_url(add_query_arg('preview', 'true',
        get_permalink($post_ID))), $singular_name),
    );
}
return $messages;
}
```

WordPress uses the messages array with ten keys to cater all the messages generated in the custom post screens. Once the **Publish** button is clicked, we can 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 message as the key and 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 will get the error message using the `get_transient` function and the default message number using the `$_GET` array. After assigning the value to a variable, we will remove the transient using the `delete_transient` function to prevent unnecessary database load.

Next, we will check whether a specific error message exists in the database using the `get_transient` function. In case errors are generated, we will 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.

Now, we have completed the basic foundation for implementing our projects post type. Other post type creations are similar in nature, and so, I will not 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 that you change the code to understand the various aspects of custom post types.

## Introducing custom post type relationships

In general, we use relational databases in developing applications, where our models will be matched to separate database tables. Each model will be related to one or more other modules. However, in WordPress, we have all the custom post types stored in the posts table. Hence, 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 WordPress core development seems positive, we still don't have it on the core version as of 4.1. You can have a look at this 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 but to look for custom solutions built by external developers. There are a few competitive plugins among the Open Source community for providing post relationship functionality. The Posts 2 Posts plugin developed by Cristian Burca and Alex Ciobica seems to be the plugin of choice for 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 relationships, and so we will 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'));
```

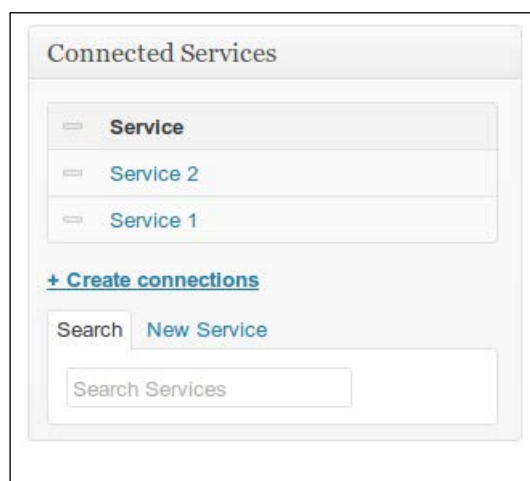
Here, we have the `p2p_init` action, which comes built-in with the Posts 2 Posts plugin. All the post relationship definitions and configurations go inside the `join_projects_to_services` function, as illustrated in the following code:

```
public function join_projects_to_services(){
    p2p_register_connection_type( array(
        'name' => 'projects_to_services',
        'from' => $this->post_type,
        'to' => 'wpwa_service'
    ) );
}
```

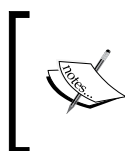
The Posts 2 Posts plugin provides a function called `p2p_register_connection_type` for defining post type relationships. The first parameter called `name` defines a unique identifier for the relationship. Then we can define two post types for the connection using `from` and `to` parameters. This will enable the services selection section for projects and project the 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 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 the following one with the associated services of a given project:



This is the most basic implementation of post relationships with this plugin.



If you are curious to learn about advanced usages, you can have a look at the plugin documentation at <https://github.com/scribu/wp-posts-to-posts/wiki>. In the following chapters, we will 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 into the system and create portfolio data for projects, services, books, and articles. Throughout 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 in the code to avoid duplicate 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, which is beyond the scope of this book. Therefore, I suggest that you look for existing open source libraries or take time to plan your own library. An alternative 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 rapidly. 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 has been becoming popular among developers in recent years. 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 does provide an extensive list of functionalities that are not available with default WordPress administrative screens.

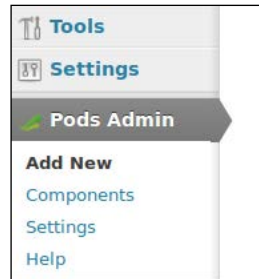
Let's consider some of the key features of the Pods framework, compared to manual implementation of custom post types:

- Create, extend, and manage custom types and fields, including custom post types, taxonomies, comments, and users
- Use default tables or custom tables for content types
- List of built-in form fields and components with necessary validations
- Manage form field-level permissions
- Create settings pages for plugins and themes

Basically, this framework provides out-of-the-box functionalities for common tasks in WordPress application development without much hassle. For example, think of how much effort you need to add a custom field for your comment form. With the Pods framework, you will be able to implement such tasks within a few clicks under 10 minutes. Pods does have an active community and hence can be recommended for rapid application development with WordPress.

Let's get our hands dirty by implementing something practical with this awesome framework.

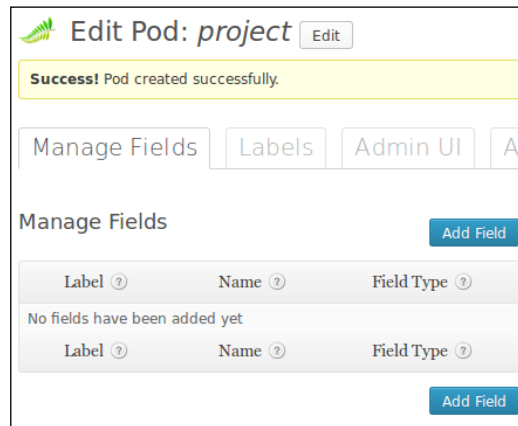
1. First, you have to grab a copy of this plugin from <http://pods.io> and get it installed on your WordPress folder.
2. Once activated, you should see the **Pods Admin** menu, as shown in the following screen:



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

A screenshot of the 'Create a New Content Type' form in the Pods Admin interface. The form is divided into two steps: '1 Create or Extend' (active, green) and '2 Configure' (inactive, grey). Below the steps, there is a text box explaining that creating a new Content Type allows control over its behavior, fields, and management. The form fields include 'Content Type' (a dropdown menu set to 'Custom Post Type (like Posts or Pages)'), 'Singular Label' (a text input field with 'Product'), and 'Plural Label' (a text input field with 'Products'). There is an 'Advanced +' link below the labels. At the bottom, there are 'Start Over' and 'Next Step' buttons.

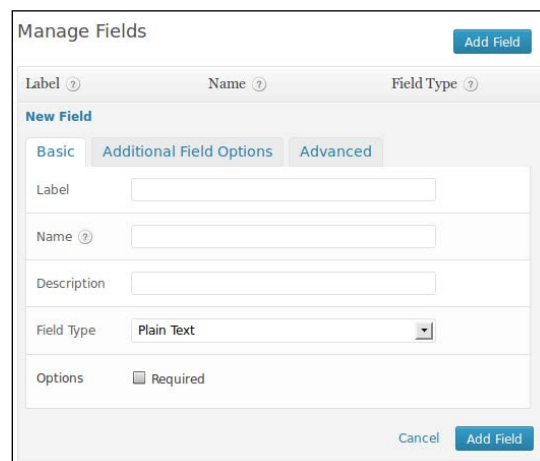
5. In the content type creation screen, you can select a custom post type and add the label as `Product`, since it's not wise to redefine projects.
6. Then, move on to the next step to get the screen shown in the following screenshot:



The screenshot shows the 'Edit Pod: project' interface. At the top, there's a green logo and the title 'Edit Pod: project' with an 'Edit' button. Below this is a yellow success message: 'Success! Pod created successfully.' The main area has three tabs: 'Manage Fields' (selected), 'Labels', and 'Admin UI'. Under the 'Manage Fields' tab, there's a section titled 'Manage Fields' with an 'Add Field' button. Below this is a table with columns 'Label', 'Name', and 'Field Type'. The table is currently empty, with the text 'No fields have been added yet' in the center. At the bottom right of the table area is another 'Add Field' button.

Surprisingly, we have the custom post ready in three clicks.

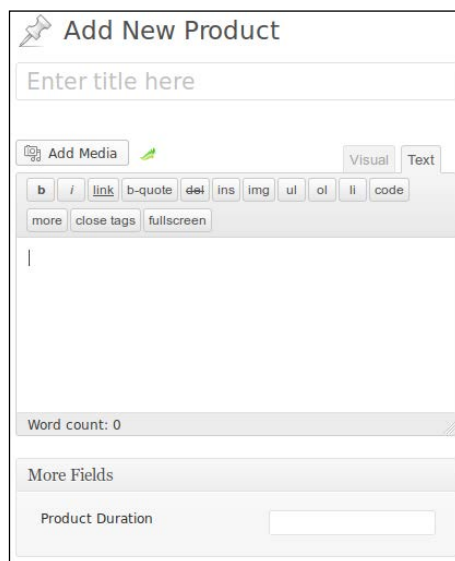
7. The next task is to add necessary custom fields and associate them with products post type. Remember the tasks we did in our manual custom field creation process. We had to create HTML for fields, use template engines, implement validations, and so on.
8. The Pods framework abstracts all these tasks behind the framework allowing us to focus on application logic. So, click the **Add Field** button to create the first custom field.



The screenshot shows the 'Manage Fields' interface with the 'Add Field' button clicked. The 'New Field' form is displayed, featuring three tabs: 'Basic' (selected), 'Additional Field Options', and 'Advanced'. The form includes input fields for 'Label', 'Name', and 'Description'. The 'Field Type' is set to 'Plain Text' in a dropdown menu. There is a checkbox for 'Required' which is currently unchecked. At the bottom right, there are 'Cancel' and 'Add Field' buttons.

All the necessary configurations are divided into three tabs at the top of the screen.

9. You can create the field by the defining necessary labels, validations, and access restrictions. Pods provides over fifteen built-in field types, including data pickers, color pickers, file fields, and text editors.
10. Once all the fields are created, click on the **Save Pod** button.
11. Now, navigate to the product creation screen and you will find all the custom fields created with Pods, as shown in the following screen:



We were able to create a complete custom post type with the necessary fields within 10 minutes. This is one of the most basic usages of the Pods framework. You can find more advanced and different usages on the official site at <http://podsframework.org/>.

## Should you choose Pods for web development?

One can definitely go with the Pods framework for applications that requires a rapid development process or low budgets. 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 and custom tables can become quite handy in complex web applications. We have to enable the **Advanced Content Types** component in order to make use of custom tables. It's recommended that you use existing tables in every possible scenario. However, there can be scenarios where custom tables can become 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 existing ones.

In short, you should be using this framework or similar ones 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 for interacting with custom content type-related functions, in order to maintain the quality of code and build maintainable systems.

There is a popular phrase, which states that "*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 a huge responsibility in 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 version upgrades
- Developers might discontinue the development of a plugin
- Plugins might not be updated regularly

That said, all 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 that you try out the following tasks in order to evaluate the things 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 can generate a new problem considering the possibility of extending. Find the issue and try to solve it by yourself.
- We enabled relationships between custom post types. Research for the possibilities of including meta data for relationships.

## Summary

In this chapter, we tackled the custom post types to learn the 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 a tough process for validating form data 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 will see how to use the WordPress plugin beyond the conventional use by implementing pluggable and extendable plugins. Until then, get your hands dirty by playing with custom post types plugins.



# 5

## Developing Pluggable Modules

Plugins are the heart of WordPress, which makes web applications possible. WordPress plugins are used to extend its 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.

Anyone who has basic programming knowledge can create plugins to meet application-specific requirements. However, it takes considerable effort to develop plugins that are reusable across a wide range of projects. In this chapter, we will build a few plugins to demonstrate the importance of reusability and extensibility. WordPress developers who don't have good experience in web application development shouldn't be skipping this chapter as plugins are the most important part of web application development.

I will assume that you have sound knowledge of basic plugin development using existing WordPress features in order to be comfortable 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 template loader with plugins
- Creating an extensible file uploading plugin

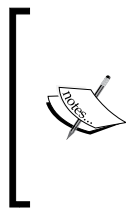


- Integrating a media uploader to custom fields
- Exploring the use of pluggable functions
- Time to practice

Let's get started.

## A brief introduction to WordPress plugins

WordPress offers one of the most flexible plugin architectures, alongside other similar frameworks such as Joomla and Drupal. The existence of over 35,000 plugins in the WordPress plugin directory proves the vital role of plugins. In typical websites, we create simple plugins to tweak the theme's functionalities or application-specific tasks. The complexity of web applications forces us to modularize the functionalities to enhance their maintainability. Most applications developer will be familiar with the concept of the **open-closed** principle.



The open-closed principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design of the code should be done in a way to allow the addition of new functionalities as new classes, keeping as much of the existing code unchanged as possible. You can find more information at <http://www.oodeesign.com/open-close-principle.html>.

We can easily achieve the open-closed principle with WordPress plugins. Plugins can be developed to open for new features through actions, filters, and pluggable functions while closed for modifications. Additional features will be implemented using separate plugins, which can be activated or deactivated anytime 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. Hence, you won't find detailed explanations about how plugins actually work behind the scenes. Plugins need to have a main file that includes the block comment in the predefined format, in order for WordPress to identify it as a plugin. The activation and deactivation of plugins can be done anytime by using the **Plugins** section of the admin area. WordPress uses a metafield called `active_plugins` in the `wp_options` table to keep the list of existing active plugins. The following screenshot previews the contents of the `active_plugins` field using the phpMyAdmin database browser:

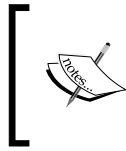
Field	Type	Value
option_id	bigint(20) unsigned	35
option_name	varchar(64)	active_plugins
option_value	longtext	<pre>a:4:{i:0;s:40:"Packt WPWA User Manager/user_manager.php";i:1;s:19:"members/members.php";i:2;s:13:"pods/init.php";i:3;s:48:"wpwa_custom_post_manager/custom_post_manager.php";}</pre>
autoload	varchar(20)	yes

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's functions. We can use built-in hooks as well as custom hooks inside plugins. As a WordPress plugin developer, it's important to understand 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](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 things using the existing features provided by WordPress. In short, all those web application-related features will be implemented using plugins. In this book, we have created a main plugin for portfolio management application. This plugin was intended to provide project specific tasks in our application and is hence not reusable in different projects. We need more and more reusable plugins and libraries as developers who are willing to take long journeys in web application development with WordPress. In this section, we will discuss the various types of such plugins:

- Reusable libraries
- Extensible plugins
- Pluggable plugins



Keep in mind that plugins are categorized into the previously mentioned types conceptually, for the sake of understanding various features of plugin development. Don't try to search using the preceding categories as they are not defined anywhere.

## Creating 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.

In *Chapter 4, Building Blocks of Web Applications*, we created a template loader to separate the main templates from its core logic. Loading templates is a common functionality for most WordPress plugins, and hence template loading is a "must-use" technique. Our template loader was created inside the WPWA Portfolio Manager plugin to manage templates for a user account and custom post type-related functionalities. However, it's not reusable across many plugins as it's located inside our main application plugin.

Usually, advanced WordPress applications are built using a combination of new and existing plugins to make different features independent from one another. Template loaders also fall into the category of independent and reusable modules, and hence, we need to convert our template loader to a reusable library using a plugin. In the following sections, we will discuss how to make it reusable and also some tricks to handle plugin dependencies.

## Planning the template loader plugin

The main purpose of building this independent plugin is to separate template loader functionalities and reuse it across many WordPress applications or plugins. However, there are other things that need to be considered when planning the plugin. Let's list down the main steps of building a template loader plugin:

1. Build a template loader as an independent standalone plugin.
2. Allow the reuse of the template loader from multiple plugins without modifying code.
3. Load the dependent plugins without errors.
4. Add extendable features to template loaders.

With these requirements in mind, let's build the template loader plugin. As usual, create a new folder called `wpwa-template-loader` inside the `wp-content/plugins` folder. Create the main plugin file called `wpwa-template-loader.php` with the following plugin definition:

```
<?php
/*
    Plugin Name: WPWA Template Loader
    Plugin URI:
    Description: Reusable template loader for WordPress plugins.
    Author: Rakhitha Nimesh
    Version: 1.0
    Author URI: http://www.innovativephp.com/
*/
define('wpwa_tmpl_url', plugin_dir_url(__FILE__));
define('wpwa_tmpl_path', plugin_dir_path(__FILE__));
?>
```

We already have a template loader inside the WPWA Portfolio Manager plugin. So, we will move the files into our new plugin with slight modifications. Consider the following code for the implementation of a template loader class:

```
class WPWA_Template_Loader{
    public $plugin_path;
    public function set_plugin_path($path){
        $this->plugin_path = $path;
    }
    public function get_template_part( $slug, $name = null, $load =
true ) {
        do_action( 'wpwa_get_template_part_' . $slug, $slug, $name );
        $templates = array();
        if ( isset( $name ) )
            $templates[] = $slug . '-' . $name . '-template.php';
        $templates[] = $slug . '-template.php';
        $templates = apply_filters( 'wpwa_get_template_part',
$templates, $slug, $name );
        return $this->locate_template( $templates, $load, false );
    }
    public function locate_template( $template_names, $load = false,
$require_once = true ) {
        $located = false;
        foreach ( (array) $template_names as $template_name ) {
            if ( empty( $template_name ) )
                continue;
```

```
        $template_name = ltrim( $template_name, '/' );
        if ( file_exists( trailingslashit( $this->plugin_path ) .
'templates/' . $template_name ) ) {
            $located = trailingslashit( $this->plugin_path ) .
'templates/' . $template_name;
            break;
        }
        elseif ( file_exists( trailingslashit( $this->plugin_path )
. 'admin/templates/' . $template_name ) ) {
            $located = trailingslashit( $this->plugin_path ) .
'admin/templates/' . $template_name;
            break;
        }
    }
    if ( ( true == $load ) && ! empty( $located ) )
        load_template( $located, $require_once );
    return $located;
}
}
$wpwa_template_loader = new WPWA_Template_Loader();
```

You might have noticed that we introduced a new class variable called `$plugin_path` and a new function called `set_plugin_path`. In the previous chapter, we used the template loader within the plugin, and hence, we were able to hardcode the path to the plugins folder to locate templates. Since we are planning to reuse it across multiple plugins, the path to templates should be specified dynamically. We can pass the plugin path dynamically using the `set_plugin_path` function and search the templates inside individual plugins. Now, it's time to use a reusable template loader inside our main plugin.

## Using the template loader plugin

First, we have to remove all the template loader-specific functionalities from our main WPWA Portfolio Manager plugin. Let's start by executing the following tasks:

1. Remove the `class-wpwa-template-loader.php` file from the main plugin.
2. Remove the `require_once` statement from the `class-wpwa-portfolio-manager.php` file.

Once these two tasks are completed, we are ready to use the template loader. You might have noticed that the only difference between the previous and current implementation of the template loader is the addition of a dynamic plugin path. So, the only change required in the main plugin is the template loader initialization.

We initialized the template loader inside the constructor of each model class using the following code:

```
public function __construct() {
    global $wpwa_template_loader;
    $this->template_parser = $wpwa_template_loader;
}
```

We can use the global `$wpwa_template_loader` object and assign it to the class variable called `template_parser` to load the templates. Now, we need to pass the plugin path of our main plugin as a parameter to set up the template loader locations. Defining the plugin path needs to be done only once, and hence, it's not ideal to implement it inside the constructor of each model. So, we will use a common initialization of the template loader inside the constructor of the `WPWA_Custom_Post_Types_Manager` class, as shown in the following code:

```
class WPWA_Custom_Post_Types_Manager {
    public function __construct() {
        global $wpwa_template_loader;
        $wpwa_template_loader->set_plugin_path(wpwa_path);
    }
}
```

We have set the plugin path in only one location and all the models will have access to the template loader with new settings, as we are using the global object. After these modifications, our main plugin should function as usual with the new reusable template loader. You can use the same technique to access a template loader from any other plugins.

## Handling plugin dependencies

In the previous section, I mentioned that the main plugin should work as usual. However, it won't work as expected and you will get an error with a blank screen, as shown in the following screenshot:

**Fatal error:** Call to a member function `set_plugin_path()` on a non-object in `/home/nimesh/example.com/wp-content/plugins/wpwa-portfolio-manager/class-wpwa-custom-post-types-manager.php` on line 16



You have to enable `WP_DEBUG` to see the errors. This can be done by setting `WP_DEBUG` to true in the `wp-config.php` file.

If you have a sound knowledge of WordPress development, you should have an idea about the cause of this error. I will explain the issue in detail and the solution for those who are not aware of the reason for this error.

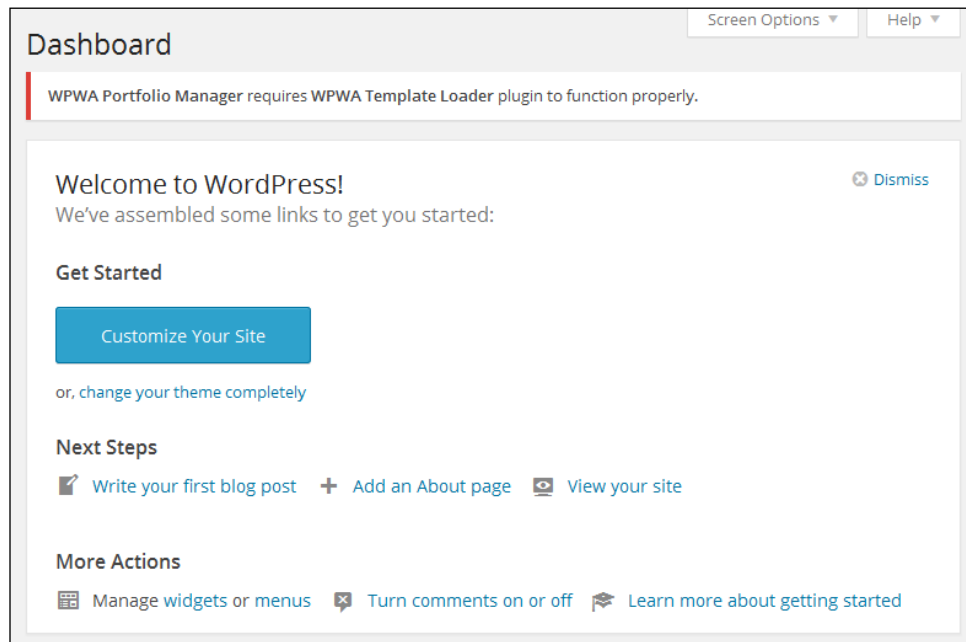
Basically, WordPress loads the plugin in a specific order. Dependent plugins should be loaded before the main plugin to prevent any dependency issues. In this scenario, we didn't handle the dependency between the WPWA Portfolio Manager plugin and the WPWA Template Loader plugin.

If you check the value `active_plugins` option, you might notice that `wpwa-portfolio-manager/class-wpwa-portfolio-manager.php` is stored before the `wpwa-template-loader/class-wpwa-template-loader.php` file. Therefore, the template loader object is not available when the features of WPWA Portfolio Manager are executed. We need a solution to delay the loading of the main plugin, until the dependent plugins are fully loaded. Let's update `class-wpwa-portfolio-manager.php` to define the necessary dependencies:

```
/* Validating existence of required plugins */
add_action( 'plugins_loaded', 'wpwa_plugin_init' );
function wpwa_plugin_init() {
    if(!class_exists('WPWA_Template_Loader')){
        add_action( 'admin_notices', 'wpwa_plugin_admin_notice' );
    }else{
        global $wpwa_custom_post_types_manager;
        $wpwa_custom_post_types_manager = new
        WPWA_Custom_Post_Types_Manager();
    }
}
function wpwa_plugin_admin_notice() {
    echo '<div class="error"><p><strong>WPWA Portfolio
Manager</strong> requires <strong>WPWA Template Loader</strong>
plugin to function properly.</p></div>';
}
```

First, we will use the `plugins_loaded` action to call a custom function called `wpwa_plugin_init`. This function is executed after all the plugins are loaded, and hence, all the plugin files are available for execution. The portfolio manager plugin is dependent on a template loader plugin, and hence, we check the existence of the `WPWA_Template_Loader` class. If the template loader plugin is activated and fully loaded, we initialize the `WPWA_Custom_Post_Types_Manager` class for the functionalities of our models. The initialization code on `class-wpwa-custom-post-types-manager.php` needs to be removed as well.

If the template loader plugin is not activated, we have to prevent the execution of dependent functionalities and inform the admin about the reason. Therefore, we add a notice to the admin section mentioning that you need to activate the template loader plugin before using our main plugin. The following screen shows the message displayed to the admin when the template loader plugin is not activated:



We completed our task on creating a reusable plugin using a template loader and identified how to solve the dependencies between plugins. We checked the existence of a class to validate whether the dependent plugin is active. There are a few other techniques for checking the active/inactive status of dependent plugins listed as follows:

- **Functions:** The existence of a function can be checked similar to the class existence check, as shown in the following code. However, we can only check procedural functions with this technique, and hence, this technique can't be used for functions inside classes (methods). Therefore, it's not possible to apply this technique to the template loader plugin:

```
if(! function_exists( 'function name' ) ) {
    // Plugin is inactive
}
```



- **Constants:** We can also check the existence of a constant within the dependent plugin. Since we do have constants, this technique can be used for our template loader. We have to use globally available constants such as the plugin version, plugin path, and so on for this validation, as shown in following code:

```
if ( ! defined( ' wpwa_tmpl_path ' ) ) {  
    // Plugin is inactive  
}
```

- **Directly checking the plugin status:** WordPress provides a function for providing the active/inactive status of a plugin. However, we have to pass the plugin folder and filename, and hence, it's not a reliable solution, unless you are confident that plugins or filenames won't change. The following code shows how to directly check the status of a plugin:

```
if ( is_plugin_active( ' wpwa-template-loader/ wpwa-  
template-loader.php' ) ) {  
    // Plugin is active  
}
```

Now, we have completed the development of a reusable plugin. Next, we have to make the new plugin extensible using actions and filters. The following section explains the process of creating extendable plugins, and hence, we will be completing the template loader functionality in the next section.

## Extensible plugins

In the previous section, we created a reusable plugin for template loading. However, 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 techniques for extending the plugins. We can make plugins extendable using two ways:

- Extend plugins with WordPress core actions and filters
- Extend plugins with custom actions and filters

In the following section, we will look at both these techniques using different plugin implementations.

## Extend plugins with WordPress core actions and filters

In this section, we will look at actions and filters provided by the WordPress core framework. So, we will create a reusable and extensible plugin for automating the file upload process for custom metafields. Let's get started.

### Planning a 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 the heavy usage of custom metafields and there can be a number of file fields within a single screen. Integrating a media uploader to each and every field can become a tedious and unnecessary task. So, we need a method to automatically integrate file fields with the media uploader. In *Chapter 4, Building Blocks of Web Applications*, we created all the custom post types and fields for the portfolio management application. However, we skipped the screen uploading process for projects. Here, we will complete the implementation while building an extensible plugin. So, let's begin with the planning:

- All the metafile fields should be automatically converted to buttons, which opens 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 to limit allowed file types

Before we begin the implementation, it's necessary to modify the WPWA Portfolio Manager plugin created in the previous chapter, to include the file field for uploading project screenshots. Open the main plugin and navigate to the `templates` folder. Include the following code at the end of the `project-meta-template.php` file:

```
<tr>
    <th style=''><label for='<?php echo $project_screens_label;
?>'><?php echo $project_screens_label; ?></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. A 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 will name it WPWA File Uploader. Create a folder called `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 on the admin side. Based on your requirements, the `wp_enqueue_scripts` action can also be used to enable the conversion in the frontend.

Create a new folder called `js` inside the `wpwa-file-uploader` folder and put in an empty JavaScript file called `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. The latest version of WordPress uses a modified media uploader, which is simple and interactive compared to the IFRAME-based uploader provided in earlier releases. So, we have to check for 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 button and image container. 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 no conflict variable. In jQuery, each loop 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. Then, 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 ID with a static prefix to be used for media uploader handling. Finally, we remove the file field using a jQuery `remove` method.



Make sure to define the `wpwa_multi_file` class on file fields to avoid potential conflicts. Otherwise, you need to check the type of the 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 similar to the following screenshot:

Having completed the field conversion, we can now move on to the media uploader integration process.

## Integrating the media uploader to buttons

WordPress provides a quick and flexible way of integrating media uploader to any type of field. The implementation can vary based on the WordPress version. We are using version 4.2.2 throughout the book, and hence, we can use the following code snippet for integration:

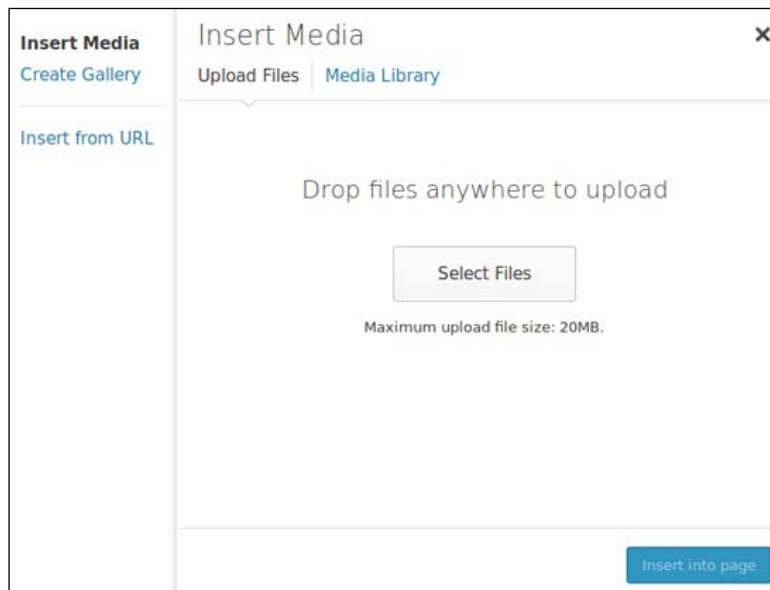
```
$jq(".wpwa_upload_btn").click(function(){  
    var uploadObject = $jq(this);
```

```

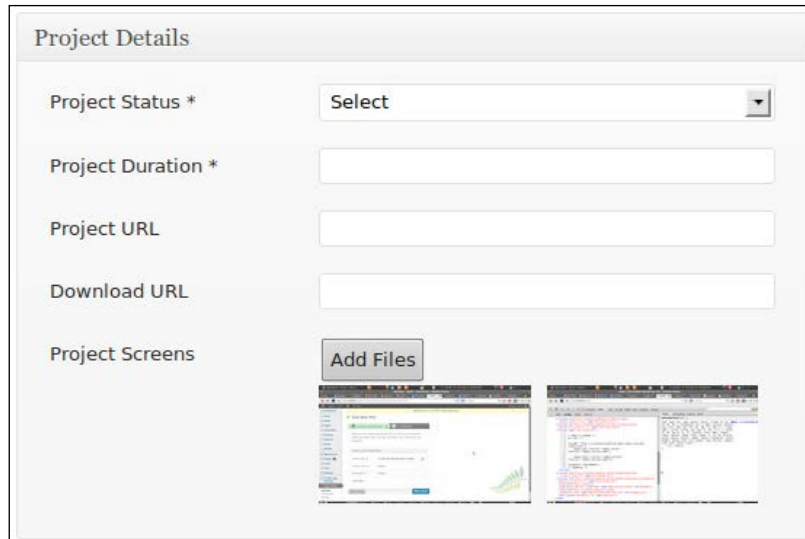
var sendAttachmentMeta = wp.media.editor.send.attachment;
wp.media.editor.send.attachment = function(props, attachment)
{
    $jq(uploadObject).parent().find(".wpwa_preview_box").append("<img
class='wpwa_img_prev' style='width:75px;height:75px' src='"+
attachment.url +"' />");
    $jq(uploadObject).parent().find(".wpwa_preview_box").append("<input
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 screen button and you will get the modern media uploader as illustrated in the following screenshot:



Upload an image to the project screens field and click the **Insert into page** button on the bottom of the page to assign a preview of the image inside the dynamic image container, under the upload button of the project screens field. You will have something similar to the following screenshot:



The screenshot shows a 'Project Details' form. It contains the following fields and elements:

- Project Status \***: A dropdown menu with 'Select' as the current value.
- Project Duration \***: A text input field.
- Project URL**: A text input field.
- Download URL**: A text input field.
- Project Screens**: A section containing an 'Add Files' button and two image thumbnails. The first thumbnail shows a web browser interface, and the second shows a code editor.

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 double-click:

```
$jq("body").on("dblclick", ".wpwa_img_prev" , function() {  
    $jq(this).parent().find(".wpwa_img_prev_hidden").remove();  
    $jq(this).remove();  
});
```

In jQuery, dynamically created elements can't be assigned directly to events. We need to use the `on` function to attach events to 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 dynamically created images. Then, we remove the `img` tag and the related hidden field from the preview box. Try uploading a few images and double-click 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 WordPress plugins for web development. So far, we have completed the core functionality of this plugin to upload images through custom metafields. 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:

- Customize the allowed types of images
- Customize the media uploader features
- Validate image sizes and dimensions

These are few of the most important enhancements and there can be more depending on your project. In this section, we will look at the first requirement for creating extending capabilities.

## Customize the allowed types of images

Usually, we do allow the .jpg, .jpeg, .png, and .gif types in image uploads. However, 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, let's consider the implementation of the `filter_mime_types` function for the image restricting process:

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

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. This means only .jpg files will be allowed for each and every post type within WordPress. Ideally, this filtering should be extensible to allow 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 execution, unless specific priority values are given.




 Those who are not familiar with priority parameters for actions and filters can take a look at the official documentation at [http://codex.wordpress.org/Function\\_Reference/add\\_filter](http://codex.wordpress.org/Function_Reference/add_filter).

A better solution is to define the filter in a specific place 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;  
}
```

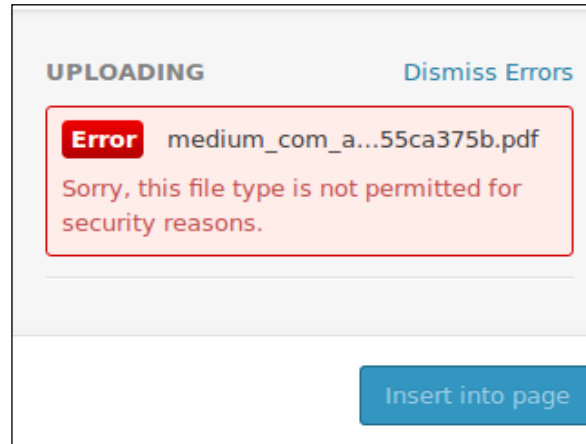
In the modified version, we have a WordPress action called `wpwa_custom_mimes`. With the use of action, any developer can extend the function to include their own requirements. In this code, the `mimes` array is passed as a reference variable to the action. Therefore, the original `mimes` array can be modified through the extended versions. WordPress uses global variables for most functionalities. Experienced web developers prefer not to use global variables. Hence, I have used reference passing instead of using global variables.

 The functionality of the WordPress `do_action` and `do_action_ref_array` functions is similar. Usually, most developers will 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 custom functions 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 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:



So, 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 the proper places to allow extending 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 post 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-template.php` 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 `class-wpwa-model-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 metabable 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 existing images:

```
public function display_projects_meta_boxes() {
    global $post,$template_data;
    $data = array();
    // Get the exisitng values from database
    $template_data['project_meta_nonce'] = wp_create_nonce('wpwa-
project-meta');
    $template_data['project_url'] = esc_url(get_post_meta( $post-
>ID, '_wpwa_project_url', true ));
    $template_data['project_duration']= esc_attr(get_post_meta(
$post->ID, '_wpwa_project_duration', true ));
    $template_data['project_download_url']=
esc_attr(get_post_meta( $post->ID, '_wpwa_project_download_url',
true ));
    $template_data['project_status'] = esc_attr(get_post_meta(
$post->ID, '_wpwa_project_status', true ));
    $template_data['project_screens'] = (array)
json_decode(get_post_meta($post->ID, "_wpwa_project_screens",
true));
    $template_data['project_status_label'] = __('Project
Status','wpwa');
    $template_data['project_duration_label'] = __('Project
Duration','wpwa');
    $template_data['project_url_label'] = __('Project
URL','wpwa');
    $template_data['project_download_url_label'] = __('Download
URL','wpwa');
    $template_data['project_screens_label'] = __('Project
Screens','wpwa');
    ob_start();
    $this->template_parser->get_template_part( 'project','meta');
    $display = ob_get_clean();
    echo $display;
}
```

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 metafields.

Finally, we have to modify the template file to display the existing image previews, as shown in the following code of the `project-meta-template.php` file:

```
<tr>
  <th style=''><label for='<?php echo $project_screens_label;
?>'><?php echo $project_screens_label; ?></label></th>
  <td><input class='widefat wpwa_multi_file' type="file"
id="project_screens" />
  <div class='wpwa_preview_box' id='project_screens_panel' >
    <?php foreach($project_screens as $screen){ ?>
      <img class='wpwa_img_prev'
style='width:75px;height:75px' src='<?php echo $screen; ?>' />
      <input class='wpwa_img_prev_hidden' type='hidden'
name='h_project_screens[]' value='<?php echo $screen; ?>' />
      <?php } ?>
    </div>
  </td>
</tr>>
```

After the file field, we will include the image preview, image hidden field, and the preview box container in the same format we used in the `uploader.js` file. 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 anytime.

Now, we have completed the project creation by including the project screens upload section. You can test the plugin by adding and removing project screens.

## Extend plugins with custom actions and filters

In this section, we will be looking at how to use our own custom filters for creating extensible plugins. We discussed the need for extending the template loader in the previous section on *Creating reusable libraries with plugins*. Here, we will complete the implementation by creating extendable features and explaining how to extend these features within other plugins.

We are planning to use the template loader across multiple plugins. So, we need to have support for loading templates from different locations, beyond the default templates and `admin/templates` folders. Let's have a look at the modified `locate_template` function of the `WPWA_Template_Loader` class in order to understand the use of custom filters:

```
public function locate_template( $template_names, $load = false,
$require_once = true ) {
  $located = false;
```

```
foreach ( (array) $template_names as $template_name ) {
    if ( empty( $template_name ) )
        continue;
    $template_name = ltrim( $template_name, '/' );
    if ( file_exists( trailingslashit( $this->plugin_path ) .
'templates/' . $template_name ) ) {
        $located = trailingslashit( $this->plugin_path ) .
'templates/' . $template_name;
        break;
    }
    elseif ( file_exists( trailingslashit( $this->plugin_path ) .
'admin/templates/' . $template_name ) ) {
        $located = trailingslashit( $this->plugin_path ) .
'admin/templates/' . $template_name;
        break;
    }
    else{
        /* Enable additional template locations using filters */
        $template_locations = apply_filters('wpwa_template_loader_
locations',array());

        foreach($template_locations as $location){
            if(file_exists( $location . $template_name)){
                $located = $location . $template_name;
                break;
            }
        }
    }
}
if ( ( true == $load ) && ! empty( $located ) )
    load_template( $located, $require_once );
return $located;
}
```

The modified implementation of the `locate_template` function contains a filter called `wpwa_template_loader_locations` for defining other template locations for different plugins. All the template locations specified using this filter will be located inside the `foreach` loop. Now, third-party developers can extend the template loader with various template loader locations. The following code shows you how to extend the template loader with custom locations:

```
add_filter('wpwa_template_loader_locations','template_locations');
function template_locations($locations){
```

```
$location = trailingslashit( wpwa_addon_path ) .  
'/email_templates/';  
array_push($locations,$location);  
return $locations;  
}
```

We can define the filter within the plugin that we are planning to access the template loader. Here, we have added another template folder called `email_templates`. This technique can be used to define multiple locations within the same plugin or within multiple plugins. WordPress actions and filters is an amazing technique for creating extensible plugins and allowing customizations for third-party developers. In the next section, we will discuss pluggable plugins with the use of pluggable functions in WordPress.

## Pluggable plugins

WordPress provides the ability to use pluggable functions through its pluggable architecture. Pluggable functions are no longer added to the core, due to the limitations in comparison to using actions and filters. WordPress codex defines pluggable functions as functions that let you override certain core functions via plugins. You can find all the pluggable functions in the `pluggable.php` file located inside the `wp-includes` folder of WordPress. The following are some of the popular pluggable functions provided by WordPress:

- `wp_logout`: This is used to log the user out of the system. You can do tasks such as removing custom session variables and recording the user session time to the database by writing a custom `wp_logout` function.
- `wp_mail`: This is used to customize the e-mail settings before sending e-mails through WordPress.
- `wp_new_user_notification`: This is used to customize the subject and contents of the e-mail sent on new user registrations.

However, there are many plugins and themes that take advantage of this technique. These plugins can be considered as different versions 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 will start with the plugin folder creation and definition. Create a folder called `wpwa-pluggable-plugin` and create a main file called `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 users' e-mails. The following is a basic implementation of such a requirement using the 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 take the e-mail heading and content, and send an e-mail message to the specified address. An important thing to consider is the use of `function_exists` function. First, it allows us to check whether a function with the same name is already defined. This function will be executed, when an other function with same name is not available. So, plugin developers can redefine the function to extend the capabilities of a core function.



In the extensible plugins, we extended a 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 on to the plugged version of this function. You can define the modified function inside any other plugin. Here, I have kept both functions inside the same plugin for simplicity:

```
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 fixed template and dynamic template. So, all the existing function calls to `wpwa_send_newsletter` 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.



In this scenario, we have used the template code inside PHP variables to explain the features of pluggable functions. Ideally, the template code used in this example should be separated into a template file and use the template loader plugin to locate the template.

Now, let's look at the execution of the newsletter sending a function on post save and update:

```

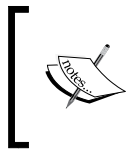
add_action('save_post', 'wpwa_create_newsletter');
function wpwa_create_newsletter($post_id) {
    if ( !wp_is_post_revision( $post_id ) ) {

```



```
    $post_title = get_the_title($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 custom functions on post save or update. Here, we are calling the `wpwa_send_newletter` function with the post's title as the heading and the post's URL as the content. Also, we have used a template called `projects`.



A function can only be reassigned this way once, so you can't install two plugins that plug the same function for different reasons. For safety, it is best to always wrap your functions with `if ( !function_exists() );` otherwise, you will produce fatal errors on plugin activation.

With pluggable functions, we can turn on or off new functionality any time without affecting the existing code. Since WordPress uses procedural function calling, pluggable plugins through functions makes 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.

## Tips for using pluggable functions

Pluggable functions seem to be an easy way of extending the functionality of plugins. However, you should be aware of the WordPress file loading process in order to make use of pluggable functions without causing unexpected issues. The following are some of the tips to be considered before creating pluggable functions:

- All the custom pluggable functions should be placed inside plugins since plugins are loaded first.
- If plugins do not contain pluggable functions, the default core function will be used.
- You shouldn't be overriding core pluggable functions in your theme files since themes are loaded after pluggable functions. Hence, the default function will be used.

Up to now, we have discussed the various types of reusable plugins suitable for web applications. Using 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 to try out the following tasks to get the best out of this chapter:

- In this chapter, we integrated a media uploader to custom fields and restricted the file types using actions. However, restrictions will be global across all types of posts. Try to make the restrictions based on custom post types and custom fields. We should be able to customize the media uploader for each field.
- Use the `wp_handle_upload` function to implement a 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 the previous chapters, we developed one main plugin 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. The integration of a media uploader was very important for web applications, which works with file-related functionalities.

In the next chapter, *Chapter 6, Customizing the Dashboard for Powerful Backends*, we will master the use of the WordPress admin section to build highly customizable backends using 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, which makes it possible to provide a prebuilt 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 it enables the 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
- Using feature-packed admin list tables
- Awesome visual presentation with admin themes
- The responsive nature of the admin dashboard

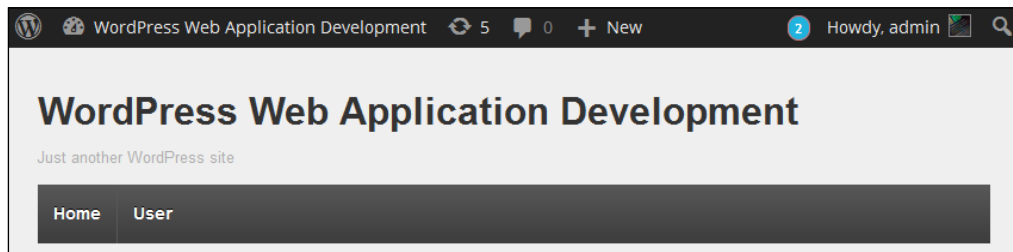
## Understanding the admin dashboard

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 will look at some of the remaining components in the perspective of web application development. Let's identify the list of sections we will 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 will 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:



Let's add a new class called `class-wpwa-dashboard.php` to our main portfolio manager plugin for functionalities in the admin section:

```
class WPWA_Dashboard {
    public function __construct() { }
}
$admin_dashboard = new WPWA_Dashboard();
?>
```



Be sure to include the `class-wpwa-dashboard.php` file inside `class-wpwa-portfolio-manager.php` using the following line of code:

```
require_once wpwa_path.'class-wpwa-dashboard.php';
```

Now, we are ready to get started with the implementation of admin features.

## Removing the admin toolbar

WordPress allows us to configure the visibility settings of the admin toolbar at the frontend. Unfortunately, it does not 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 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 the editing profile, log out, and setting a 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' ) );
```

Let's have a look at the steps:

1. 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.

2. Then, we can use `print_r()` on the returned result to identify the nodes. The following code 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
    )
)

```

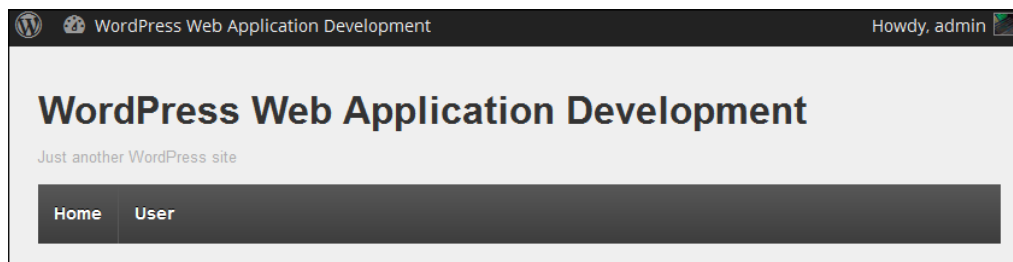
3. We need to use those unique IDs to add or remove menu items. Now, we will remove all the items other than the first item and create menu items specific to the portfolio application. So, let's remove the preceding code and modify the `wpwa_customize_admin_toolbar` function as follows:

```

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');
}

```

4. 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.
5. 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:



6. 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')) {

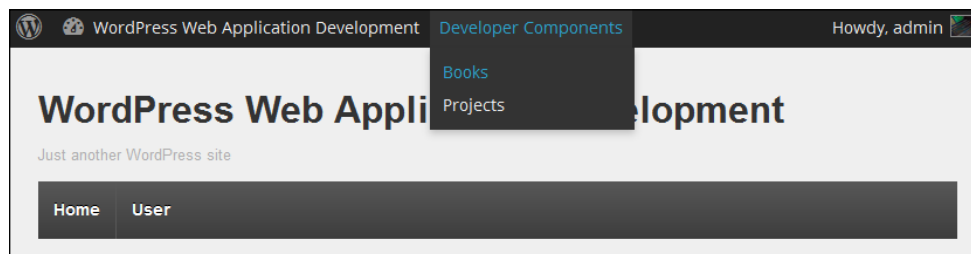
```



```
$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 for developers, containing two submenu items for books and projects. Other menu items can be implemented similarly and have been 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 to add or remove 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-hand side of the screen where we have access to all the sections of the application. In a similar way 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 an 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); echo "</pre>"; 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 the altering of the menu will be done using index values.

Up until this point, we have used 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 will see how menu items can be added and removed to cater to advanced requirements in the future. Let's get started by removing the **Dashboard** menu item 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 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 that you try the preceding method to get an understanding about the menu slugs and links before moving onto these functions.

The following code contains the functionality for removing the **Dashboard** menu item 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 a menu item. We have to provide 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.

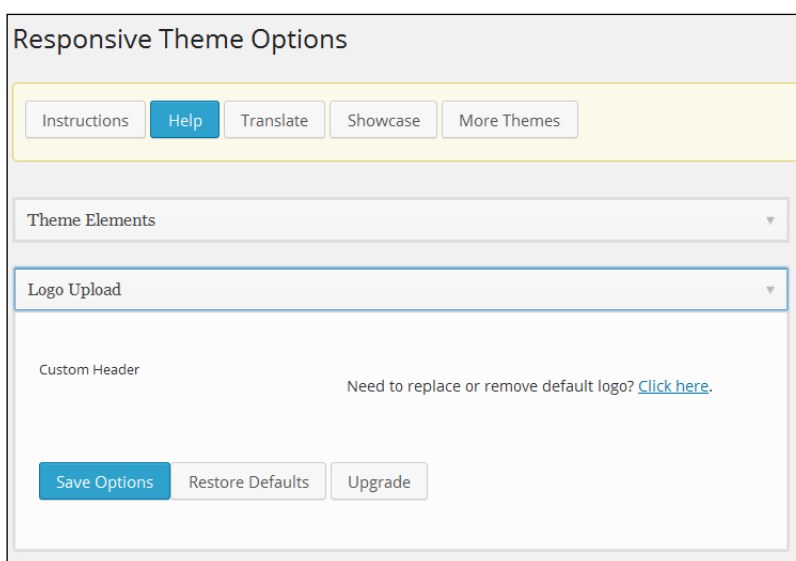
## Adding features with custom pages

WordPress was originally created as a blogging platform and evolved into a content management system. Hence, most of the core functionality is 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 two main types of custom pages in the default context:

- **Custom menu pages:** Generally, these pages are blank by default. We need to implement the interface as well as implementation for catering custom requirements that can't take advantage of the core features of WordPress.
- **Options pages:** These are used to manage the options of the application. Even though options pages are generally used for theme options, we can manage any type of applications-specific settings with these pages.

## Building options pages

The theme options page is implemented in each and every WordPress theme by default. 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 the Responsive theme using the following screenshot:



The Responsive theme uses its own layout structure for the options page. Generally, we have two ways of creating options pages for plugins:

- Using custom menu pages with our own template and processing
- Using WordPress options pages with the options and settings API

Both techniques can be effectively used for web applications. However, most developers will pick custom menu pages for large scale applications. Let's identify the differences between the two techniques:

- Custom menu pages create a separate menu item on the left menu, while the options page adds a submenu to the **Settings** menu.
- Options created with the options and settings API will be stored as individual options in the `wp_options` table. When using custom menu pages, we can save all the options inside one field and also options can be stored in any database table according to our preferences.
- Options pages with the settings API will automatically save all the options, while custom menu pages require the manual implementation of the options saving process.
- Custom menu pages provide more flexibility in design as well as the options saving process for large applications.

Basically, options pages with the options and settings API are useful for simple applications, while custom menu pages with our own implementations will be more suited for complex large-scale web applications. Having identified the differences, we can now move on to the implementation of our application settings panel with custom menu pages.

## Creating a custom layout for options pages

We decided to create our own layout, instead of using default WordPress options pages. Therefore, the design of the settings panel can be created based on our preference without any restrictions on HTML elements as well as CSS classes. Let's create a new template file inside the `template` folder of our main plugin as `settings-template.php`.

Once completed, we can start defining the options required for our application. Here, we will create two main sections called **Subscription Settings** and **Frontend Widget Settings** containing one option field each and we will continue to add more options with new requirements. So, add the following code to the `settings-template.php` file for basic options for portfolio management application:

```
<div id="wpwa-settings-panel">
```

---

```

    <h2><?php echo __('Portfolio Management Application
Settings', 'wpwa'); ?></h2>
    <form name="wpwa-settings-frm" id="wpwa-settings-frm"
method="POST">
        <div id="wpwa-subscription-setting" class="wpwa-settings-
tab"><?php echo __('Subscription Settings', 'wpwa'); ?></div>
        <div class="wpwa-settings-content">
            <div id="wpwa-subscription-setting-content"
class="wpwa-settings-tab-content">
                <div class="label"><?php echo __('Newsletter
Template', 'wpwa'); ?></div>
                <div class="field"><textarea id="wpwa_newsletter"
name="wpwa[newsletter]" class="" ><?php echo $wpwa_newsletter;
?></textarea></div>
            </div>
        </div>
        <div id="wpwa-widget-setting" class="wpwa-settings-
tab"><?php echo __('Frontend Widget Settings', 'wpwa'); ?></div>
        <div class="wpwa-settings-content">
            <div id="wpwa-widget-setting-content" class="wpwa-
settings-tab-content">
                <div class="label"><?php echo __('Number of
records in lists', 'wpwa'); ?></div>
                <div class="field"><input type="text"
id="wpwa_num_records" name="wpwa[num_records]" value="<?php echo
$wpwa_num_records; ?>" /></div>
            </div>
        </div>
        <div >
            <div class="wpwa-settings-tab-content">
                <div class="label">&nbsp;</div>
                <div class="field"><input type="submit"
id="wpwa_settings_submit" name="wpwa_settings_submit" class=""
value="<?php echo __('Save Settings', 'wpwa'); ?>" /></div>
            </div>
        </div>
    </form>
</div>

```

We have two main tabs called **Subscription Settings**, with an option called **Newsletter template** and **Frontend Widgets** with an option called **Number of records in lists**. Next, we can move on to the implementation of the options panel and the options saving process using the settings template.

## Building an application options panel

Create a new class called the `class-wpwa-settings.php` file inside the main folder of the portfolio application plugin. Be sure to include the file in the `class-wpwa-portfolio-manager.php` file. First, we need to register the menu page for the portfolio management application settings. WordPress offers functions called `add_menu_page` and `add_sub_menu_page` for manually creating blank menu pages. Let's take a look at the initial implementation of the `WPWA_Settings` class with menu page creation, as shown in the following code:

```
class WPWA_Settings{
    public function __construct(){
        add_action('admin_menu', array($this, 'add_menu'), 9);
    }
    public function add_menu(){
        add_menu_page(__('WPWA Settings', 'wpwa'), __('WPWA
Settings', 'wpwa'),'manage_options','wpwa-
settings',array($this,'settings'));
    }
    public function settings(){}
}
```

We have to use the `admin_menu` action to create a unique menu page for the settings panel. Inside the `add_menu` function, we define the **WPWA Settings** page and the function called `settings` to handle the settings panel. The `settings` function should be used to load the template we created earlier in creating the custom layout section. The following code previews the `settings` function with the updated code:

```
public function settings(){
    global $wpwa_template_loader,$template_data_settings;
    $wpwa_options = (array) get_option('wpwa_options');
    $template_data_settings['wpwa_newsletter'] =
isset($wpwa_options['newsletter']) ? $wpwa_options['newsletter'] :
'';
    $template_data_settings['wpwa_num_records'] =
isset($wpwa_options['num_records']) ? $wpwa_options['num_records']
: '';
    ob_start();
    $wpwa_template_loader->get_template_part( 'settings');
    $display = ob_get_clean();
    echo $display;
}
```

As usual, we will use the `$wpwa_template_loader` object created from our reusable plugin to load the settings template into the menu page. You will also notice the use of the `$wpwa_options` and `$template_data_settings` variables in this code. We use an option called `wpwa_options` to save all the options inside a single meta key in the `wp_options` table. This option will be empty until we save the options for the first time. Then, we use the `$template_data_settings` variable to pass the data to the templates using a global variable.

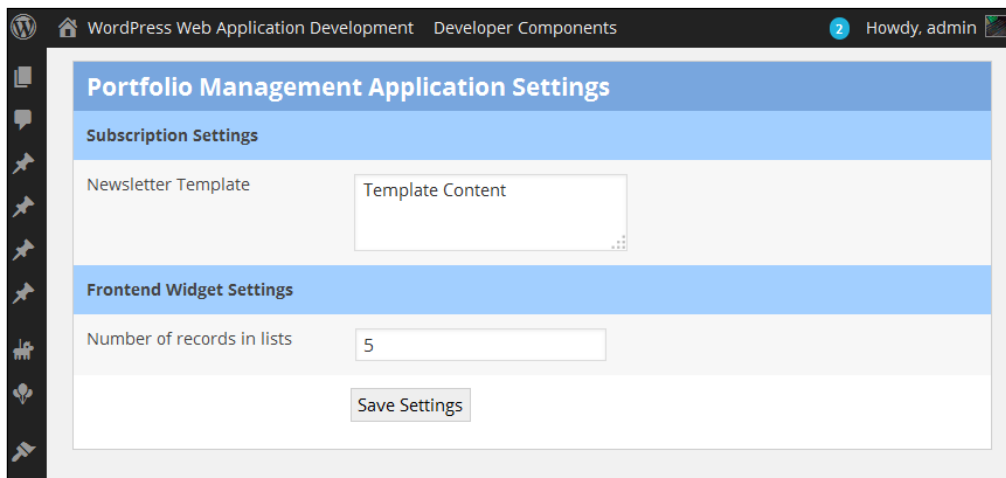
Now, we need to apply the styles to our template design. You can find the CSS styles for the settings template inside the `settings.css` file inside the `css` folder. Let's update the constructor with the following code to include the CSS file:

```
add_action('admin_enqueue_scripts', array($this, 'add_scripts'),
9);
```

Then, we can register and include the CSS file using the implementation of the `add_scripts` function, as shown in the following code:

```
public function add_scripts() {
    wp_register_style('wpwa_settings_styles', wpwa_url.
'css/settings.css');
    wp_enqueue_style('wpwa_settings_styles');
}
```

We should be able to see the new menu page for **WPWA Settings** on the left menu. Once you click on the menu item, you should see the settings page similar to following screenshot:





The final part of this task is the implementation of the options saving process. Since we are using custom menu pages, we have to develop the code from scratch to save the options. Let's begin by updating the class constructor with the following action:

```
add_action('init', array($this, 'save_settings'));
```

Usually, we use the `init` action to intercept the `GET` or `POST` requests in WordPress. Our settings panel works on normal form submission, and hence, we have to extract data from the `$_POST` array and store the options in the database. Let's consider the implementation of the `save_settings` function with the options saving code:

```
public function save_settings(){
    if(isset($_POST['wpwa_settings_submit'])){
        $wpwa_options = (array) get_option('wpwa_options');
        foreach($_POST['wpwa'] as $setting=>$val){
            $wpwa_options[$setting] = $val;
        }
        update_option('wpwa_options',$wpwa_options);
        add_action('admin_notices', array($this,'settings_notice' ));
    }
}

public function settings_notice() {
    ?>
    <div class="updated">
        <p><?php _e( 'SettingsUpdated!', 'wpwa' ); ?></p>
    </div>
    <?php
}
```

First, we check the availability of the submit button using the `wpwa_settings_submit` key in the `POST` array. You might have noticed that we defined all the form field names inside an array called `wpwa` for simplified access. We can get the existing setting from the `wp_options` table and update it based on the settings available in the `$_POST['wpwa']` variable. Finally, we add an admin notice to display the success message.

Now, you should be able to save the settings for our application. This settings panel is at the most basic level at this stage. As we get more settings, we will have to update the options saving process with conditional checks and filtering.

## Using the WordPress options API

We choose the options panel with custom menu pages over default WordPress options managing technique. However, it's important to know how to use the WordPress options and settings API in situations where you need a simple options panel. 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 for working 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 into the database. It doesn't do anything if the option already exists in the database.
- `delete_option`: This is used to remove existing option/value pair from the database. It returns `true` when the option is deleted successfully and `false` on failure or when the option does not exist.
- `get_option`: This is used to retrieve option/value pairs from the database. It returns `false` if the options don't exist in the database.
- `update_option`: This is used to update option/value pairs in the database. First, it checks for the existence of the option and updates it accordingly. If the option does not exist, it will be added using the `add_option` function.

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 hence, are considered as the safest way of working with the `wp_options` table. You can look at the complete WordPress options API at [http://codex.wordpress.org/Options\\_API](http://codex.wordpress.org/Options_API).

Let's start building a simple options page with the default technique. First, we 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_options_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:

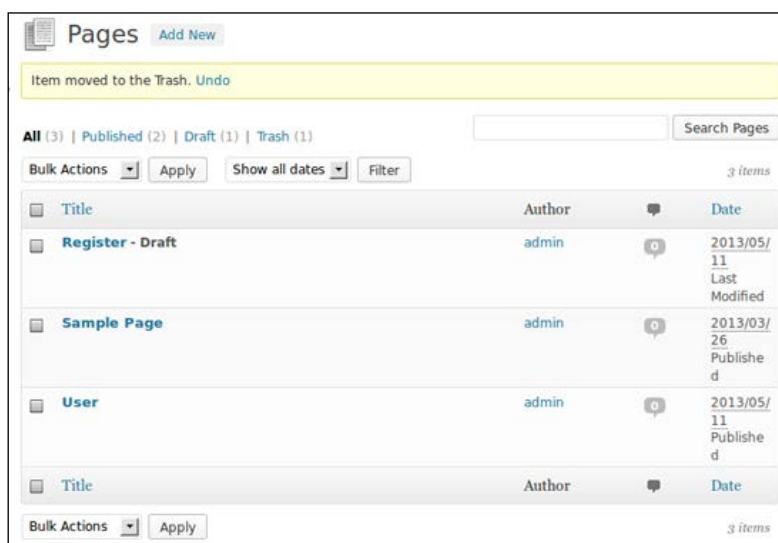
```
<?php
function wpwa_options_page() {
?>
<div class="wrap">
<form method="post" action="options.php">
  <?php settings_fields( 'wpwa-settings-group' ); ?>
  <table class="form-table">
    <tr valign="top">
      <th scope="row">Option1</th>
      <td><input type="text" name="option1" value="<?php 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 for saving 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 names for the field name 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 fields 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 creating options panels. Make sure to test both the techniques to identify the pros and cons of each.

## 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 will discuss the default admin lists and how we can customize them in our applications.

## Working with default admin list tables

We have two ways of working with admin list tables. First, we can customize the existing admin lists for posts, users, comments, and so on using the available actions and filters. The second method is to use the WordPress code for admin list tables and create our own tables with all the built-in features. The first option is the easiest and most recommended of the two options. In this section, we will be looking at how to customize the default list tables available in the WordPress admin section.

## The post list

The post list can be accessed using the **Posts | All Posts** menu item; it contains all the available posts in the WordPress database. By default, it will display the data for the following fields:

- Title
- Author
- Categories
- Tags
- Number of comments
- Date

On a normal WordPress site, blog posts will be listed in this list, and hence, we rarely need to make any customizations. However, WordPress custom post types are frequently used in web application development. You won't find the custom posts inside the default WordPress post list. Instead, WordPress provides a separate list of each custom post type. Assume that we have a custom post type called project. Then, we can access the project post list from the **Projects | All Projects** menu item. We can customize the admin list features for custom post types. Let's look at the customizable features for custom post types and practical use cases:

- Custom actions for custom post types
- Custom filters for custom post types
- Post status links
- Post list columns

## Creating custom actions for custom posts

We can find the actions list for custom post types in the top-left corner of the list in the **Bulk Actions** dropdown. The default actions for custom posts are **Edit** and **Move To Trash**. In web applications, we need custom actions to manage custom posts. So, let's see how we can create and execute custom actions for the posts list:

```
add_action('admin_footer', 'wpwa_project_action_buttons');
function wpwa_project_action_buttons() {
    $screen = get_current_screen();
    if ( $screen->id != "edit-wpwa_project" )
        return;
?>
<script type="text/javascript">
```

---

```

    jQuery(document).ready(function($) {
        $('option').val('wpwa_pro_planned_switch').text('Switch
Project to Planned').appendTo("select[name='action']");
        $('option').val('wpwa_pro_failed_switch').text('Switch
Project to Failed').appendTo("select[name='action']");
    });
</script>
<?php
}

```

WordPress doesn't offer filters or actions for changing the default **Bulk Actions** list. So, we have to make use of other actions with some workarounds. We use the `admin_footer` action so that we can embed the necessary JavaScript code for the footer. First, we have to check the current screen so that we only enable custom actions for necessary post types. We should check whether the screen matches `edit-{custom post type}`. Then, we add the new options to the existing dropdown field with unique values. Now, you should see two new options inside the **Bulk Actions** dropdown for the project post type.

The next part is to use the custom options to execute some custom tasks. Let's take a look at the code:

```

function wpwa_project_page_loaded(){
    if( (isset($_GET['action']) && $_GET['action'] ===
'wpwa_pro_planned_switch') ||
        (isset($_GET['action2']) && $_GET['action2'] ===
'wpwa_pro_planned_switch')) {
        $projects = isset($_GET['post']) ? $_GET['post'] : '';
        if ('' != $projects) {
            foreach ($projects as $project) {
                update_post_meta($project, '_wpwa_project_status',
'planned');
            }
        }
    }
}

```

First, you have to select some projects from the list and then select **Switch Project to Planned** from the **Bulk Actions** dropdown. Once you click the **Apply** button, we check the proper action using the value of dropdown and then get the projects list. Finally, we update the status of all the projects to the **Planned** status. This is a great feature for quickly working on a large number of custom post types at once. We have only implemented one custom action in our example. You can implement the same for the rest of the project statuses.

## Creating custom filters for custom post types

The admin list provides another useful feature for filtering custom post types. By default, we can filter the custom post types by date. In web applications, we can use this feature to create our own filters by using a technique similar to the custom action creation. Let's see how we can add a filter for project statuses:

```
function wpwa_project_list_filters() {
    global $typenow;
    $project_status = isset($_GET['wpwa_project_status']) ?
    $_GET['wpwa_project_status'] : '';
    if( $typenow == 'wpwa_project' ){
        $display = "<select name='wpwa_project_status'
id='wpwa_project_status' class='postform'>";
        $display .= "<option value=''>Show All Projects</option>";
        $display .= "<option value='planned' ".
selected($project_status, 'planned', false) ." >Planned</option>";
        $display .= "<option value='pending' ".
selected($project_status, 'pending', false) .">Pending</option>";
        $display .= "<option value='failed' ".
selected($project_status, 'failed', false) .">Failed</option>";
        $display .= "<option value='completed' ".
selected($project_status, 'completed', false)
.">Completed</option>";
        $display .= "</select>";
        echo $display;
    }
}
add_action( 'restrict_manage_posts', 'wpwa_project_list_filters' );
```

We can find an action called `restrict_manage_posts` inside WordPress core files to filter the records of normal posts as well as custom post types. In this scenario, we create a custom filter by defining a dropdown field with the necessary filters. We can get the selected filter value from the `$_GET` array and display it as the selected value for the filter. It's important to check the post types using the `$typenow` global variable to prevent new filters for all post types. Now, we have the input fields to filter projects from our list. The next task is to apply the filters into post list query. Consider the following code for filtering projects based on the status value:

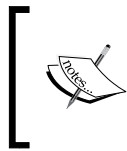
```
add_filter( 'parse_query', 'wpwa_project_list_filter_query' );
function wpwa_project_list_filter_query( $query ){
    global $pagenow;
    $type = 'wpwa_project';
    if (isset($_GET['post_type'])) {
        $type = $_GET['post_type'];
    }
}
```

```

if ( 'wpwa_project' == $type && is_admin() &&
    $pagenow=='edit.php' && isset($_GET['wpwa_project_status']) &&
    $_GET['wpwa_project_status'] != '' ) {
    $query->query_vars['meta_key'] = '_wpwa_project_status';
    $query->query_vars['meta_value'] =
$_GET['wpwa_project_status'];
}
}

```

We have to use the WordPress `parse_query` filter to apply our custom filters into the post list query.



The `parse_query` filter is an action triggered after `WP_Query->parse_query()` has set up query variables (such as the various `is_` variables used for conditional tags). We can use this action to modify the queries in the current page or post.

As usual, we have to check the proper post type and current page. The post list is loaded from the `edit.php` file, and hence, we have to use it for the conditional check. In this scenario, we are filtering the project status, and hence, we should also check the availability of the `wpwa_project_status` key inside the `$_GET` array. If all conditions are satisfied, we change the default query variables to include the meta key and meta value for the project status. Now, the modified query will only return the projects matching the specified status.



We created the project status as a custom field, and hence, it's stored in the `wp_postmeta` table. Therefore, we can directly change the query by using the meta key and value. If you store these values on a different database table or you are filtering a value from a different table, custom filtering will be complex and will require manual queries.

Now, you can select a specific status from the new dropdown field and click on the **Filter** button to filter the projects by status. You can repeat the same technique to create more filters as required.

## Creating custom post status links

WordPress posts can have one of the many statuses at any given time, and this is used to determine how the post is handled. We have eight post statuses by default for normal posts as well as custom post types. The following are the default post statuses with their meaning within a post life cycle:

- **Published:** The post is published and viewable for everyone
- **Future:** The post is scheduled to be published on a future date



- **Draft:** The incomplete post is viewable by anyone with the proper user level
- **Pending:** The post is awaiting the approval of a user with the `publish_posts` capability to publish
- **Private:** The post is viewable only to WordPress users at an administrator level
- **Trash:** The posts in **Trash** are assigned the trash status
- **Auto-Draft:** These are the revisions that WordPress saves automatically while you are editing
- **Inherit:** This is used with a child post (such as attachments and revisions) to determine the actual status from the parent post (`inherit`)

These post statuses will be displayed on top of the projects list. However, it only displays statuses with at least one post. This is another way of filtering posts by post statuses. In this application, we used default post statuses and created a meta key to manage the project status. We can also have used post statuses for projects by removing the default statuses and creating custom statuses. Let's see how we can create a status link for the custom post status using the following code:

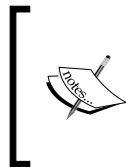
```
add_action('admin_footer-post-new.php',
'wpwa_create_post_status_list');
add_action('admin_footer-post.php', 'wpwa_create_post_status_list'
);
function wpwa_create_post_status_list() {
    global $post;
    $complete = '';
    $label = '';
    if($post->post_type == 'wpwa_project'){
        $complete = ' selected=selected ';
        $label = "<span id='post-status-display'>Released</span>";
    }
    ?>
    <script>
    jQuery(document).ready(function($) {
        $("select#post_status").append("<option value='wpwa_pro_
released' <?php echo $complete; ?> >Released
Status</option>");
        $(".misc-pub-section label").append("<?php echo $label;
?>");
    });
    </script>
```

```

<?php
}
}

```

We have to use `admin_footer-post-new.php` and `admin_footer-post.php` to include custom statuses on the project creation screen as well as projects list. As usual, we embed the new status into the post status dropdown using JavaScript. Now, the new status will be available automatically as a link in the projects list.



We can create a new post status by using the `register_post_status` function on the `init` action. You should never create new post statuses before the `init` action. You can find complete guidelines at [http://codex.wordpress.org/Function\\_Reference/register\\_post\\_status](http://codex.wordpress.org/Function_Reference/register_post_status).

Once you click on the link, projects with the `wpwa_pro_released` status will be displayed. This is a useful feature for applications with many custom post statuses.

## Displaying custom list columns

We have discussed all the major features of the admin post list, such as custom actions, filtering, and post statuses. Finally, we will complete this section by using custom columns in post lists. The custom post list usually displays title and date columns in the post list. Consider our portfolio application with managing projects. Generally, we would need the project status, duration, and a clickable URL in the list to make it easier to manage multiple projects at the same time. Let's take a look at the following code for adding custom columns to the list:

```

add_filter( 'manage_edit-wpwa_project_columns',
'wpwa_project_list_columns' );
function wpwa_project_list_columns( $columns ) {
    $columns = array(
        'cb' => '<input type="checkbox" />',
        'title' => __( 'Project' , 'wpwa' ),
        'duration' => __( 'Duration' , 'wpwa' ),
        'status' => __( 'Project Status' , 'wpwa' ),
        'date' => __( 'Date' , 'wpwa' ),
    );
    return $columns;
}

```

We can decide the columns in the post list using the `manage_edit-{custom post type}_columns` filter. In this scenario, we are trying to add two new custom columns called `Duration` and `Project Status`. If needed, we can remove the existing columns by removing it from the array. Now, you should see the new columns in the projects list with empty values. Next, we have to retrieve and display the values for those new columns. Consider the following code for displaying values:

```
add_action( 'manage_wpwa_project_posts_custom_column',
'wpwa_manage_project_columns', 10, 2 );
function wpwa_manage_project_columns( $column, $post_id ) {
    global $post;
    switch( $column ) {
        case 'duration' :
            $duration = get_post_meta( $post_id,
'_wpwa_project_duration', true );
            if ( empty( $duration ) )
                echo __( '-' );
            else
                echo $duration;
            break;
        case 'status' :
            $status = get_post_meta( $post_id,
'_wpwa_project_status', true );
            if ( empty( $status ) )
                echo __( '-' );
            else
                echo $status;
            break;
        default :
            break;
    }
}
```

We have a built-in action called `manage_{custom post type}_posts_custom_column` for managing and filtering the values in the post list. All the available columns are passed as a parameter to this function. We can switch the columns and get the values from the `wp_postmeta` table using the respective meta keys. If needed, we can also filter and format the values before sending them to the list. Now, you should see the new columns and values inside the projects list. The following screenshot previews the projects list after all the customizations are completed:

All (3)   Published (3)   Trash (1)				Search Projects	
Bulk Actions	Apply	All dates	Show All Projects	Filter	3 items
Project	Duration		Project Status	Date	
Project 1	2		planned	6 mins ago Published	
Project 2	3		pending	5 mins ago Published	
Project 3	4		planned	5 mins ago Published	
Project	Duration		Project Status	Date	
Bulk Actions	Apply				3 items

We have covered all the major features in the post list throughout the previous sections, and now you should be able to customize any custom post list according to your preference. You can find the source code for this section inside the `wpwa_actions_filters.php` file.


## The user list

We can access the backend user list by navigating to **Users | All Users** in the admin panel. The WordPress user list is also built on top of the `WP_List_Table` class, and hence, we can have most of the features we discussed in the previous section on the post lists. I won't discuss all those features for the user list as the code is similar to the variation of actions and filters. I suggest that you try custom actions, filters, and custom columns for the user list on your own and check with the source code provided in the official book website at <http://www.innovativephp.com/wordpress-web-applications>.


We will discuss the practical usages of these features in the user list instead of discussing the code. The user list only contains `Delete` as a custom action for deleting multiple users at once. In large applications, we will need features such as:

- Confirming the e-mail address after registration
- Receiving user approval before login
- Enabling and disabling users

In such scenarios, we can use these as custom actions in the **Bulk Actions** dropdown to work with multiple users instantly.

[  You can use the `admin_footer` action for adding custom actions to the dropdown field and the `load-users.php` action for updating the users after selecting a specific action. ]

Adding dynamic columns to the user list is similar to the technique used in the post list, and hence, we won't discuss it in detail. Next, we have user roles as links on the top instead of post statuses. We have custom user roles in our portfolio application and those will be added as links for filtering users. Finally, we can create custom filters in the user list to filter approved/unapproved, confirmed/unconfirmed, and enabled/disabled users based on the scenario we discussed.


[  You can use the `restrict_manage_users` action for creating the new filter dropdowns for users and the `pre_user_query` action for customizing the query based on the filter values. ]

We have discussed the practical customization of the user list and mentioned the filters for implementing them. Make sure that you try these implementations on your own and check with the code on the official book website.

## The comments list

**Comments** is the other major list used for web applications and can be accessed using the **Comments** menu in the admin panel. Similar to what we did with the user list, we will only be only discussing the practical usage instead of code.

First, we can consider the custom actions for comments. By default, we have **Unapprove**, **Approve**, **Mark as Spam**, and **Move to Trash** as the custom actions. Let's recall our example from *Chapter 1, WordPress as a Web Application Framework*. We used comments to provide answers to questions created through custom post types. So, we can have custom actions, such as **Mark answer as correct**, **Mark answer as incorrect**, and so on.

[  You can use the `admin_footer` action for adding custom actions to the dropdown field and the `init` action for updating the comments after selecting a specific action. ]

Next, we can add new filters to the comments list. We can add a filter for correct/incorrect answers. Also, we can add another filter for answers from a specific user. Finally, we might need a filter for answers from a specific question. These are some of the use cases in our simple example, and you will find many such filters in advanced applications.



You can use the `restrict_manage_comments` action for creating the new filter dropdowns for comments and the `pre_get_comments` action for customizing the query based on the filter values.

Finally, we have comment status links on top instead of post status links. However, WordPress doesn't provide a function for adding new custom comment statuses, and hence, we can't change the filtering links available on the top of the list.

Throughout the previous three sections, we discussed how to customize the default user, post, and comments list. However, we haven't covered the most important part of creating our own lists with the admin list table. In large web applications, we will be using custom database tables more than the default tables, and hence, custom list building is a vital feature. We will be covering the `WP_List_Table` class and its usage in the next section.

## Building extended lists

The extended version of `WP_List_Table` can be created by manually overriding each and every function in the base class. However, we will 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.

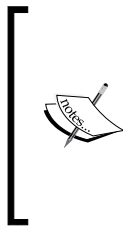
Create a new class called `class-wpwa-list-table.php` and copy the `list-table-example.php` file from the downloaded plugin. Then, you can change the plugin descriptions and information if necessary. Now, we are ready to customize the template.

## Using the admin list table for the following developers

In requirements planning, we identified two roles called `follower` and `developer`, 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. The 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
- Selected developers should be saved in a custom database table with follower details on executing custom action

Let's get started.



We need to have the `wp_subscribed_developers` table for implementing this feature. It's not yet available in our database. The activation handler of our portfolio application plugin is updated with the code for creating the new database table. So, you have to deactivate the plugin and replace the plugin with the new version. Then, you can activate the plugin again to create the new database table for subscribed developers.

### 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 the instance variables

The 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 follows:

```
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 name of the listed records
        'singular' => 'developer',
        //plural name of the listed records
        'plural' => 'developers',
        //does this table support ajax?
        'ajax' => false
    ));
}
```

Inside the array of configurations, we have to define a singular and plural name for the records. This should be a unique name and has no relation to database tables or columns. We can also define the support for AJAX, although it will be not discussed here.

## Step 4 – implementing the custom column handlers

In this step, we need to define the 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 data set using the following code:

```
Array
(
    [0] => Array
```



```
(
  [ID] => 24
  [developer_name] => John Doe
)
[1] => Array
(
  [ID] => 22
  [developer_name] => Mark
)
)
```

The preceding data set is generated manually to contain custom keys. When we are using the direct database result for the dataset, these keys will be replaced by database columns. Here, we are using a column name 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 the column default handlers

In the previous step, we created column functions for available columns in the list. If you skip the definition of specific function for a column, we should create a default callback 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
  }
}
```

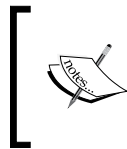
Here, we need to define each and every column that will not be defined separately. Even though we have defined `developer_name`, it won't be used as we have a specific function called `column_developer_name`.

## 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 the records and execute specific actions on the **Bulk Actions** dropdown 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 a singular label we created inside the constructor to set the name of checkbox as an array. The second parameter contains the ID for the row as defined in our data set. This value should be the ID of the record in the database table.



We can define any key for the ID in the data set. However, 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 reused across all the 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', 'wpwa')
    );
    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. Later, 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 list

The `get_sortable_columns` function is pretty straightforward like the previous one, where we define the columns to be sortable. Consider the following modified implementation of this 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 data set from the database, the key and value will be same.

## Step 9 – creating a list of bulk actions

In the default post list, we can see different options called **Edit**, **Move to Trash**, and so on inside the **Bulk Actions** dropdown. Similarly, we can include custom actions in custom lists. This is one of the most powerful features of this list, in implementing complex requirements in web applications. Consider the modified implementation of the `get_bulk_actions` function:

```
function get_bulk_actions() {
    $actions = array(
        'follow' => __('Follow', 'wpwa'),
    );
    return $actions;
}
```

The preceding function is prebuilt and returns a 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

Up until now, we have carried out the configuration part of the list, and now, we are moving onto the exciting part by 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. However, I prefer keeping the function at 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 a 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 will include the list on the left-hand side of the 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');
```

The 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 the 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();
}
```

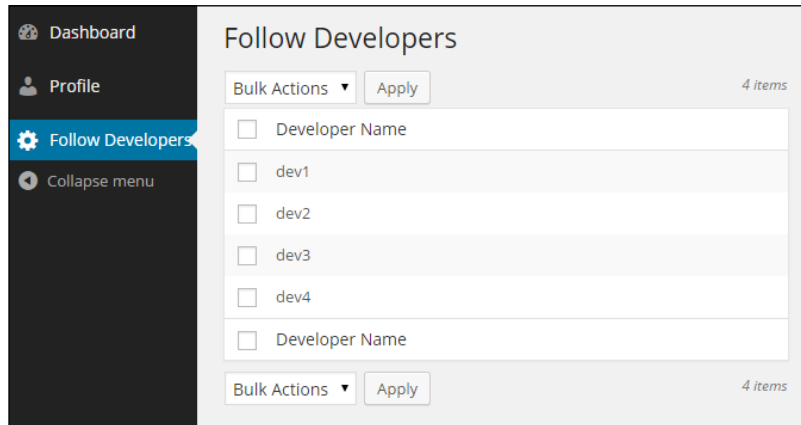
We can begin the implementation by initializing an object of the `WPWA_List_Table` class. Then, we execute a WordPress query using a built-in `WP_User_Query` class to retrieve the list of users with the role of a developer. We have chosen to manually create the dataset by traversing through the database 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 into 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><?php echo __('Follow Developers','wpwa'); ?></h2>
    <form 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 a different behavior 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 the 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
                )
            );
        }
    }
}
```

```
$msg = __('Successfully completed.', 'wpwa') . "<a href='" .  
admin_url() . "?page=wpwa_subscriptions">  
" . __('Follow More Developers', 'wpwa') . "</a>";  
wp_die($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 developer's list.

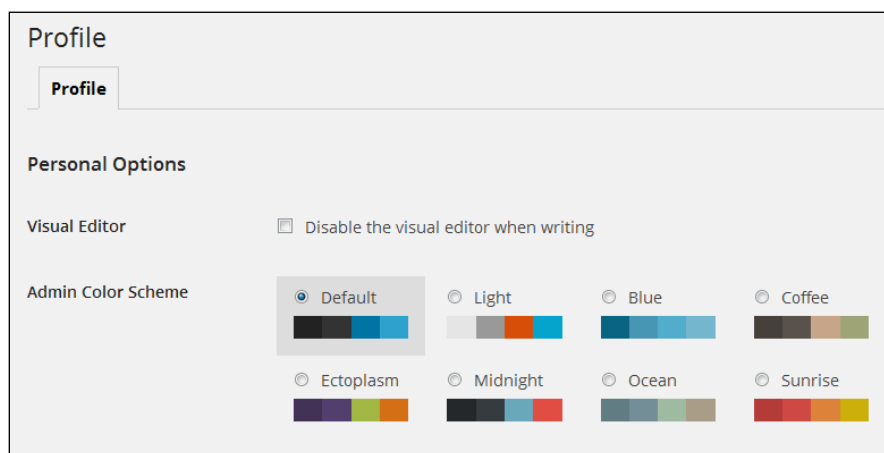
Now, we have a fully featured custom list with all the basic grid functionalities. We can create as many lists as possible by creating new templates or creating a reusable class for this library. It's for the future, but for now, you can test the list by following developers.

## An awesome visual presentation for admin screens

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 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. However, we can provide eye-catching interfaces by changing the default styles of the admin theme.

## Using existing themes

WordPress Version 3.8 and higher provide the ability to change the color theme of the admin section using eight different color schemas. This is a great feature for changing the look and feel of admin screens in WordPress. Users are allowed to pick their own color scheme for the site, making it flexible for different users with different color preferences. You can change the color theme from the **Your Profile** section of the **Users** menu in the WordPress admin screen. The following screenshot previews the available color themes in the user profiles section:



This is the most basic version of admin design customization as it only changes the color scheme of the admin section. Other features such as elements, dimensions, icons, and so on won't be changed. These features let users choose their color scheme. However, most of you will need a simplified admin design or feature-rich modern dashboards. This is where the admin theme plugin comes into action. We will be covering admin theme plugins in the next section.

## Using plugin-based third-party admin themes

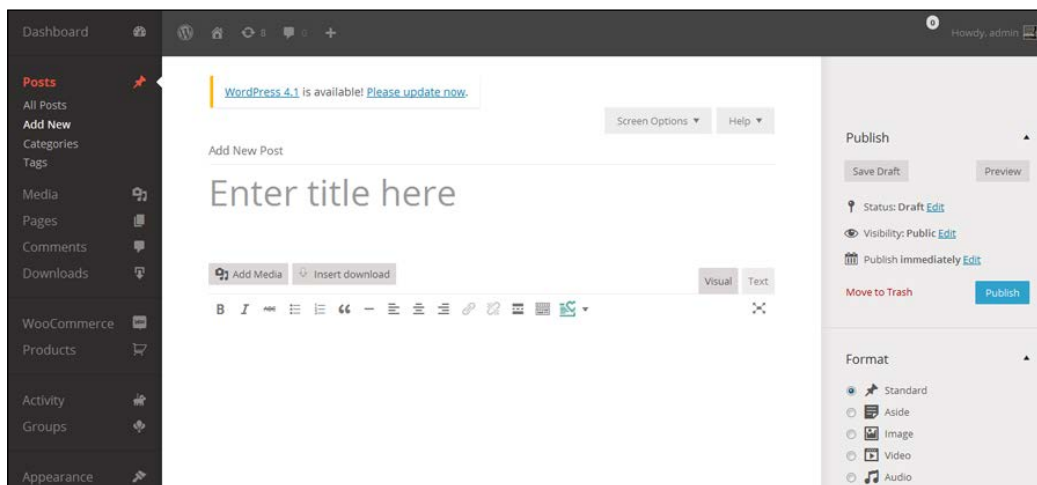
The experienced WordPress developers know that the theme is designed to provide the frontend functionality of a WordPress site. You can change the theme as you wish and create custom themes based on your preference. However, the WordPress admin features and design templates are stored inside the WordPress core files. WordPress doesn't provide a feature for changing the admin theme or designing your own admin theme. So, we can't provide the admin designs as part of a normal WordPress theme. As a solution, we use plugins to customize the design of admin designs. You can find many free and premium third-party plugins for admin themes.



The following are some of the most popular admin theme plugins available in the WordPress plugin directory and theme marketplaces:

- **Bootstrap Admin:** A clean, minimalistic admin theme based on Twitter's Bootstrap ( <https://wordpress.org/plugins/bootstrap-admin/> )
- **Slate Admin theme:** A clean, simplified WordPress admin theme ( <https://wordpress.org/plugins/slate-admin-theme/> )
- **Retina Press:** A brand new retina display custom theme designed for the Wordpress admin ( <http://codecanyon.net/item/retina-press-wordpress-admin-theme/4872562> )

We will look at one of the listed admin theme plugins to understand how it changes the look and feel of admin screens. We will be using the Slate Admin theme as it's free and compatible with WordPress 4.0 and upwards and upwards. The following screen previews the post creation screen using the Slate Admin theme:



Once this plugin is activated, you may notice that the design of the post creation screen looks different from the default design and provides a much cleaner interface. However, all the fields, text, and structure of the page still look the same.

As a developer, it's important for you to understand how an admin theme plugin works and how you can create an admin theme plugin. We have already seen that admin templates are generated from the core. So, the elements and structure are fixed to certain elements, CSS classes, and IDs.

In WordPress themes, we can completely change the design of the frontend posts and pages by using our own templates and elements. Unfortunately, we can't use the same procedure for admin themes as we don't have access to backend templates, and we may lose some features by changing the elements on the WordPress admin dashboard. In simple terms, it's not possible to change the elements of admin templates using plugins. We can only change the design using CSS and JavaScript. Admin theme plugins offer great features for changing the design of admin screens. However, they are not flexible enough to create admin dashboards with advanced components and designs. If we need a unique admin screens with our own features, we need to remove the default admin menus and screens using filters and implement all the features by using our own templates inside custom menu pages.

So far, we have looked at default color schemes and changing designs using admin theme plugins. Next, we will discuss how to create your own admin theme using a plugin.

## Creating your own admin theme

Building a complete admin theme is a time-consuming task, which is beyond the scope of this chapter, as we need to define custom styles for all the existing CSS selectors. Therefore, we will provide a head start to the admin theme design by altering the main navigation menu. Let's start by creating a class called `class-wpwa-admin-theme.php` inside the portfolio management application. As usual, you will have to require the file inside the main plugin file.

Let's start by defining the stylesheet for the admin theme using the following 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 to create a new CSS file called `wp-admin.css` inside a folder called `css`.



The CSS file used for an admin theme is loaded after the default WordPress admin stylesheets. Therefore, it will override the existing styles provided by the default stylesheet.

In this section, we will style the main navigation menu of WordPress. You can update the CSS file with menu-specific styles, as illustrated in the following code:

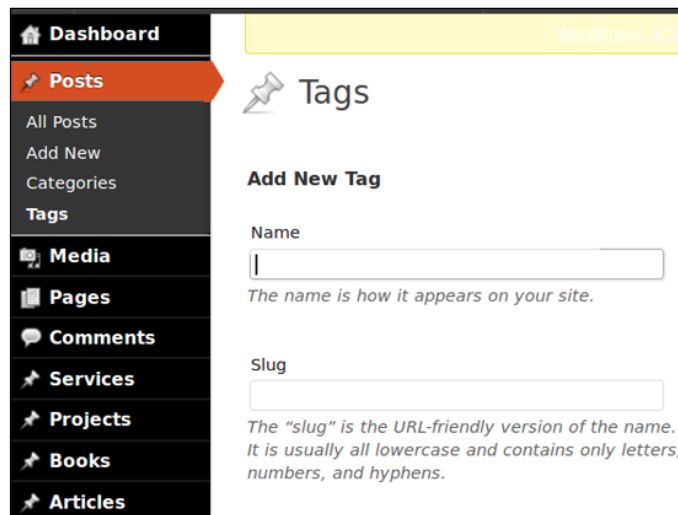
```
#adminmenuback,#adminmenuwrap { background: #000; }
#adminmenu a{ color : #FFF; }
#adminmenu a.menu-top, #adminmenu .wp-submenu .wp-submenu-head {
    border-bottom-color: #191A1B;
    border-top-color: #191A1B;
}
#adminmenu .wp-submenu, .folded #adminmenu a.wp-has-current-
submenu:focus + .wp-submenu, .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 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 that 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
```

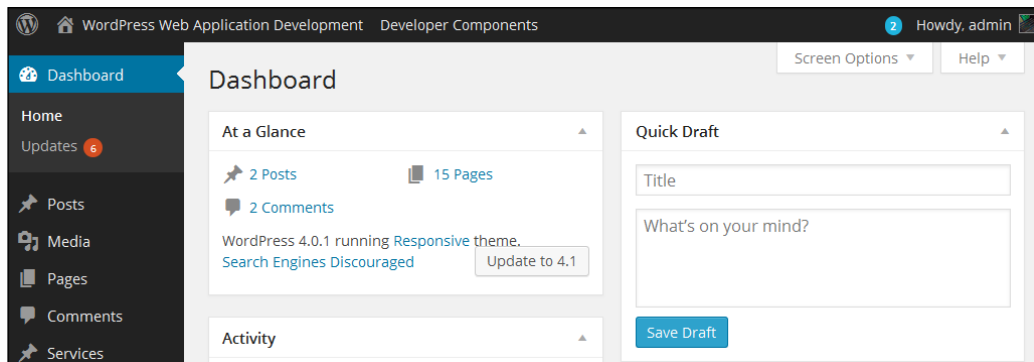
- 18.0 - Users
- 19.0 - Tools
- 20.0 - Settings
- 21.0 - Admin Footer
- 22.0 - About Pages
- 23.0 - Full Overlay w/ Sidebar
- 24.0 - Customize Loader
- 25.0 - Misc

Use the table of content provided in the `wp-admin.css` file to style the remaining components to build a complete admin theme.

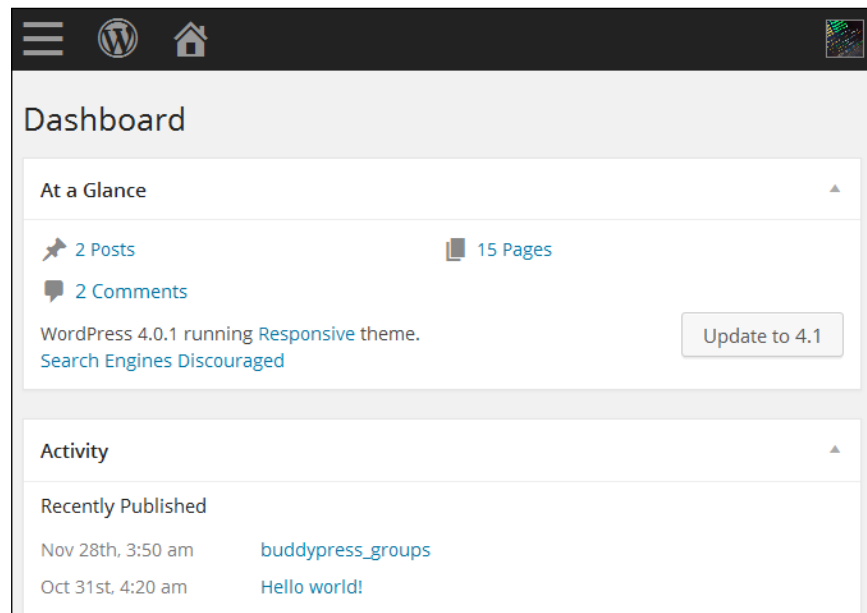
## The responsive nature of the admin dashboard

The responsive design has become one of the major trends in web application development with an 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 an admin dashboard in its default resolution to understand its 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 the 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 responsive nature .

Since the admin section is responsive by default, we don't have to do anything else to make it responsive. However, 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 an admin panel-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 the admin list table to allow subscriptions. Now, try to include AJAX-based star rating system to allow followers to rate developers by implementing a custom column in the existing list.

- 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 the 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 in building the menu.

Typical web applications contain a large amount of applications settings to let users customize the application based on their preferences. So, we looked at creating our own options pages as well as the default options management features provided by the WordPress core. Also, we looked at the default lists in the admin panel and extended the existing WordPress admin list tables to cater to custom functionality beyond core implementation.

Finally, we looked at the importance of responsive web design and how the WordPress admin dashboard adapts to responsive layouts while showing a glimpse into the WordPress admin theme design.

In the next chapter, *Chapter 7, Adjusting Theme for Amazing Frontends*, we will explore how we can manage existing WordPress themes to build complex web application layouts using modern techniques.





# 7

## Adjusting Theme 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 its user interfaces and the simplicity of using its features. Most expert-level web developers tend to give more focus on development tasks in complex applications. However, the application's design plays a vital role in building the initial user base. WordPress uses themes that allow you to create the frontend of web applications with highly extendable features that 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 focus 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 look 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:

- A basic file structure of the WordPress theme
- Understanding the template execution hierarchy
- Web application layout creation techniques
- Building a portfolio application home page
- Widgetizing application layouts

- Generating the application frontend menu
- Managing options and widgets with a customizer
- Creating pluggable templates
- Planning action hooks for layouts

## **An introduction to the WordPress application frontend**

WordPress powers its frontend with a concept called themes, which consists 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 its 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.

## **A basic file structure of the WordPress theme**

As a WordPress developer, you should have a fairly good idea about the default file structure of WordPress themes. Let's briefly introduce 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 of blogging systems, and hence, can be omitted in the context of web applications. Since custom posts are widely used in application development, we need to focus more on templates such as `single-{post_type}` and `archive-{post_type}` rather than `category.php`, `archive.php` and `tag.php`.



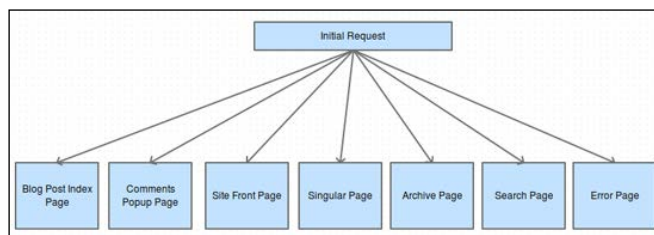
Even though default themes contain a number of files for providing default features, only `style.css` and `index.php` files are enough for implementing a WordPress theme. Complex web applications themes are possible with the standalone `index.php` file.

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 mind is the static content. However, 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 the 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 will 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](http://codex.wordpress.org/images/1/18/Template_Hierarchy.png).

Following diagram shows the template execution hierarchy in brief.

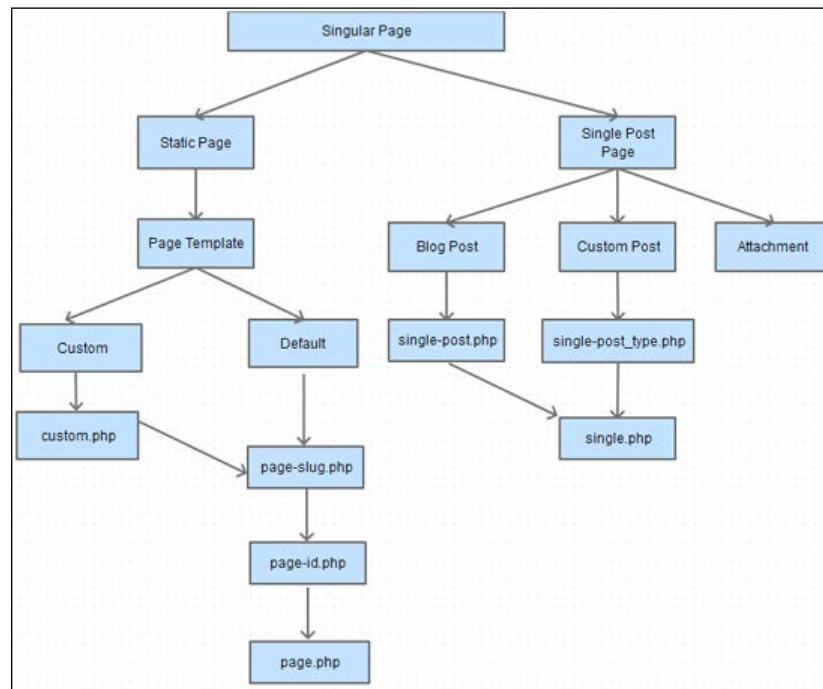


Once the initial request is made, WordPress looks for one of the main starting templates, as illustrated in the preceding screenshot. It's obvious that most of the starting templates such as the front page, comments popup, and index pages are specifically designed for content management systems. In the 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 the existing data in the system.
- **Single pages:** These are used for any type of dynamic content associated with applications. 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:



A singular page is divided into two paths that contain posts or pages. The static page is defined as custom or default page templates. In general, we will 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
*/
?>
```

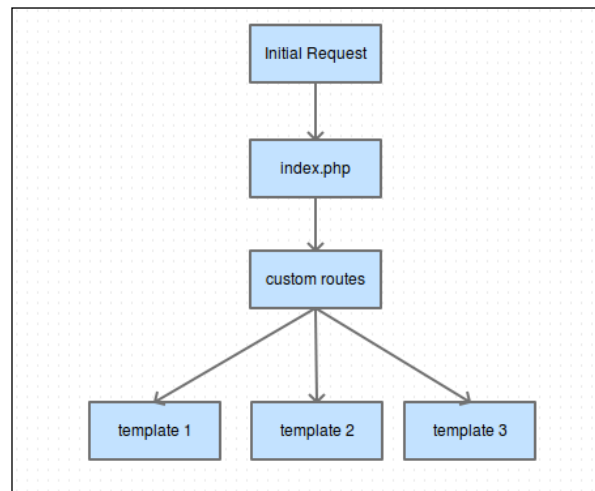
To the right of the preceding diagram, we have a single post page, which is divided into three paths called the blog post, custom post, and attachment post. Both attachment posts and blog posts are designed for blogs and hence will not be used frequently in web applications. However, the custom post template will have a major impact on application layouts. As with a 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.

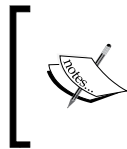
Now, we have had a brief introduction to the template loading process used by WordPress. In the next section, we will look at the template loading process of a typical web development framework to identify the differences.

## The 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:



In this process, the initial request always comes to the `index.php` file, which is similar to the process used by WordPress or any other framework. This process is known as the **Front Controller pattern** in most PHP frameworks. 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, the 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 for 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 `theme` 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 are a wide range for 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 shortcode 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 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 it 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 applications control logic. Let's consider the following scenario to help you understand the issues that occur 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. Later, 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 without even knowing. Now the application's sign-up page is broken, and the users will not be able to use the system. This is the risk of using the preceding techniques where 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

The custom routing technique allows us, as developers, to have complete control over the template generation process. 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 on by inspecting the advanced aspects of custom template techniques.



## 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 each other, thus increasing maintainability and extensibility. In ideal situations, views should have a 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 the 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. 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 called `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 for understanding the use 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. However, 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 the 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 you create each and every template file inside your theme 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 the template inclusion and functionality, as shown in the following code:

```
public function activate_user() {
    // Implementing necessary functions and data generation
    include dirname(__FILE__) . '/templates/info.php';
    exit;
}
```

In this scenario, the `info.php` template has access to all the data generated inside the `activate_user` function. Developers should execute all the business logic in the top section of the `activate_user` function. Even though `info.php` has access to all 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 the template files inside the `themes` folder or `plugins` folder and load it where necessary, making it more flexible compared to the `get_template_part` technique of WordPress.

## Theme versus plugin-based 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 most existing theme templates are used for generating CMS-related functionality, and hence, they will have a 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 reused 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:

- The heavy usage of custom fields
- Removing existing components such as sidebars, comments, and so on
- 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 they 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.

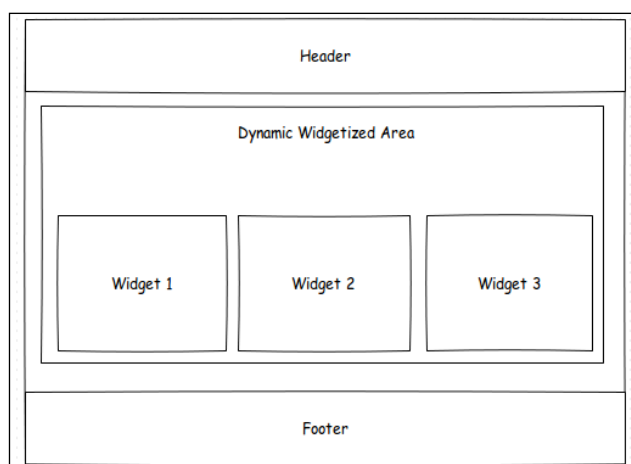
## Building the portfolio application home page

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 will 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 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 will 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 wherever 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 will create the templates inside the `plugins` folder to improve flexibility. So, let's begin the process by creating a file called `class-wpwa-theme.php` within the root directory of our main plugin and including it in the `class-wpwa-portfolio-manager.php` file. Next, we can update the constructor code as follows to register the widgetized area for the home page:

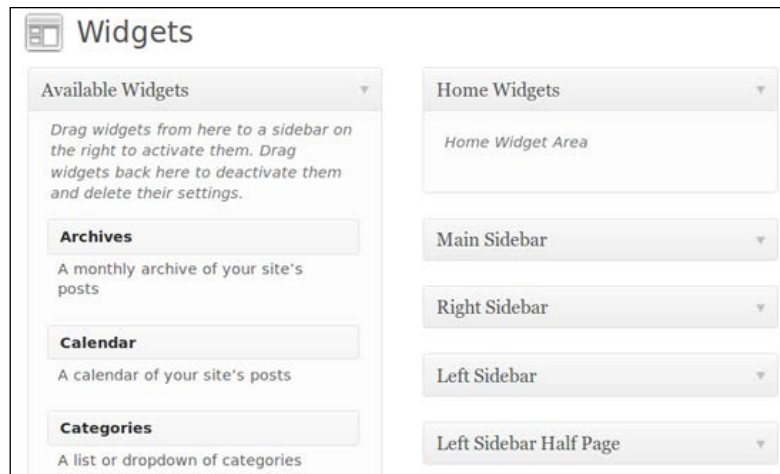
```
class WPWA_Theme {
    public function __construct() {
        $this-> register_widget_areas();
    }
    public function register_widget_areas() {
        register_sidebar(array(
            'name' => __('Home Widgets', 'wpwa'),
            'id' => 'home-widgets',
            'description' => __('Home 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. However, 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 contents.

- `before_title` and `after_title`: These are used to provide additional HTML content before and after the widget 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/class-home-list-widget.php';
    register_widget('Home_List_Widget');
}
```

We will create all three widgets required for the home page using the `class-home-list-widget.php` file inside the `widgets` folder of our `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. However, we will 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 the 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 to 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 of 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
}

```



Let's get started!

1. First, we will check the existing values of the form fields using the `$instance` array. This array will be populated with the existing values of the form fields from the database. Initially, these fields will contain empty values.
2. 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]
```

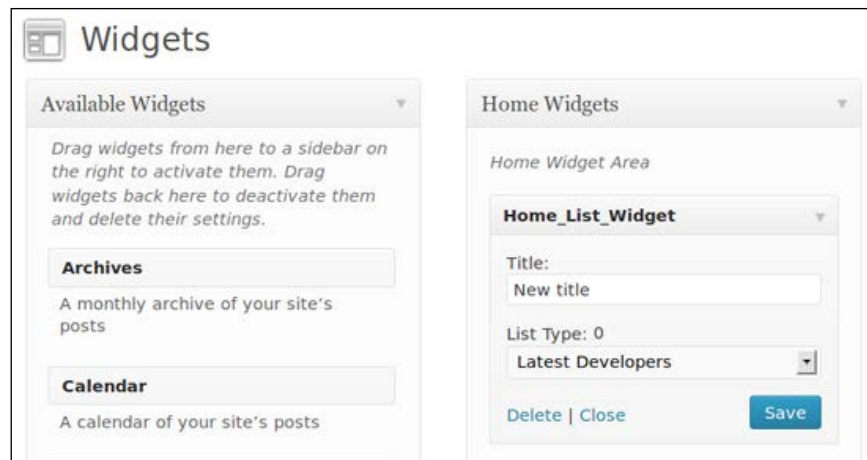
These types of field names and IDs are assigned to make the automation easier. The widgets' data-saving method is completely automated, and hence we have to only define the necessary form fields. Saving the data to the database will be done automatically by WordPress.

3. 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.

4. 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 on 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 construct the developer list for the purpose of this 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. Here, we will use the template loader plugin we developed earlier in this book.

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) {
    global $wpwa_template_loader,$home_list_data;
    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 . $after_title;
    switch ($list_type) {
        case 'dev':
            $user_query = new WP_User_Query(array('role' => 'developer' ,
            'number' => 10));
            $home_list_data = array();
            $home_list_data ["records"] = array();
            foreach ($user_query->results as $developer) {
```

```
        array_push($home_list_data ["records"], array("ID" =>
$developer->data->ID, "title" => $developer->data->display_name));
    }
    $home_list_data ["title"] = $title;
    ob_start();
    $wpwa_template_loader->get_template_part("home","list");
    echo ob_get_clean();
    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 accessed using global variable named `$wpwa_template_loader` for dynamic template loading.
- Then, we check the value of the drop-down field using the switch statement. In the widget form, we included `dev` as the key for developers.
- The other two option values can be found in the source code. Inside the `dev` case section, we query the database to retrieve the recently joined developers in the application using `WP_User_Query`. The user role for developers is used to filter the values.
- Then, we add the generated results into the `$home_list_data` array to pass them into the template.
- Finally, we call the `get_template_part` function of the template loader object by passing the template name as `home-list`. You can create a PHP file named `home-list-template.php` inside the `templates` folder. The implementation of the home page widgets template is given in the following code:

```
<?php global $home_list_data; ?>
<div class='home_list_item'>
    <div class='list_panel'>
        <?php foreach($home_list_data["records"] as $record){
?>
```

---

```

        <div class='list_row'><a href=''><?php echo
        $record['title']; ?></a><?php
        do_action('wpwa_home_widgets_controls',$record['type'],$rec
        ord['ID']); ?></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 a home page template

We have to create a file named `home-template.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 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 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. However, we haven't given instructions to WordPress to load it as the home page. So, let's go to the `WPWA_Theme` class 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 $wp_query,$wpwa_template_loader;
    $control_action = isset ( $wp_query->query_vars['control_action'] ) ? $wp_query->query_vars['control_action'] : '';
    if (is_home () && empty($control_action) ) {
        ob_start();
        $wpwa_template_loader->get_template_part("home");
        echo ob_get_clean();
        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 `get_template_part` 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 on 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 the application 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 the frontend screens as well. In the previous chapter, we looked at various ways of customizing the backend navigation menu. Here, we will look at how the frontend menu works within WordPress.

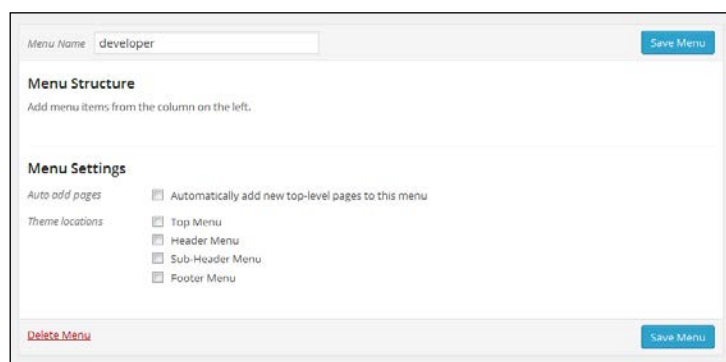
Navigate to the `themes` folder and open the `header.php` file of the Responsive theme. You will find the implementation for the frontend menu using the `wp_nav_menu` function. This function is used to display the navigation menus generated from the **Appearance** section of the WordPress admin dashboard. 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 will 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 in to the portfolio application as the admin and navigate to **Appearance | Menus** to create new frontend menus. Since most of you are familiar with creating menus for websites, I will 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 will 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 developers 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:

The screenshot shows a web interface for configuring a menu. At the top, there is a text input field labeled 'Menu Name' containing the word 'developer', and a blue 'Save Menu' button to its right. Below this is a section titled 'Menu Structure' with a sub-instruction: 'Drag each item into the order you prefer. Click the arrow on the right of the item to reveal additional configuration options.' Under this instruction, there are two items: 'Home' and 'Developers'. Each item has a 'Custom' dropdown menu to its right. Below the 'Menu Structure' section is a 'Menu Settings' section. It contains two groups of checkboxes: 'Auto add pages' with a checkbox for 'Automatically add new top-level pages to this menu', and 'Theme locations' with checkboxes for 'Top Menu', 'Header Menu', 'Sub-Header Menu', and 'Footer Menu'. At the bottom left of the settings section is a red 'Delete Menu' link, and at the bottom right is a blue 'Save Menu' button.

We have to follow this process for other user roles in the portfolio application. Make sure that you 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:

The screenshot shows a menu management interface. On the left, there is a sidebar with a 'Select a menu to edit:' dropdown menu. The dropdown is open, showing a list of menu names: 'visitors', 'developer', 'follower', 'member', and 'visitors'. The 'visitors' menu is selected. To the right of the sidebar, there is a 'Select' button and a link 'or create a new menu.'. Below this, there is a text input field labeled 'Menu Name' containing the word 'visitors', and a blue 'Save Menu' button to its right. The main content area is divided into two sections: 'Menu Structure' and 'Menu Settings'. The 'Menu Structure' section has a sub-instruction: 'Add menu items from the column on the left.' The 'Menu Settings' section contains the same checkboxes as the previous screenshot: 'Auto add pages' with a checkbox for 'Automatically add new top-level pages to this menu', and 'Theme locations' with checkboxes for 'Top Menu', 'Header Menu', 'Sub-Header Menu', and 'Footer Menu'. At the bottom left of the settings section is a red 'Delete Menu' link, and at the bottom right is a blue 'Save Menu' button.



The 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:

+ Options							
		term_taxonomy_id	term_id	taxonomy	description	parent	count
<input type="checkbox"/>			1	1	category	0	8
<input type="checkbox"/>			2	2	technologies	0	0
<input type="checkbox"/>			3	3	technology	0	1
<input type="checkbox"/>			4	4	technology	0	1
<input type="checkbox"/>			9	9	nav_menu	0	0
<input type="checkbox"/>			10	10	nav_menu	0	2
<input type="checkbox"/>			8	8	nav_menu	0	2
<input type="checkbox"/>			11	11	nav_menu	0	1

↑ Check All / Uncheck All With selected:

Finally, we have to look for the 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:

+ Options

			post_title	post_status	post_name	post_type
<input type="checkbox"/>			Developers	publish	developers	nav_menu_item
<input type="checkbox"/>			Home	publish	home	nav_menu_item
<input type="checkbox"/>			Dev	draft		nav_menu_item

Check All / Uncheck All With selected:

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

After 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 | Manage 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 the 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.

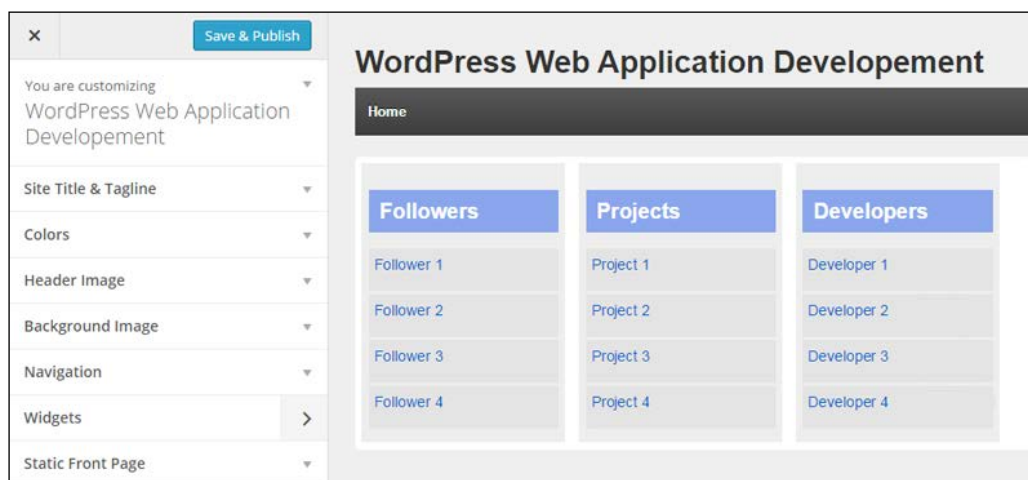
Once logged in, each user will have a frontend menu specific to their user role, and now, we have a basic user role-based menu for a portfolio application.

## Managing options and widgets with customizer

The WordPress theme customizer is a great feature for customizing a theme's settings and components from the frontend. This feature lets you preview those changes in real time and is hence useful for administrators. Generally, customizer is capable of editing the following sections in real time:

- **Site title and Tagline**
- **Colors**
- **Header Image**
- **Background Image**
- **Navigation**
- **Widgets**
- **Static Front Page**

The sections of the customizer are theme-dependent, and hence, you will see more or less sections in various themes. You can access the theme customizer by navigating to the **Appearance | Customize** menu. The following screenshot previews the default customizer screen of the Responsive theme:



All these sections play a part in designing a website with WordPress. However, theme options and widgets are the most important components from the web development perspective, and hence, we will discuss the usage of options and widgets.

## Adding custom options to the theme customizer

Apart from widgets, all other sections mentioned in the previous section are related to theme options. These are built-in options of any given theme. In web applications, we need the custom options panel to configure the features and design. In *Chapter 6, Customizing the Dashboard for Powerful Backends*, we created a simple options page with two options. This options panel can be easily integrated into the theme customizer for real-time customizations and testing. In this section, we will learn how to add custom options to the theme customizer. After this chapter is completed, you can use these techniques to integrate the complete options page into the theme customizer. Let's get started!

Assume that we want to let an admin customize the text color of one of the application-specific components. First, we need to add a custom options section to the theme customizer. Then, we can add the necessary options as settings into the main section. Let's modify the constructor of WPWA\_Theme to add the following actions:

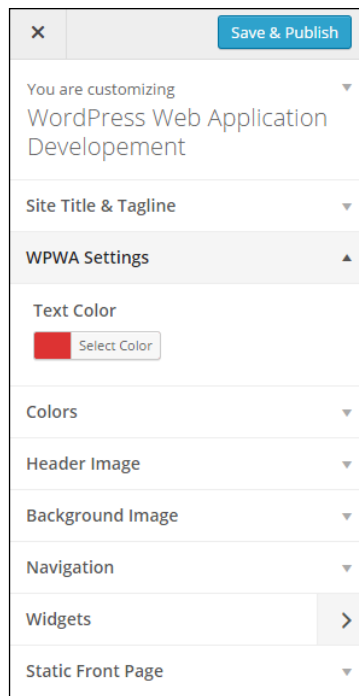
```
add_action( 'customize_register', array($this, 'customize_panel'
));
add_action( 'wp_head', array($this, 'apply_custom_settings'));
```

WordPress uses the `customize_register` action to add custom sections to the theme customizer. Next, we use the `wp_head` action to apply the options to our application. We can start by implementing the `customize_panel` function, as shown in the following code:

```
public function customize_panel( $wp_manager ){
    $wp_manager->add_section( 'wpwa_settings_section', array(
        'title' => __('WPWA Settings','wpwa'),
        'priority' => 35,
    ) );
    $wp_manager->add_setting( 'color_setting', array(
        'default' => '#000000',
    ) );
    $wp_manager->add_control( new WP_Customize_Color_Control(
$wp_manager, 'color_setting', array(
        'label' => __('Text Color','wpwa'),
        'section' => 'wpwa_settings_section',
        'settings' => 'color_setting',
        'priority' => 6
    ) ) );
}
```

First, we add a custom section called **WPWA Settings** for our custom options. This code will add an additional tab to the customizer tabs on the left. Then, we add a setting called `color_setting` to specify the text color of a specific element within the application. Finally, we add an input control for the setting using the `add_control` function. In this scenario, we are using the `WP_Customize_Color_Control` class as we need a color picker. You can find other available input control types at [http://codex.wordpress.org/Class\\_Reference/WP\\_Customize\\_Manager/add\\_control](http://codex.wordpress.org/Class_Reference/WP_Customize_Manager/add_control).

Now, refresh the theme customizer and you will see our new settings section, as shown in the following screenshot:



Once the setting is defined, we have to apply it to the website. Here, we are changing the text color, and hence, our CSS should be updated while changing the color of this setting. Consider the following implementation of the `apply_custom_settings` function to apply the setting to the website:

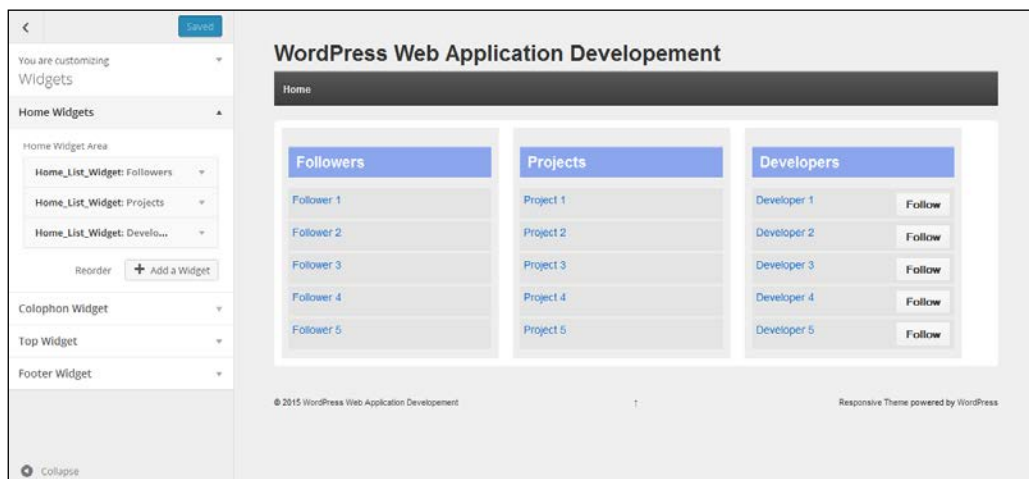
```
function apply_custom_settings() {
    ?>
    <style>
    body {
        color: <?php echo get_theme_mod(
        'color_setting' ); ?>;
    }
```

```
}  
</style>  
<?php  
}
```

We can get the value of settings by passing the settings field key to the `get_theme_mod` function. In this example, we are applying the color to the page `<body>` tag. Ideally, you should target application-specific design elements using these settings. Now, you can change the text color from our new settings control and the font color of the page will be changed in real time. This technique can be used to add advanced settings panels into the theme customizer for simplifying the customizing process and saving time on backend operations.

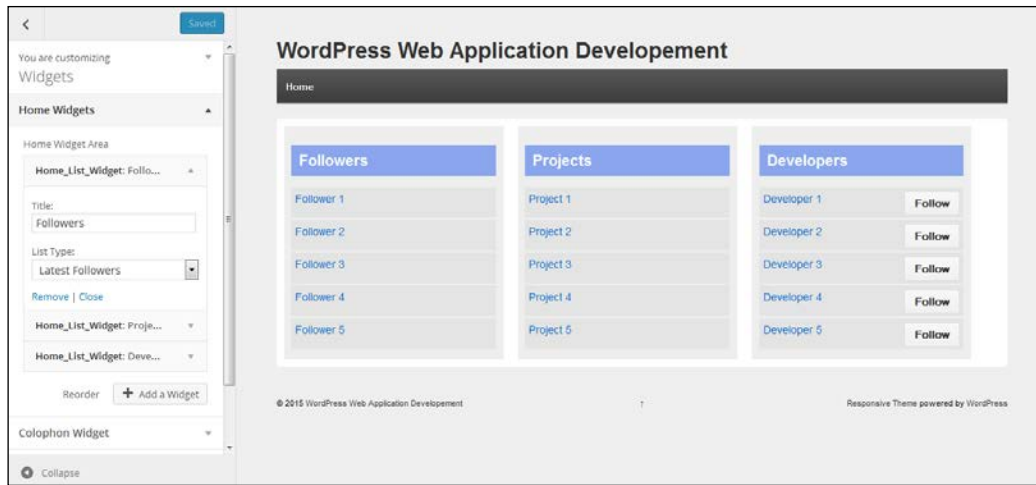
## Handling widgets in the theme customizer

Widgets are heavily used in web application development, and hence, we need to know how to use them within the theme customizer. Also, widgets are the only non-design-specific component in the theme customizer. We can access the widgets by clicking on the **Widgets** tab in the customizer panel. The following screenshot previews the default widget areas of our application:

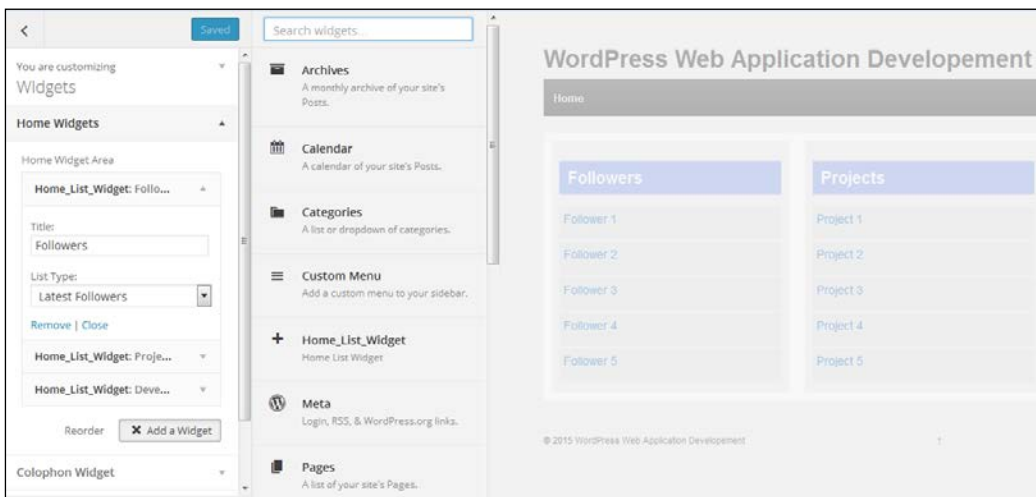


We created a custom widget area called **Home Widgets**, and it contains the three widgets assigned in the previous sections. Also, you can see some other widget areas available by default in the Responsive theme. Changing widget configurations is quite a simple task. Let's look at the following steps:

1. Click on the arrow on the right side to open the **Followers** widget. You will get a screen similar to following one with the available widget options:



2. Now, change the title or list type values. Here, we have changed the title to **Followers List** and it's immediately applied on the widget on the right-hand side of the page. Similarly, we can check any type of advanced widget settings and test our widgets until we get the expected output. In this scenario, we are working with an application-specific widget area, and hence, you can see the changes on the page. However, normal widgets are displayed on the sidebar of the page. So, you need to open a normal post/page with the sidebar before using the customizer options.
3. The last task in this section is to identify how we can add new widgets into widget areas without using the backend functionality. You can click the **Add a Widget** button and it will open up a screen similar to the following one:



We can search the widgets from the available list and click on it to add to the selected widgets area.



All the changes made through the theme customizer are temporary for previewing purposes and will not be saved until you click the **Save and Publish** button.

We discussed the importance of the theme customizer and how we can adapt and customize it for web applications. Now, it's time for you to integrate the application options panel into the theme customizer.

## Creating pluggable 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. The 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.

These types of templates are highly adaptable and flexible for the future enhancements or modification of existing features. 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 reusable templates work in WordPress and normal web applications. However, the final output is quite similar in nature. Let's find out how pluggable templates are used in web applications before digging into WordPress.

## Pluggable templates in WordPress

WordPress uses a hook-based architecture for adding new functionalities to the 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 a new behavior to existing templates. However, at present, they are 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 a developer 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 used only 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.

## 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. The 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 to the widget by modifying the existing code. As a developer, 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

The following are the steps to customize widgets to enable extendable locations:

1. 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.php` file inside the `templates` folder and replace the existing code with the following code:

```
<div class='home_list_item'>
  <div class='list_panel'>
    <?php foreach($home_list_data ["records"] as $record){
    ?>
      <div class='list_row'>
        <a href=''><?php echo
$record['title']; ?></a>
        <?php
do_action('wpwa_home_widgets_controls',$record['type'],$rec
ord['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.

2. Next, we have to change the widget code to pass the type parameter. We must pass a value for the 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.
3. 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 2 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 functionality 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 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 later. These hooks are mainly placed before and after the main content of the templates. This technique works well for designing themes for websites. However, 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. However, we can only add new content before and after the menu container. There is no way to change 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 hardcoded 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 match exactly 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 discussed some of the advanced techniques in WordPress themes. Developers who don't have any exposure to advanced application development with WordPress might find it a bit difficult to understand these techniques. Therefore, I suggest that you try out the following tasks to get familiar with the advanced theme creation techniques:

- Automate the process of frontend menu item creation. The admin should be able to add menu items to multiple navigation menus in a single event or the 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 that you 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's 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 talk about extending the capabilities of the WordPress theme files using the 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 will 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. Underscore.js, 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 discuss 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 WordPress core
- Open source JavaScript libraries in 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 OpenAuth
- 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 one hand, they are completely free and allow developers to customize and create their own versions, while on the other, 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:

- A large community support
- The 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.

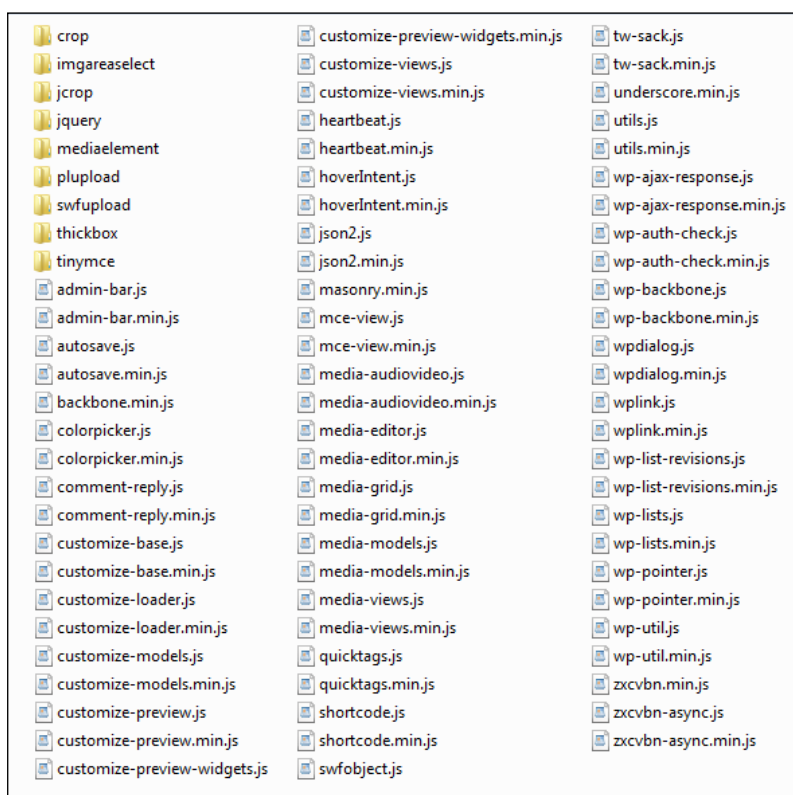
As we move into web application development, we should understand the need for using existing libraries whenever possible due to the following reasons:

- WordPress contains libraries that are more stable and compatible
- This also prevents the duplication of libraries and reduces the size of the project files

I am sure you are 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/js` folder of your WordPress installation. Let's take a look at the following screenshot for JavaScript libraries available with WordPress 4.1:



As you can see, there are 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 will 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 to easily integrate 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 already discussed the importance of separating the concerns in web application development throughout the first few chapters of this book. The MVC architecture is heavily used in server-side implementation for separating the concerns. However, 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 to learn 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:

Project Name	Status	Duration
Project 1	planned	23
Project 2	planned	12
Project 3	planned	12

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 with the server
- The 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 a specific functionality such 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](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 will implement the mentioned tasks with the use of Backbone.js and Underscore.js. As usual, we will update our main portfolio management plugin with the necessary code. Create a class called `class-wpwa-backbone-projects.php` inside the root folder of our plugin. Then, we can include the class inside `class-wpwa-portfolio-manager.php` using the `require_once` statement.


Our first task will be to load the personal information for developers. First, we need some custom rewrite rules to handle the routing for the Backbone.js-based projects screen. Let's start by creating the necessary action hooks to handle the custom routing. We already have a custom route handler function called `manage_user_routes` in our main plugin. Let's update the function to include the new custom rewrite rules, as shown in the following code:

```

public function manage_user_routes() {
    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' );
}

```

In this scenario, we need another rewrite rule for working with the ID parameter. Hence, we have introduced another rule to match the `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 reused across all the editor load functions by ID.

 The developer profile can be accessed by using /user/profile/ID where ID is the unique ID in the wp\_users table.

We have to also register a new query variable for record\_id. So, we can update our existing manage\_user\_routes\_query\_vars function, as shown in the following code:

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

Next, we have to load the profile page through a custom controller. We already have a controller in our main plugin inside the front\_controller function. Let's update the existing front\_controller function to include the new control action for this section:

```
public function front_controller() {  
    global $wp_query,$backbone_projects;  
    $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;  
        case 'login':  
            do_action( 'wpwa_login_user' );  
            break;  
        case 'activate':  
            do_action( 'wpwa_activate_user' );  
            break;  
        case 'profile':  
            $record_id = isset($wp_query->query_vars['record_id']) ?  
$wp_query->query_vars['record_id'] : '' ;  
            $developer_id = $record_id;  
            $backbone_projects->create_developer_profile($developer_id);  
            break;  
        default:  
            break;  
    }  
}
```

This is the same technique we used in *Chapter 2, Implementing Membership Roles, Permissions, and Features*, for working with custom routing. We will introduce a new control action called `profile`. Here, we have an additional query parameter for the developer ID named as `record_id`. Finally, we can look at the `create_developer_profile` function of the `WPWA_Backbone_Projects` class for generating developer profile data:

```
public function create_developer_profile($developer_id) {
    global $project_data,$wpwa_template_loader;
    $user_query = new WP_User_Query(array('include' =>
array($developer_id)));
    $project_data = array();
    foreach ($user_query->results as $developer) {
        $project_data['display_name'] = $developer->data-
>display_name;
    }
    $current_user = wp_get_current_user();
    $project_data['developer_status'] = ($current_user->ID ==
$developer_id);
    $project_data['developer_id'] = $developer_id;
    $wpwa_template_loader->get_template_part("developer");
    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 will assign the name to the `$project_data` array to be passed into the template.



In the final chapter, we will update the developer profile with additional information to be displayed in the frontend.

Next, we will check whether the developer of this profile is logged into the application to show or hide the **Add New** button on the projects screen. Finally, we render the developer template with the `$wpwa_template_loader` global object. This object is provided by the reusable template loader plugin we created earlier in this book. In order to complete the initial page loading, we need to create the `developer-template.php` template inside `templates` folder.

Here, are the initial contents of the `developer.php` template:

```
<?php
    global $project_data;
    get_header();
?>
```



```
<div class='main_panel'>
  <div class='developer_profile_panel'>
    <h2> <?php echo __('Personal Information','wpwa'); ?> </h2>
    <div class='field_label'><?php echo __('Full Name','wpwa');
?> </div>
    <div class='field_value'><?php echo esc_html($project_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,
        'nameRequired' => __('Project name is required','wpwa'),
        'statusRequired' => __('Status is required','wpwa'),
        'durationRequired' => __('Duration is required','wpwa'),
    );
    wp_localize_script('developerjs', 'wpwaScriptData',
    $config_array);
}
```

We create a new JavaScript file called `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. Later, we will 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 `Project` and collection will be `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 a model, as shown in the following code:

```
$jq =jQuery.noConflict();
$(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 will 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 basics of Backbone.js. So, I suggest that 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 project list for developers:

```
var projectsList;
var ProjectListView = Backbone.View.extend({
  el: $('#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><?php echo __('Projects','wpwa'); ?></h2>
  <div >
    <table id='list_projects'>
    </table>
  </div>
</div>
```

As defined in the view, this will be the main container used for displaying 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 necessary function, as shown in the following code:

```
var ProjectListView = Backbone.View.extend({
  el: $('#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, let's update the constructor of the `WPWA_Backbone_Projects` class 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 `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 raw data of 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>.

The backbone `fetch` function uses the GET request with no parameters, and hence, `$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 called `#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. So, let's take a look at the two templates stored inside the `developer-template.php` file:

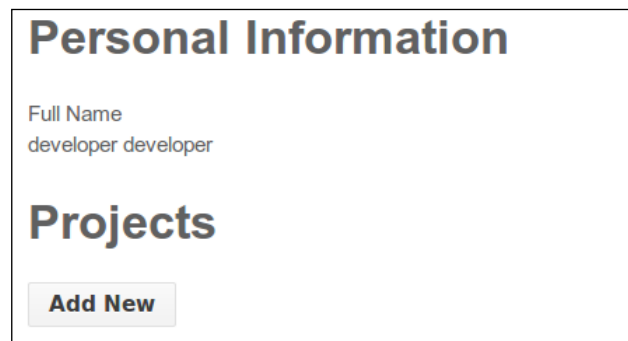
```
<script type="text/template" id="project-list-template">
  <% _.each(projects, function(project) { %>
    <tr class="project_item">
      <td><%= project.get('name') %></td>
      <td><%= project.get('status') %></td>
      <td><%= project.get('duration') %></td>
    </tr>
  <% }); %>
</script>
<script type="text/template" id="project-list-header">
  <tr >
    <th><?php echo __('Project Name','wpwa'); ?></th>
    <th><?php echo __('Status','wpwa'); ?></th>
    <th><?php echo __('Duration','wpwa'); ?></th>
  </tr>
</script>
```

Using `text/template` as the `script type` parameter is a common way of defining templates inside HTML files. The browser doesn't execute these client side scripts due to the `text/template` type. Here, we have two templates, where first one generates the list of projects using 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 necessary data. Then, we use the `e1` element of the view to find the `#list_projects` table, and populate the output generated from the templates as the content using `html` function. Now, we will get the data grid populated with the project list on page upload. So far, we have discussed the usage of 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 will 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-template.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="<?php
    __('Add New', 'wpwa'); ?>" />
<?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-template.php` template after the preceding code:

```
<div id='pro_add_panel' style='display:none' >
  <div class='field_row'>
    <div class='field_label'><?php echo __('Project Name','wpwa');
?></div>
    <div class='field_value'><input type='text' id='pro_name'
/></div>
  </div>
  <div class='field_row'>
    <div class='field_label'><?php echo __('Status','wpwa');
?></div>
    <div class='field_value'>
      <select id="pro_status">
        <option value="0"><?php echo __('Select','wpwa');
?></option>
        <option value="planned"><?php echo __('Planned','wpwa');
?></option>
        <option value="pending"><?php echo __('Pending','wpwa');
?></option>
        <option value="failed"><?php echo __('Failed','wpwa');
?></option>
        <option value="completed"><?php echo
__('Completed','wpwa'); ?></option>
      </select>
    </div>
  </div>
  <div class='field_row'>
    <div class='field_label'><?php echo __('Duration','wpwa');
?></div>
    <div class='field_value'><input type='text' id='pro_duration'
/></div>
  </div>
  <div class='field_row'>
    <div class='field_label'><input type='hidden'
id='pro_developer' value='<?php echo
$project_data['developer_id']; ?>' /></div>
    <div class='field_value'><input type='button' id='pro_create'
value='<?php echo __('Save','wpwa'); ?>' /></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 by 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 will use Backbone.js events to structure the code properly.

## Integrating events to 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` variable 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 called `events` inside the view. We have used a click event on `#add_project` and `#pro_create` to call `addNewProject` and `saveNewProject` functions respectively. The `addNewProject` function uses the jQuery `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:



The screenshot shows a web form titled "Projects". At the top left is a button labeled "Add New". Below this button are three input fields: "Project Name" (a text box), "Status" (a dropdown menu with "Select" as the current value), and "Duration" (a text box). At the bottom left is a button labeled "Save".



We used several rewriting rules with a similar structure throughout this book. So, it's important to keep the rewrite rules in 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 for 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 = wpwaScriptData.nameRequired;
    if (attrs.status == 0) errors.status =
wpwaScriptData.statusRequired;
    if (!attrs.duration) errors.duration =
wpwaScriptData.durationRequired;
    if (!_.isEmpty(errors)) {
      console.log(errors);
      return errors;
    }
  }
});
```

The validate function is automatically executed before saving data to server, by passing the attributes of the model as parameter. We can execute the necessary validation within this function and return the errors as an object.

## Creating new models in the server

This is the final section of the process for saving new projects to 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 called `options` for defining success and failure functions for the project creation. Here, we are logging the result values to browser console. In real implementations, the result should be displayed to the user as a message.

Then, 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 `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 a new project to database. This will generate a POST request to the AJAX action handler called `wpwa_process_projects`. Finally, we complete the process by implementing the rest of the `process_projects` function, as shown in the following code:

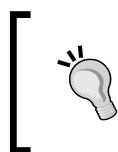
```
public function process_projects() {
    $request_data = json_decode(file_get_contents("php://input"));
    $project_developer = isset($_GET['developer_id']) ?
    $_GET['developer_id'] : '0';
    if (is_object($request_data) && isset($request_data->name)) {
        $project_name = isset($request_data->name) ?
        $request_data->name : '';
        $project_status = isset($request_data->status) ?
        $request_data->status : '';
        $project_duration = isset($request_data->duration) ?
        $request_data->duration : '';
        $err = FALSE;
        $err_message = '';
        if ($project_name == '') {
            $err = TRUE;
            $err_message .= __('Project name is required.','wpwa');
        }
        if ($project_status == '0') {
            $err = TRUE;
            $err_message .= __('Status is required.','wpwa');
        }
        if ($project_duration == '') {
            $err = TRUE;
            $err_message .= __('Duration is required.','wpwa');
        }
        if ($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 server. Now, we will 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 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 data is successfully validated, we create a new project using the `wp_insert_post` function. Once the project is successfully saved, we can execute `update_post_meta` functions 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 WordPress frontend. In the last part, we learned the uses of Backbone.js `create` and `validate` functions plus handling effects.



Keep in mind that edit and delete functionalities 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 look 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 will 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 called `class.phpmailer.php`.

## Usage of PHPMailer within the WordPress core

Basically, the functionality of this library is invoked using a common function called `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 to 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`: Changing the e-mail of the sender, which defaults to `wordpress@{sitename}`

- `wp_mail_from_name`: Changing the name of the sender, which defaults to WordPress
- `wp_mail_content_type`: Changing the e-mail content type, which defaults to text/plain
- `wp_mail_charset`: Changing the e-mail character set, which defaults to charset assigned in settings

In web applications, we might need advanced customizations that 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, as listed here:

- Creating a custom version of the pluggable `wp_mail` function
- Loading PHPMailer inside plugins and creating custom functions

## Creating a custom version of a pluggable `wp_mail` function

We briefly introduced pluggable functions in *Chapter 5, Developing Pluggable Modules*. These functions are located inside the `pluggable.php` file 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 existing function within a plugin and change the code as 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 hence, 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 will implement the custom PHPMailer function while implementing the subscriber notification process of portfolio management application. Let's list the tasks for this process:

- Creating custom function to use PHPMailer
- Sending e-mails in SMTP using an authenticated account
- Retrieving the subscribers from the database
- Sending notifications on new projects, articles, books, and services

We can start the process by creating a new class called `WPWA_Open_Source` in a file called `class-wpwa-open-source.php`. Then, include the following action inside the constructor of the `WPWA_Open_Source` class:

```
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 the post status transfer. Subscribers should get notifications about all the projects, articles, books, and services, once they are published. So, we use the above post status transfer actions to execute a custom function on 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 = "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->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, 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. So, we start by defining 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. The 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 the 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 large amount 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 notification 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 will not 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 OAuth

Login 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. Let's take a look at the definition of OAuth by Wikipedia:

*"OAuth is an open standard for authorization. OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner, or end-user. The client then uses the access token to access the protected resources hosted by the resource server."*

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 the OpenAuth login into WordPress. Here, we will upgrade our plugin to let users log in and register through popular social networking sites such as Twitter, Facebook, and LinkedIn. We will use third-party OAuth connection libraries to build this functionality as creating a OAuth library from scratch, which is beyond the scope of this book.

Now, we are ready for implementing the OpenAuth login integration for our portfolio application. The 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 into the login screen:

1. Directly embed the HTML code under the default login button.
2. Define an action hook and implement the hook within a plugin.

Even though both techniques does the same job, we have more 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 `login-template.php` file inside the `templates` folder of our main plugin. The following code previews the last part of login template with the new action for social logins:

```
<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" >&nbsp;</label>
    <input type="submit" name="submit" value="Login" />
  </li>
</ul>
</form>
<?php do_action('wpwa_social_login'); ?>
```

Here, we execute an action called `wpwa_social_login` with the `do_action` function. This allows the possibility of adding dynamic content to the login screen using plugins. Next, we have to generate the necessary login links to populate the `wpwa_social_login` area. Create a new class called `WPWA_Social` in a file called `class-wpwa-social.php` for including the functions related to social login links. The first task is to create the login links for different social networks and add them to the login form. Let's look at the initial version of `WPWA_Social` class:

```
<?php
class WPWA_Social{
  public function __construct(){
    /* Add the social login buttons to the registration and login
    forms based on the settings */
```

---

```

        add_action('wpwa_social_login',
        array($this, 'wpwa_social_login_buttons'));
    }
    public function wpwa_social_login_buttons($html){
        $allowed_networks = array('Twitter', 'Linkedin', 'Facebook');
        if (get_option('users_can_register') == '1') {
            $html = '<div align="center" style="margin:10px">';
            foreach ($allowed_networks as $key => $network) {
                $link =
                '?wpwa_social_login=' . $network . '&wpwa_social_action=login';
                $html .= '<a class="wpwa-social-link" href="' . $link .
            "" >
                Login with ' . $network . '
            </a>';
            }
            $html .= '</div>';
        }
        echo $html;
    }
}
$social_obj = new WPWA_Social();

```

We have implemented the `wpwa_social_login` action with the use of `wpwa_social_login_buttons` function inside the `WPWA_Social` class. The simplicity of the HTML code made me to output it using an `echo` statement. Ideally, we should be using template parts for loading the display code. The `href` attribute contains the action to be executed and the name of the social network. We will need both parameters for the upcoming implementation.

## Configuring login strategies

We need to implement the login strategy for each social network. So, we need a base class for handling common functionality for all the social networks. Let's create a new class called `WPWA_Social_Connect` inside a file called `class-wpwa-social-connect.php`. This file needs to be created in the root of our `plugins` folder. As usual, we need to include this file inside `class-wpwa-portfolio-manager.php` using a `require_once` statement. The following code contains the initial version of `WPWA_Social_Connect` class:

```

<?php
class WPWA_Social_Connect{

```

```
public function callback_url() {
    $url = 'http://' . $_SERVER["HTTP_HOST"] .
    $_SERVER["PHP_SELF"];
    if (strpos($url, '?') === false) {
        $url .= '?';
    } else {
        $url .= '&';
    }
    return $url;
}
public function redirect($redirect) {
    wp_redirect($redirect); exit;
}
public function register_user($result) {}
}
```

These are the most common functions for all the social networks at this stage. However, we might need additional functions as we cater to advanced requirements of social login. First, we have a function called `callback_url` for defining the return page after completing the authentication with the social network. We use the current page URL as the callback URL. Then, we have a generic function for making the redirections using the `wp_redirect` function. Finally, we have the `register_user` function. This function will be used to either register a new user or login existing user, after successful authentication with social network. Now, we have the base functionality to implement the social login.

In this chapter, we will implement a social login with LinkedIn. We omit other networks, as the process is similar. So first, we need to find an OAuth library and LinkedIn API library. The OAuth library provided by <https://code.google.com/p/oauth/> is the popular choice among developers. You can select the PHP version and grab the `OAuth.php` class. Since it's a library, we will create a new folder called `lib` inside our main plugin. We can place the `OAuth.php` class inside the `lib` folder and will require it in our main plugin file as usual.

Then, we can grab a library for LinkedIn API functions at <https://code.google.com/p/simple-linkedinphp/>. This is the most popular and simplest choice. You can download the library and copy the `linkedin_3.2.0.class.php` file into `lib\LinkedIn` folder. As usual, we need to require this file from our main plugin file.

Now, we are ready for the implementation of social login with LinkedIn. We created a class called `WPWA_Social` for all the common functionality for social login. So, we need to create a class for each social network and extend the `WPWA_Social` class for the common functionality. Let's create a new class called `WPWA_LinkedIn_Connect` in a file called `class-wpwa-linkedin-connect.php`. This file needs to be included after the `Oauth.php` and `linkedin_3.2.0.class.php` files. First, we need to identify the functionality of this class as follows:

- This class should redirect the user to the respective social network to authorize the account with our application
- Once account is authorized and redirected to our site, we need to verify the account details and log in the user
- We need to handle any errors generated from the social network

We will implement these features one by one.

## Implementing LinkedIn account authentication

Let's start with redirecting to LinkedIn and authorizing the application. The following code shows the initial implementation of the `WPWA_LinkedIn_Connect` class with the `login` function:

```
<?php
class WPWA_LinkedIn_Connect extends WPWA_Social_Connect{
    public function login(){
        $callback_url = wpwa_add_query_string($this->callback_url(), 'wpwa_social_login=Linkedin&wpwa_social_action=verify');
        $wpwa_social_action = isset($_GET['wpwa_social_action']) ? $_GET['wpwa_social_action'] : '';
        $response = new stdClass();
        /* Configuring settings for LinkedIn application */
        $app_config = array(
            'appKey' => 'app key',
            'appSecret' => 'app secret',
            'callbackUrl' => $callback_url
        );
        @session_start();
        $linkedin_api = new LinkedIn($app_config);
```

```
        if ($wpwa_social_action == 'login'){
            /* Retrive access token from LinkedIn */
            $response_linkedin = $linkedin_api-
>retrieveTokenRequest(array('scope'=>'r_emailaddress'));
            if($response_linkedin['success'] === TRUE) {
                /* Redirect the user to LinkedIn for login and
authorizing the application */
                $_SESSION['oauth']['linkedin']['request'] =
                $response_linkedin['linkedin'];
                $this->redirect(LINKEDIN::_URL_AUTH .
                $response_linkedin['linkedin']['oauth_token']);
            }else{
                // Handle Error
            }
        }
    }
    return $response;
}
```

We start the implementation by extending the `WPWA_Social_Connect` class. Consider the first three lines of the `login` function. We add the `wpwa_social_login` and `wpwa_social_action` parameter to the callback URL generated from the parent class. Here, we have used a custom function called `wpwa_add_query_string`, to add the query variables to the callback URL. Implementation of the `wpwa_add_query_string` function can be found inside the `functions.php` file of the main plugin. Then, we assign the action to the `$wpwa_social_action` variable and create a `stdClass` object to handle the response.

Afterwards, we have to define the LinkedIn app key and secret in the `$app_config` array, along with the callback URL. We will cover more details on the app key and secret in the next section on *Building a LinkedIn app*.

Now, we can start the process of authenticating user account with our application. First, we start a new session using `session_start` to hold the data retrieved from LinkedIn. Then, we initialize the third-party LinkedIn class by passing the `$app_config` array. We have to check for the proper action using `$wpwa_social_action` as we have multiple actions in this process. If the action is `login`, we call the `retrieveTokenRequest` function of LinkedIn API class with a scope called `r_emailaddress`. You can learn more about scope in LinkedIn API at <https://developer.linkedin.com/documents/authentication>.

Then, we save the information to session and redirect the user to LinkedIn on a successful response from the `retrieveTokenRequest` function execution. If this function generates any errors, we need to handle it using a custom function. We will be omitting the error handling part considering the scope of this chapter. You will find complete implementation on the book website. After redirecting to LinkedIn, the user can authenticate the application with their LinkedIn account. Once the authentication is completed, the request will be redirected to our application using the callback URL. So, we need to start the implementation to verify account details and log in the user, as we discussed in point 2 on the functionality of the `WPWA_LinkedIn_Connect` class.

## Verifying LinkedIn account and generating response

We created an `if` statement in the previous code to check for the availability of the login action. Now, we need to extend it with the `else if` statement to check for the response from LinkedIn. The following code contains the implementation of account verification process:

```
elseif(isset($_GET['oauth_verifier'])) {
    $response_linkedin = $linkedin_api-
>retrieveTokenAccess($_SESSION['oauth']['linkedin']['request']['oauth_token'],
$_SESSION['oauth']['linkedin']['request']['oauth_token_secret'],
$_GET['oauth_verifier']);
    if($response_linkedin['success'] === TRUE) {
        $linkedin_api-
>setTokenAccess($response_linkedin['linkedin']);
        $linkedin_api-
>setResponseFormat(LINKEDIN::_RESPONSE_JSON);
        $user_result = $linkedin_api->profile('~:(email-address,id,first-name,last-name,picture-url)');
        if($user_result['success'] === TRUE) {
            /* setting the user data object from the response */
            $data = json_decode($user_result['linkedin']);
            $response->status = TRUE;
            $response->wpwa_network_type = 'linkedin';
            $response->first_name = $data->firstName;
            $response->last_name = $data->lastName;
```



```
        $response->email    = $data->emailAddress;
        $response->username  = $data->emailAddress;
        $response->error_message = '';
    }else{
        /* Handling LinkedIn specific errors */
    }
}
}else{
    /* Handling LinkedIn specific errors */
}
}
```

The LinkedIn API redirects the request to our application with a URL parameter called `oauth_verifier`, and hence, we check the existence of the parameter before proceeding. Once its set, we call the `retrieveTokenAccess` function of API class with session parameters and the value of `oauth_verifier`. This function verifies the request and requests the users access token from the LinkedIn API. Once successful, the response is returned and we call the `setTokenAccess` and `setResponseFormat` functions of API class. Having completed the verification process, we can now request the user details using the access tokens generated earlier. So, we execute the `profile` function of the API class with necessary information such as `email-address`, `id`, `first-name`, `last-name`, `picture-url`, and so on. Once the profile information request is successful, we assign the necessary profile data to our response object.



We have omitted all the error handling conditions in this function. You can use the book website to look at the implementation of handling errors.

We have completed the process requesting profile information from LinkedIn. However, we didn't create a LinkedIn app for our application. Now, let's build an app to generate these keys for LinkedIn. The app creation for these social networking sites mentioned here is explained commonly in many online resources. Hence, we will create a LinkedIn app to define a new strategy.

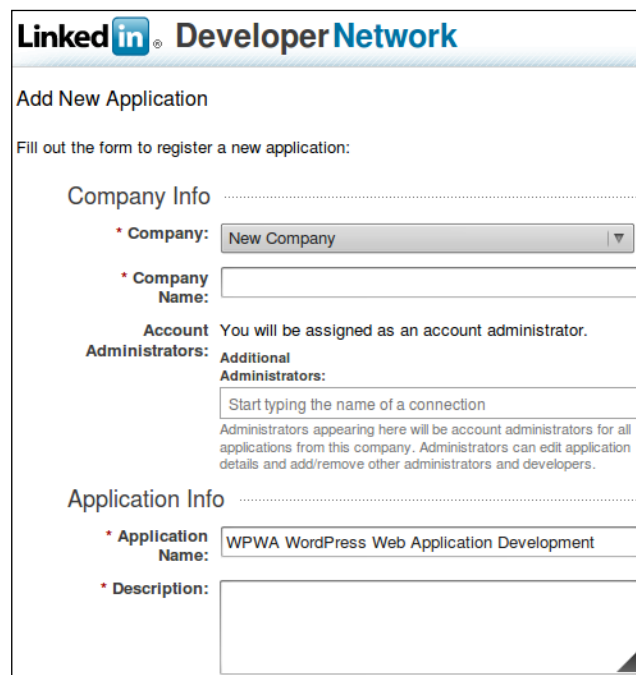
Those who are not familiar with the creation of Twitter, Facebook, and Google+ apps can use the resource section for this chapter in the book's website for step-by-step app creations for each of these strategies.

## Building a LinkedIn app

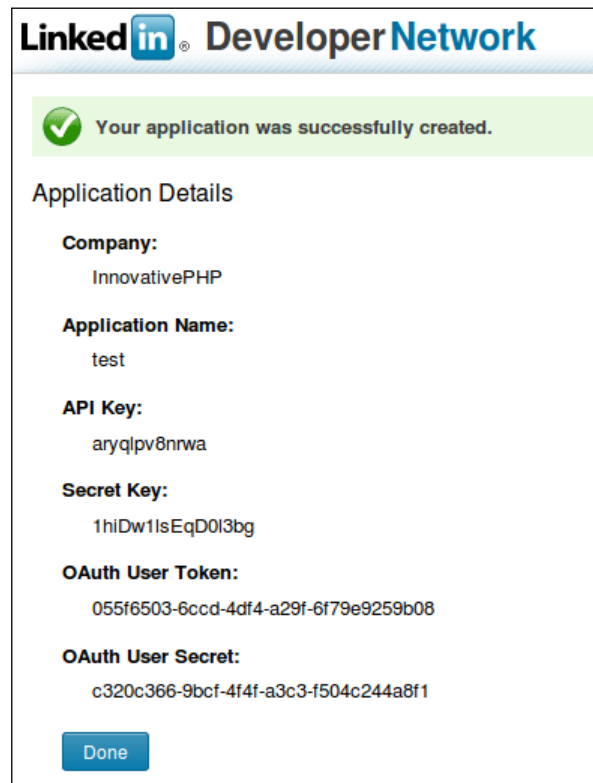
First, you have to log in to your existing LinkedIn account. Then, you have to visit the developer section using <https://www.linkedin.com/secure/developer>. You will get a screen similar to following with the existing apps, if there are any:



You can click on the **Add New Application** link to get the following screen:

The screenshot shows the "Add New Application" form on the LinkedIn Developer Network page. The form is titled "Add New Application" and includes the instruction "Fill out the form to register a new application:". The form is divided into two main sections: "Company Info" and "Application Info".  
**Company Info:**  
- \* Company: A dropdown menu with "New Company" selected.  
- \* Company Name: A text input field.  
**Account Administrators:**  
- You will be assigned as an account administrator.  
- Additional Administrators: A text input field with the placeholder "Start typing the name of a connection".  
- A note below the field states: "Administrators appearing here will be account administrators for all applications from this company. Administrators can edit application details and add/remove other administrators and developers."  
**Application Info:**  
- \* Application Name: A text input field with "WPWA WordPress Web Application Development" entered.  
- \* Description: A large text area for the application description.

Make sure that 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 hit the **Add Application** button, you will get a screen similar to the following one with all the application specific details, including **API Key** and **Secret key**:



The screenshot shows the LinkedIn Developer Network interface. At the top, there's a header with the LinkedIn logo and 'Developer Network'. Below this is a green success message: 'Your application was successfully created.' with a green checkmark icon. The main section is titled 'Application Details' and contains the following information:

- Company:** InnovativePHP
- Application Name:** test
- API Key:** aryqlpv8nrwa
- Secret Key:** 1hiDw1lsEqD0l3bg
- OAuth User Token:** 055f6503-6ccd-4df4-a29f-6f79e9259b08
- OAuth User Secret:** c320c366-9bcf-4f4f-a3c3-f504c244a8f1

At the bottom of the details section is a blue button labeled 'Done'.

Now, we can move back to the configuration array in the `login` function and include the LinkedIn configuration details, as shown in the following code:

```
$app_config = array(  
    'appKey' => 'h8274rwoerp1',  
    'appSecret' => 'W7xE1KINoDajgl0a',  
    'callbackUrl' => $callback_url  
);
```

We have completed the implementation of the `WPWA_LinkedIn_Connect` class. Now, we need to initialize this process and log in the users into our application.

## The process of requesting the strategies

We have the login links assigned to the login screen with href values as Facebook, Twitter, and LinkedIn. The URL of the login screen is `http://www.yoursite.com/user/login/`, and once you click the link for LinkedIn, you will get a URL such as `http://www.yoursite.com/user/login/?wpwa_social_login=Linkedin&wpwa_social_action=login`. The verification and authentication process will be processed based on these parameters.

## Initializing the library

We have implemented all the functionality for authenticating users from LinkedIn, using `WPWA_LinkedIn_Connect` and `WPWA_Social` classes. However, nothing will work yet as we haven't initialized the social login library class. So, we have to implement the initialization code inside the `WPWA_Social` class. Let's start the process by including a `wp_loaded` action to execute the initialization. The following code displays the modified constructor of the `WPWA_Social` class:

```
public function __construct(){
    add_action('wp_loaded', array($this,
    'wpwa_social_login_initialize'));
    add_action('wpwa_social_login', array($this,
    'wpwa_social_login_buttons'));
}
```

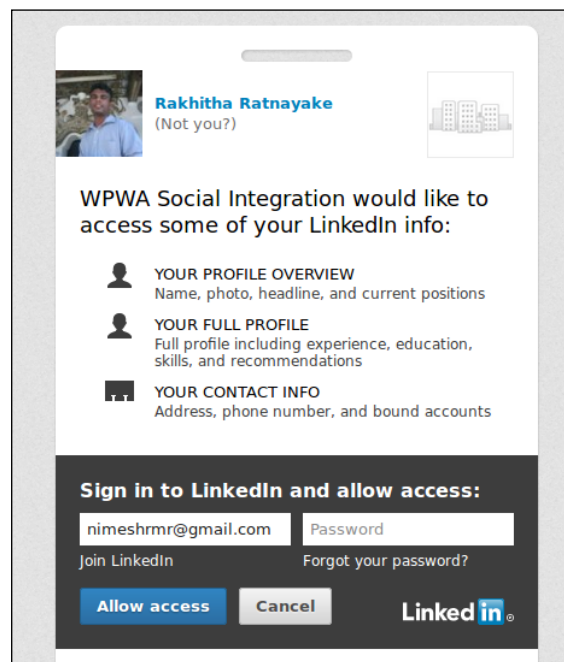
Next, we need to implement the `wpwa_social_login_initialize` function, as shown in the following code:

```
public function wpwa_social_login_initialize(){
    $wpwa_social_login_obj = false;
    $wpwa_social_login = isset($_GET['wpwa_social_login']) ?
    $_GET['wpwa_social_login'] : '';
    $wpwa_social_action = isset($_GET['wpwa_social_action']) ?
    $_GET['wpwa_social_action'] : '';
    if('' != $wpwa_social_login ){
        switch ($wpwa_social_login) {
            case 'Linkedin':
                $wpwa_social_login_obj = new
                WPWA_LinkedIn_Connect();
                break;
            default:
                break;
        }
        if($wpwa_social_login_obj){
```

```
        $login_response = $wpwa_social_login_obj->login();  
    }  
}  
}
```

First, we retrieve the `wpwa_social_login` and `wpwa_social_action` variables from the `$_GET` array. We will check the existence of the `wpwa_social_login` variable before proceeding further. Then, we switch the `wpwa_social_login` variable to identify the social network. Here, we have only included LinkedIn as we are only implementing the LinkedIn version in this chapter. You can add the remaining networks once you complete the implementation.

So, we initialize the `WPWA_LinkedIn_Connect` class using the `$wpwa_social_login_obj` object. Finally, we check the availability of valid object in the `$wpwa_social_login_obj` variable. Then, we call the login function to start the authentication process for social login. 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 `$app_config` file will be used as the redirection path. The next step in this process is to handle the response and authenticate the user into our application.

## Authenticating users to 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. However, 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:

- User clicks on the LinkedIn link
- User is redirected to LinkedIn for granting permissions to our application
- User is redirected back to our application on successful authentication with profile details
- Application checks whether 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 profile for completing remaining details

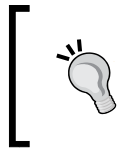
Now, let's see how we can handle the response object generated from the WPWA\_LinkedIn\_Connect class, to authenticate users into our application. First, we have to update the wpwa\_social\_login\_initialize function as follows to include the call to the register\_user function:

```
public function wpwa_social_login_initialize(){
    $wpwa_social_login_obj = false;
    $wpwa_social_login = isset($_GET['wpwa_social_login']) ?
    $_GET['wpwa_social_login'] : '';
    $wpwa_social_action = isset($_GET['wpwa_social_action']) ?
    $_GET['wpwa_social_action'] : '';
    if('' != $wpwa_social_login ){
        switch ($wpwa_social_login) {
            case 'Linkedin':
                $wpwa_social_login_obj = new WPWA_LinkedIn_Connect();
                break;
            default:
                break;
        }
    }
    if($wpwa_social_login_obj){
        $login_response = $wpwa_social_login_obj->login();
        $wpwa_social_login_obj->register_user($login_response);
    }
}
```

Now, we can take a look at the implementation of the `register_user` function inside `WPWA_Social_Connect`. Let's start with the first part of this function using the following code:

```
public function register_user($result){
    if($result->status){
        if($result->wpwa_network_type != 'twitter'){
            $user = get_user_by('email',$result->email);
        }else{
            $user = get_user_by('login',$result->username);
        }
        // Remaining Code
    }else{
        // Handle Errors
    }
}
```

We start the process by checking the status of the `$result` object. This is the response object generated after authenticating the account with LinkedIn. If a successful response is received, we check the network type for login. We assigned the network type inside the `login` function of the `WPWA_LinkedIn_Connect` class. We have to do the error handling part for unsuccessful responses. There are some key points we need to know about the username before moving forward.



Both the LinkedIn and Facebook API provide the e-mail address of the user, and hence, we use it to verify the users. However, the Twitter API doesn't provide an e-mail address, and hence, we have to use the Twitter username for verifying the user account.

So, we get the user by `login (username)` for Twitter and user by `email` for other social network. If the user is already registered with the given `login` or `email`, we will get a valid `$user` object. Otherwise, it will return `false`. Now, we can move into the next section of the `register_user` function:

```
public function register_user($result){
    if($result->status){
        // Retrieving the user from the database
        if(!$user){
            // Create a new user for the application
        }else{
            // Automatically authenticating existing users
        }
    }
}
```

---

```

        wp_redirect(admin_url('profile.php'));
    }else{
        // Handle Errors
    }

```

A valid object for `$user` means that the existing user is trying to log in through the LinkedIn connect. So, we authenticate and automatically log on the user and redirect to the profile page of WordPress backend. If valid user is not found, the user is trying to login through LinkedIn connect for the first time, and hence, we have to create a new account for the user. Let's start by looking at the user creation process inside the `if` statement:

```

if($result->wpwa_network_type != 'twitter'){
    $username = strtolower($result->first_name.$result->last_name);
    if(username_exists($username)){
        $username = $username.rand(10,99);
    }
}else{
    $username = $result->username;
}
$sanitized_user_login = sanitize_user($username);
$user_pass = wp_generate_password(12, false);
/* Create the new user */
$user_id = wp_create_user($sanitized_user_login, $user_pass,
    $result->email);
if (!is_wp_error($user_id)) {
    update_user_meta($user_id, 'user_email', $result->email);
    update_user_meta($user_id, 'wpwa_network_type', $result->
        wpwa_network_type);
    wp_update_user( array ('ID' => $user_id, 'display_name' =>
        $result->first_name.' '.$result->last_name) );
    wp_set_auth_cookie($user_id, false, is_ssl());
}

```

We already mentioned that the Twitter API provides the username for the user, and hence, we can use the same username for our application. The Facebook and LinkedIn APIs don't provide a username. So, we have to build a dynamic custom username using a combination of first and last names. However, the first and last name combination may not be a unique username. In such cases, we add a dynamic random number to the end of the username to make it unique. Then, we use the `sanitize_user` function for stripping out unsafe characters and generate the password using the `wp_generate_password` function.



Finally, we create a new user by passing username, password, and e-mail to the `wp_create_user` function. If the user creation is successful, we update the user with first and last names in the `wp_users` table. We also add the user e-mail and network type to the `wp_usermeta` table. Next, we will use the `wp_set_auth_cookie` function to create the authentication cookies for WordPress and log on the user automatically. Once registered, the user will be authenticated by setting the WordPress authenticate cookie and 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 profile with a new password to complete the registration.

Now, we can have a look at the `else` part of the code for existing users. It's pretty simple as we only have to automatically log on the existing users. So, we use the following line to create authenticate cookies and log on the user:

```
wp_set_auth_cookie($user->ID, false, is_ssl());
```

We have completed the user registration and login process using social networks. The intention of this implementation was to learn how to integrate third-party open source libraries into WordPress web applications. So, we choose the OAuth library and LinkedIn API class for integrating third-party services for OAuth login. Now, we have completed the library integration and OAuth login. Make sure that you test the user registration and login using different services. Having completed this process, you should be capable of integrating other libraries for networks such as Facebook and Twitter 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 web application by installing a bunch of third-party plugins. Ideally, developers should be focusing on limiting the number of plugins within an application to improve the structure of code and the possible conflicts.

On the other hand, some third-party libraries can 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% accurate, and we can't guarantee the results. The following are some of the plugins for checking malicious code and vulnerabilities of your plugins and themes:

- **Theme Authenticity Checker (TAC):** You can find this plugin at <https://wordpress.org/plugins/tac/>
- **Exploit Scanner:** You can find this plugin at <https://wordpress.org/plugins/exploit-scanner/>

- **Theme Check:** You can find this plugin at <https://wordpress.org/plugins/theme-check/>

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 projects list using Backbone.js
- Integrate the OAuth login for Facebook and Twitter
- Implement the subscriber notification process with cronjob 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 Oauth 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 will look at WordPress XML-RPC functions to build a simple yet flexible API for the portfolio application. Until then, make sure that you 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 rapid development processes 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 core logic of application. 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 the data of our applications.

WordPress provides the ability to create an API through its built-in API powered by XML-RPC. Also, WordPress is moving towards the JSON REST API, and hopefully, it will be available in the near future. The existing XML-RPC API caters to the blogging and CMS functionalities, while allowing developers to extend the APIs with custom functionalities. 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
- The WordPress XML-RPC API for web applications
- Building the API client
- Creating the custom API
- Integrating the API user authentication
- Integrating the API access tokens

- Providing the API documentation
- Time for action

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 allows 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 the 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.

## The 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 the 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 normally don't know your application
- Generally, APIs automate tasks that require user involvement, allowing much better and quicker experience for the users

With the increasing use of mobile-based 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 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 of building an API to extend the functionalities of web applications. So, we will look at the WordPress API in the next section.

## The WordPress XML-RPC API for web applications

With the latest versions, the WordPress API has matured into a secure and flexible solution that easily extends to cater to complex features. This was considered to be an insecure feature that exposed 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 of the WordPress API:

- Posts
- Taxonomies
- Media
- Comments
- Options
- Users

The complete list of components and respective API functions can be found in the WordPress codex at [http://codex.wordpress.org/XML-RPC\\_WordPress\\_API](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 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:

- **The API server:** This is the application where the API function resides
- **The API client:** This is a third-party application or service that requests the functionality of an API

Since we will use the existing API functions, we don't need to worry about the server, as it's built inside the core. So, we will build a third-party client to access the service. Later, we will improve the API server to implement custom functionalities that go beyond the existing API functions. The API client is responsible for providing the following features:

- Authenticating the user with the API
- Making XML-RPC requests to the server through the curl command
- Defining and populating the API functions with the necessary parameters

With 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 : ", "wpwa") .
$results);
    }
    return xmlrpc_decode($results);
}
}

```

Let's have a look at the following steps to build the API client:

1. First, we will 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.
2. 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. The 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.



curl is a command-line tool and library for transferring data with URL syntax, supporting various protocols, including HTTP and FTP. This is the most popular tool used in API requests and responses. On some servers, curl might not be enabled, so make sure that you enable curl on your server.

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.

3. Then, we can pass the converted XML file to a curl request to invoke the API functions on the server.
4. 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 the string, integer, or array format.



The 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 a 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.

5. Next, we need to define functions inside the client for invoking various API functions. Here, we will implement the 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);
}
```

Now, 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 in the earlier 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 preceding order as WordPress looks for 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 is mentioned that the response of `wp.getPosts` will only contain posts that the user has permission to edit. Therefore, the user will only receive the permitted post list. If you want to allow public access to all the post details, the custom API function needs be developed to query the database.

Now, let's see 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 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 the 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 have looked at the usage of existing API functions within WordPress. Next, we will 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 go beyond the generic blogging functionality. We need the implementation for both the server and client to create a custom API. Here, we will build an API function that outputs the list of developers in the portfolio application. Here, we will use a separate plugin for API creation as an API is usually a separate component from the application. Let's get started by creating another plugin folder called `wpwa-xml-rpc-api` with the main file called `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 called `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 the existing API methods as well as the methods added by plugins.



The `xmlrpc_methods` filter allows for the customization of the methods exposed by the XML-RPC server. This can be used to both add new methods and remove built-in methods.

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 called `getDevelopers` in the `wpwa` namespace to call a function called `developers_list`. The following code contains the implementation of the `developers_list` function for generating the entire 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 is 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 the permission.

Now, we know the basics of creating a custom API with WordPress. In the next section, we will 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. However, once you have an API, 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. Also, providing the API through SSL is almost a must to secure your API.

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 the authentication process for custom API methods. Let's create another API method for subscribing to the developers of the portfolio application. This feature is already implemented in the admin dashboard using admin list tables. Now, we will provide the same functionality for the 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" => __("Subscription
Completed.", "wpwa"));
    }
    } else {
```

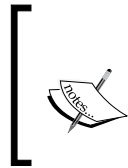
```

    return array("error" => __("Invalid Developer ID.", "wpwa"));
}
return $args;
}

```

Let's have a look at the steps.

1. First, we will retrieve the username and password from the arguments array and call the built-in `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 for retrieving the arguments, and hence, 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.

2. 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 the method parameters.
3. Next, we get the details of the preferred developer by using the `WP_User_Query` class by passing the role and developer ID.
4. Finally, we insert the record into the `wp_subscribed_developers` table to create a new subscription for a developer.

The other parts of the code contain the necessary error handlings based on various conditions. Make sure that you 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 called `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 accessing it unnecessarily. Therefore, we need to implement user tokens for limiting 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:

- To avoid the unnecessary overloading of server resources
- To bill the users based on API usage

If you are developing a premium API, it's important to track its usage for billing purposes. Various APIs use unique parameters to measure the API usage. Here are some of the unique ways of measuring API usage:

- The Twitter API uses the number of requests per hour to measure API usage
- Google Translate uses the number of words translated to measure API usage
- Google Cloud SQL uses input and output storage to measure API usage

Here, we won't measure the usage or limit the access to the portfolio API. Instead, we will be creating user tokens for measuring the usage in future. We will allow API access to the follower 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 will not discuss the preceding code in detail as we have already done it in the previous chapters. Only followers are allowed to access the API through tokens and thus, a capability named `follow_developer_activities` is used to integrate the screen for 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() {
    global $wpwa_template_loader, $api_data;
    $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 );
        }
    }
    $api_data['api_token'] = $api_token;
    ob_start();
    $wpwa_template_loader->get_template_part('api-settings');
    $html = ob_get_clean();
    echo $html;
}
```

In the preceding code, the same `user_api_settings()` function is used to generate the screen as well as to handle the form submission. We have a HTML form for the API settings in a template called `api-settings-template.php`. We can use our reusable template loader plugin to load the template through the global `$wpwa_template_loader` object. Let's look at the template file using the following code:

```
<?php global $api_data;
extract($api_data);
```



```
?>
<div class="wrap"><form action="" method="post" name="options">
  <h2><?php _e('API Credentials', 'wpwa'); ?></h2>
  <table class="form-table" width="100%" cellpadding="10">
    <tbody>
      <tr>
        <td scope="row" align="left">
          <label><?php _e('API Token :', 'wpwa'); ?><?php echo
$api_token; ?></label>
        </td>
      </tr>
    </tbody>
  </table>
  <input type="submit" name="api_settings" value="Update"
/></form>
</div>
```

Here, we have a HTML form with a submit button and a label for displaying the API key. We don't have any input fields as the key is generated automatically. Once the form is submitted, a new token needs to be generated as a hashed string.

Now, both the template and loading part are ready. However, this template resides outside our main plugin, and hence, the template loader object will not be able to identify the `templates` folder of this plugin. So, we need to use the extendable features to add the necessary template locations. First, we have to update the constructor to include the following code:

```
add_filter('wpwa_template_loader_locations', array( $this,
'api_template_locations' ));
```

Next, we can have a look at the implementation for the `api_template_locations` function to include custom template locations, as shown in the following code:

```
public function api_template_locations($locations){
    $location = trailingslashit( plugin_dir_path(__FILE__) ) .
'templates/';
    array_push($locations,$location);
    return $locations;
}
```

So, we add the path of the `templates` folder of our new plugin to the existing template locations array. Now, the template loader should be able to identify the `templates` folder.

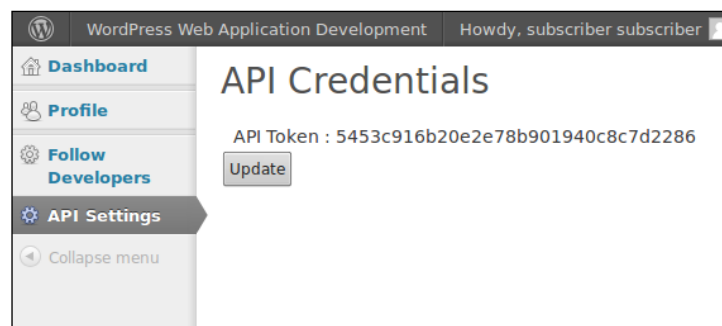
Now, we can move back to the API key generation process:

1. First, we check for the submission of the form.
2. Then, we generate a new token using a custom function called `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
    WXYZ';
    $random_string = '';
    for ($i = 0; $i < $length; $i++) {
        $random_string .= $characters[rand(0,
        strlen($characters) - 1)];
    }
    $random_hash = wp_hash($random_string);
    return $random_hash;
}
```

We can generate a dynamic hashed string by passing a random string to the existing `wp_hash` function provided by WordPress.

3. Then, we update the generated token using the `update_user_meta` function. If a user is loading the screen without submission, we will display the existing token to the user.
4. Now, you will have a menu item called **API Settings** in the admin menu bar to generate the API token screen, as shown in the following screenshot:



5. Having created the token, we now have to check for the token values before providing 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" => __("Subscription
Completed.", "wpwa"));
        }
    } else {
        return array("error" => __("Invalid Developer ID.",
"wpwa"));
    }
}
```

```

        return array("error" => __("Invalid Token.",
"wpwa"));
    }
    return $args;
}

```

- Now, the user will have to log in to the WordPress admin page, and generate a token before using the API. In a premium API, you can either 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:

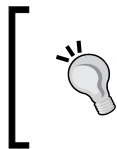
```

$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 the other API methods and parameters. This allows third-party users to request an API method and get the details about all the other functions.



WordPress uses the API method called `system.listMethods` for listing all the existing methods inside the API. Here, we will take one step further by providing the parameters of API methods 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", "wpwa"), __("API
Token", "wpwa")),
    "result" => __("Subscribing to Developer Activities", "wpwa")
    );
    $api_doc["wpwa.getDevelopers"] = array("authentication" =>
"optional",
    "api_token" => "optional",
    "parameters" => array(),
    "result" => __("Retrieve List of Developers", "wpwa")
    );
    return $api_doc;
}
```

Here, we have added all 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 the user needs a token to proceed. A list of allowed parameters to the API method is defined by `parameters`, and finally, the `result` parameter defines what a user will get 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 practical experience of the preceding techniques. So, I recommend that you try the following tasks to extend the knowledge gathered in this chapter:

- You can measure the API usage through the number of requests.
- We created an API function to list all the projects and services of the application. Try to introduce the 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 the 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 measuring 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 next chapter, we will be restructuring our application plugin to improve consistency of code, while looking at some of the incomplete areas of the 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 that should be planned and managed with well-defined processes. Typically, we separate large applications into smaller submodules, where each submodule is tested independently from other modules. Finally, we integrated all the modules to complete the application. The integration of modules is one of the most difficult tasks in application development.

The portfolio management application created throughout this book was intended to illustrate the advanced concepts of WordPress web application development. Therefore, we had to use different techniques in different modules to understand the issues and find feasible solutions. In real world, we have to limit the use of different techniques and keep the consistency across all the features of the application. So, we will be fixing the inconsistencies of the application while restructuring the necessary components. After the completion of this chapter, developers should be able to build similar or complex applications without any difficulty.

In this chapter, we will cover the following topics:

- Integrating and structuring the portfolio application
- Integrating the template loader into a user manager
- Working with a restructured application
- Updating the user profile with additional fields



- Scheduling subscriber notifications
- Time for action
- Final thoughts

Let's get started!

## Integrating and structuring the portfolio application

Throughout the first nine chapters, we implemented the functionality of a developer portfolio management system using one main plugin and several independent plugins. In each chapter, we explained some of the advanced concepts while developing the features related to that concept. So, our application was structured based on modules in WordPress. In the real world, we plan all the features of the application at the same time while separating them into sections based on functionality. Here, we have separated them into sections based on WordPress core modules. So, let's restructure and integrate the inconsistent components in our application before moving forward.

We can go through the code of the main plugin and sub plugins developed through the application to identify the following inconsistencies and issues:

- We used the template loader as an external reusable plugin. So, we have to check the availability of the template loader before using it inside classes. The availability checking is done inside the `wpwa_plugin_init` function. We have only implemented it for the `WPWA_Custom_Post_Types_Manager` class.
- User management functionalities are implemented inside the main plugin class. Ideally, we should separate user functionality into a new class while keeping the common functionality in the main plugin class.
- Add the template loader into the user management functionality.
- Separate modules based on the application logic instead of WordPress modules.

So, now we need to restructure the application to resolve the preceding issues and build a solid foundation for developing any type of web application with WordPress. Let's get started!

## Adding the template loader dependencies

We have four classes in the main plugin that depend on the template loader and we have only checked the dependency for the `WPWA_Custom_Post_Types_Manager` class. Let's define the dependencies for the remaining classes. The following code contains the current code for the `wpwa_plugin_init` function inside the `class-wpwa-portfolio-manager.php` file:

```
function wpwa_plugin_init(){
    if(!class_exists('WPWA_Template_Loader')){
        add_action( 'admin_notices', 'wpwa_plugin_admin_notice' );
    }else{
        global $wpwa_custom_post_types_manager;
        $wpwa_custom_post_types_manager = new
        WPWA_Custom_Post_Types_Manager();
    }
}
```

Now, let's add the remaining dependencies inside this function, as shown in the following code:

```
function wpwa_plugin_init(){
    if(!class_exists('WPWA_Template_Loader')){
        add_action( 'admin_notices', 'wpwa_plugin_admin_notice' );
    }else{
        global $wpwa_custom_post_types_manager, $wpwa_theme,
        $wpwa_settings, $backbone_projects ;
        $wpwa_custom_post_types_manager = new
        WPWA_Custom_Post_Types_Manager();
        $backbone_projects = new WPWA_Backbone_Projects();
        $wpwa_settings = new WPWA_Settings();
        $wpwa_theme = new WPWA_Theme();
    }
}
```

We have added the remaining dependencies by initializing the objects after checking the existence of the `WPWA_Template_Loader` class. Now, we have to remove the object initializations within the respective classes. You have to follow this process for any new class with a dependency to the template loader object.

## Integrating the template loader into a user manager

In *Chapter 2, Implementing Membership Roles, Permissions, and Features*, we used direct file inclusions to load the necessary templates. A few chapters later, we improved the loading of templates by introducing a common template loader. Now, we can integrate the template loader into the user management functionality to keep the code consistent. Also, we need to move the user-related functionality into its own class instead of keeping it inside the main plugin file. First, we'll move the user-related functionality to its own class.

Create a new model called `WPWA_Model_User` inside the `models` folder of our main plugin. Then, we can move all the user-related functions from the `WPWA_Portfolio_Manager` class to the `WPWA_Model_User` class. The following code shows the structure and function definitions of the `WPWA_Model_User` class after the changes. The implementation of the functions will be the same and hence avoided in the code here in the `WPWA_Model_User` class:

```
<?php
class WPWA_Model_User {
    public function __construct(){
        add_action('wpwa_register_user',array($this,'register_user')
    );
        add_action( 'wpwa_login_user', array( $this, 'login_user' ) );
        add_action('wpwa_activate_user',array($this,'activate_user'
    ));
        add_filter( 'authenticate', array( $this, 'authenticate_user'
    ),30, 3 );
    }
    public function add_application_user_roles() {}
    public function remove_application_user_roles() { }
    public function add_application_user_capabilities() {}
    public function activate_user() { }
    public function login_user() {}
    public function authenticate_user( $user,$username,$password) {}
    public function register_user() {}
}
?>
```

This class included all the main functions related to a user. The last four functions are called within the constructor. However, we can't find the calls to the first three functions. These functions were earlier implemented in `WPWA_Portfolio_Manager`, and hence, the function calls were made within that class. Now, we have to make external function calls from the `WPWA_Portfolio_Manager` class to the `WPWA_Model_User` class. Let's update the `activate_portfolio_manager` function as follows:

```
public function activate_portfolio_manager(){
    global $wpwa_user;
    $wpwa_user->add_application_user_roles();
    $wpwa_user->remove_application_user_roles();
    $wpwa_user->add_application_user_capabilities();
    $this->flush_application_rewrite_rules();
    $this->create_custom_tables();
}
```

Now, we are using the global `$wpwa_user` model object to make the function calls. However, we didn't initialize the `$wpwa_user` object. So, let's update the `wpwa_plugin_init` function as explained earlier to include the initialization of the `$wpwa_user` object. Now, we are ready to modify the template loading of the user management functionality.

Let's apply the template loader instead of using direct template inclusions. Consider the template loading section of the `register_user` function, as shown in the following code:

```
include dirname(__FILE__) . '/templates/register-template.php';
exit;
```

Now, we can take a look at the modified version with the support of the template loader object, as shown in the following code:

```
$wpwa_template_data['errors'] = isset($errors) ? $errors: array();
$wpwa_template_data['user_login'] = isset($user_login) ?
$user_login : '';
$wpwa_template_data['user_email'] = isset($user_email) ?
$user_email : '';
$wpwa_template_data['user_type'] = isset($user_type) ? $user_type
: '';
ob_start();
$wpwa_template_loader->get_template_part('register');
echo ob_get_clean();
exit;
```

With the implementation of the new process, we can pass the template data through a global variable called `$wpwa_template_data`. This is the process recommended by WordPress instead of including templates directly.



We have completed the refactoring process and fixed the potential impacts to the existing components. There can be several other impacts, which aren't discussed here. Feel free to find them and discuss them on the book website. Once refactoring is completed, you should also carry out regression testing for the existing test cases against the modified code.

WordPress has emerged as a web development framework in recent years. However, it still has a very limited amount of 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. Yet, there is still 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 website and help to 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 will build the developer list page with 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 a developers list. All the existing developers will be displayed in the initial page load. Then, users can use the autocomplete textbox to search 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 created all the rewriting rules inside the `manage_user_routes` function of the `WPWA_Portfolio_Manager` class. So, let's look at the updated code with the inclusion of the 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');
}

```

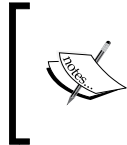
Once the rule is defined, we can match all the list-based pages with the `control_action` parameter. In this scenario, our URL will be `/list/developers`, and hence, we need to match the `control_action` parameter to the value of `developers`. Now, we have to update the `front_controller` function with the new control action, as shown in the following code:

```

case 'developers':
    $result = $wpwa_developer->list_developers();
    $wpwa_template_data['developers'] = $result;
    ob_start();
    $wpwa_template_loader->get_template_part("developer_list");
    echo ob_get_clean();
    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.



We have used a template loader object to load the template, and hence, you need to define `$wpwa_template_data` and `$wpwa_template_loader` as global objects just after the function declaration.

## Building the developer model

Models are mainly used for working with the database. We created four models in the custom post manager for projects, books, articles, and services. The developer also plays 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 called `class-wpwa-model-developer.php` inside the `models` folder.

An autoloader created in the earlier section uses a prefix called `WPWA_Model_` to load the models, and hence, we will name the new model class as `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;
    }
}
```

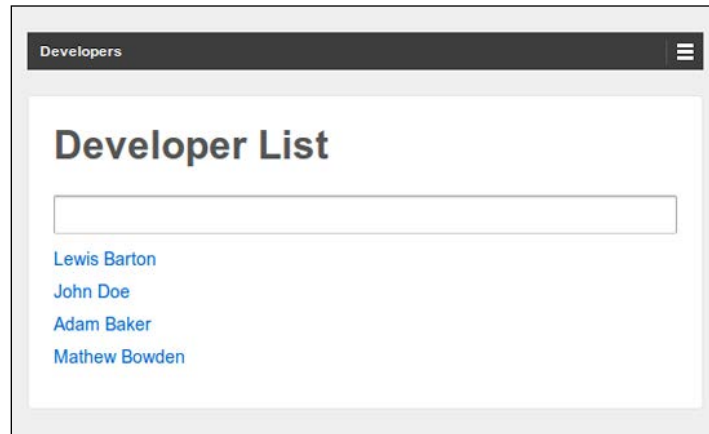
Here, we use the `WP_User_Query` class to retrieve 25 developers to be displayed on the initial page load. Next, we have to create a template to display the data retrieved from the `list_developers` function.

## Designing the developer list template

With the new structure, all 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 developers' list. So, let's create a new template file called `developer_list-template.php`, as shown in the following code:

```
<?php
    global $wpwa_template_data;
    extract($wpwa_template_data);
    get_header(); ?>
<div class='main_panel'>
    <div class='developer_profile_panel
developer_profile_panel_list'>
        <h2><?php echo __("Developer List", "wpwa"); ?> </h2>
        <div class='field_label'><input type="text"
id="autocomplete_dev_list" name="autocomplete_dev_list" /></div>
        </div>
        <div id='developer_list'>
            <?php foreach($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's list page. The following screenshot previews the developer page with an initial dataset:



## Enabling AJAX-based filtering

AJAX-based filtering is becoming a trend in web development, used by many popular sites such as Facebook and Google. Here, we will implement the filtering for the developer's list. Once a user presses 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 `wpwa-developer.js` file with the following code to enable AJAX-based filtering:

```
$jq(document).ready(function() {
    $jq("#autocomplete_dev_list").keyup(function() {
        $jq.post( wpwacnf.ajaxURL, {
            action:"wpwa_developer_list",
            search : $jq(this).val(),
        }, function( response ) {
            console.log(response);
            ajax_developer_list(response);
        }, "json");
    });
});
```



We invoke a jQuery AJAX request on the keyup event of the #autocomplete\_dev\_list textbox. Here, we have used JSON as the data type and ajax\_developer\_list as the success handler function. We pass the search string and the corresponding server side action to the wpwa\_developer\_list action. Let's add the AJAX action handler to the constructor of the WPWA\_Model\_Developer class, as shown in the following code:

```
public function __construct(){
    add_action('wp_ajax_nopriv_wpwa_developer_list', array($this,
    'developer_list'));
    add_action('wp_ajax_wpwa_developer_list', array($this,
    'developer_list'));
}
```

Let's go through the implementation of the 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;
}
```

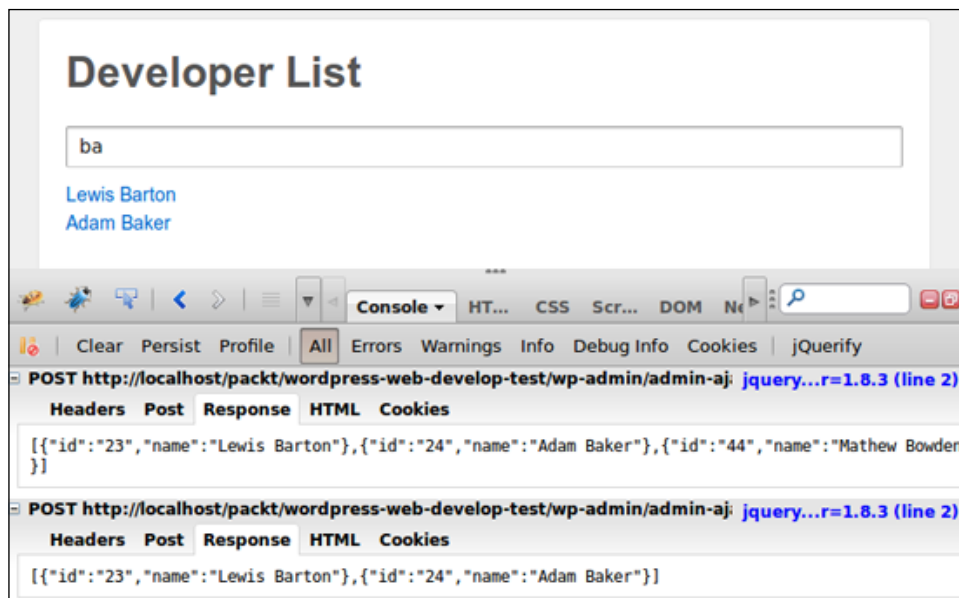
First, we retrieve the search string from the \$\_POST data for the request. Then, we use the custom query by joining the users table and the usermeta table to search by display name. We can check for the necessary user roles using a LIKE statement on the wp\_capabilities key on the usermeta table. Here, we had to use a custom query as WP\_User\_Query does not offer any 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:

```
var ajax_developer_list = function(result){
    $jq("#developer_list").html("");
    $jq.each(result, function(i, val) {
        if(val){
            $jq("#developer_list").append('<div class="developer_row">\n\
            <a href="'+wpwacnf.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 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's list. Users can enter the search text inside the textbox to filter the values. Consider the following screenshot for the filtered developer list on pressing two keys:



As illustrated in the preceding screenshot, the 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. An alternative option is to start the searching once a user inputs at least  $n$  number of characters. So, it's important to use this technique wisely with the optimization of database with indexing.

Up until this point, we 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 uncompleted. So, here we will complete the foundation of the portfolio application by completing those remaining tasks. In the next two sections, we will 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

The 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 the users. Here, we will capture more information by using additional fields on the profile screen of the WordPress dashboard. So, let's update the constructor function of the `WPWA_Model_User` class to add the necessary actions for editing the profile, as shown in the following code:

```
add_action('show_user_profile', array($this,
    "add_profile_fields"));
add_action('edit_user_profile', array($this,
    "add_profile_fields"));
```

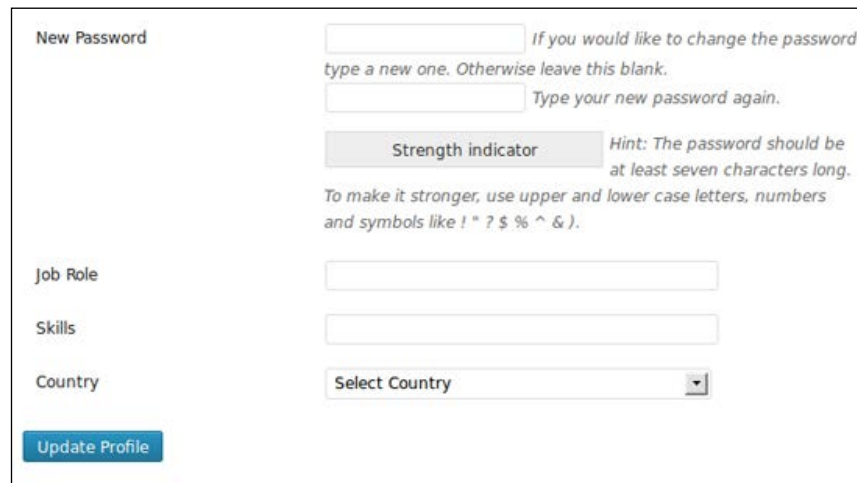
We have defined two actions to be executed on the user profile screen. Both the `show_user_profile` and `edit_user_profile` actions are used to add new fields to the end of the user edit form. According to the preceding code, the addition of new fields will be implemented in the `add_profile_fields` function of the `WPWA_Model_User` class. Let's look at the implementation of the `add_profile_fields` function:

```
public function add_profile_fields() {
    global $user_ID, $wpwa_template_loader, $wpwa_template_data;
    ob_start();
    $wpwa_template_loader->get_template_part("profile_fields");
    echo ob_get_clean();
}
```

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:

```
<?php
    global $wpwa_template_data;
    extract($wpwa_template_data);
?>
<table class="form-table">
    <tr>
        <th><label for="job_role"><?php _e("Job Role", "wpwa");
?></label></th>
        <td><input type="text" class="regular-text" value="<?php echo
$job_role; ?>" id="job_role" name="job_role"></td>
    </tr>
    <tr>
        <th><label for="skills"><?php _e("Skills", "wpwa");
?></label></th>
        <td><input type="text" class="regular-text" value="<?php echo
$skills; ?>" id="skills" name="skills"></td>
    </tr>
<?php
    $countries = array(
        'AF' => 'Afghanistan',
        'AL' => 'Albania',
    );
?>
    <tr>
        <th><label for="country"><?php _e("Country", "wpwa");
?></label></th>
        <td>
            <select name="country" id="country">
                <option value="" ><?php _e("Select Country", "wpwa");
?></option>
                <?php foreach($countries as $country_name){ ?>
                    <option <?php echo($country_name == $country)? "selected":
"";?> value="<?php echo $country_name;?>"><?php echo $country_
name;?></option>
                <?php } ?>
            </select>
        </td>
    </tr>
</table>
```

Basically, we have three fields for storing the job role, skills, and the country name of the developers. At this stage, the `$wpwa_template_data` array is empty, and hence, these fields won't have default values. It's important to use the CSS class `form-table` for keeping the consistency of design with the existing fields. Now, your profile page should look something similar to the following screenshot:



## Updating the values of the profile fields

Once a user clicks the **Update Profile** button, all the custom fields need to be saved automatically into the database. So, we have to define another two actions called `edit_user_profile_update` and `personal_options_update`. We have to add them inside the constructor of the `WPWA_Model_User` class as shown in the following code:

```
add_action('edit_user_profile_update', array($this,
    "save_profile_fields"));
add_action('personal_options_update', array($this,
    "save_profile_fields"));
```

Actions hooks defined in the preceding code are generally used to update additional profile fields. So, we will invoke the `save_profile_fields` function of a user model to cater to the persisting tasks. Consider the implementation of the `save_profile_fields` function inside the `WPWA_Model_User` class, 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);
}

```

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 screen. 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 screen:

```

public function add_profile_fields() {
    global $user_ID, $wpwa_template_loader, $wpwa_template_data;
    $job_role = esc_html(get_user_meta($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));
    $wpwa_template_data['job_role'] = $job_role;
    $wpwa_template_data['skills'] = $skills;
    $wpwa_template_data['country'] = $country;
    ob_start();
    $wpwa_template_loader->get_template_part("profile_fields");
    echo ob_get_clean();
}

```

With the new implementation, we can access the template variables inside the template using the `$wpwa_template_data` array. Having completed the profile field creation, we can now move onto our main goal of displaying the profile details inside the developer profile screen in 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 inside the `WPWA_Backbone_Projects` class:

```

public function create_developer_profile($developer_id) {
    global $project_data, $wpwa_template_loader;
    $user_query = new WP_User_Query(array('include' =>
array($developer_id)));
    $project_data = array();
    foreach ($user_query->results as $developer) {

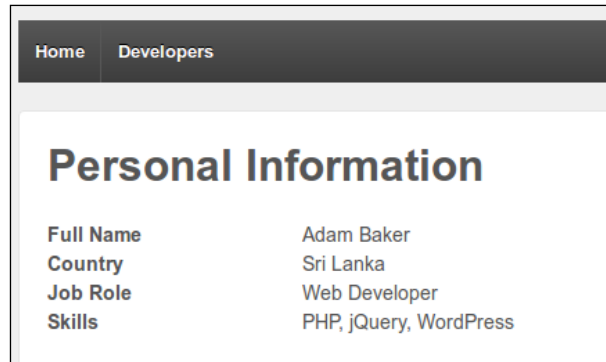
```

```
$project_data['display_name'] = $developer->data-
>display_name;
$project_data['job_role'] = esc_html(get_user_meta($developer-
>data->ID, "_wpwa_job_role", TRUE));
$project_data['skills'] = esc_html(get_user_meta($developer-
>data->ID, "_wpwa_skills", TRUE));
$project_data['country'] = esc_html(get_user_meta($developer-
>data->ID, "_wpwa_country", TRUE));
}
$current_user = wp_get_current_user();
$project_data['developer_status'] = ($current_user->ID ==
$developer_id);
$project_data['developer_id'] = $developer_id;
ob_start();
$wpwa_template_loader->get_template_part("developer");
echo ob_get_clean();
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-template.php` template to include the profile field data, as shown in the following code:

```
<div class='developer_profile_panel'>
  <h2><?php echo __('Personal Information','wpwa'); ?></h2>
  <div class='field_label'><?php echo __('Full Name','wpwa');
?></div>
  <div class='field_value'><?php echo
esc_html($project_data['display_name']); ?></div>
  <div class='field_label'><?php echo __("Country","wpwa");
?></div>
  <div class='field_value'><?php echo
esc_html($project_data['country']); ?></div>
  <div class='field_label'><?php echo __("Job Role","wpwa");
?></div>
  <div class='field_value'><?php echo
esc_html($project_data['job_role']); ?></div>
  <div class='field_label'><?php echo __("Skills","wpwa");
?></div>
  <div class='field_value'><?php echo
esc_html($project_data['skills']); ?></div>
</div>
```

Now, go to the browser and access `/user/profile/{user id}`, and you will get a screenshot similar to the following one:



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 be completing the implementations of 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 will 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 functionality with less flexibility. In WordPress, this action will be triggered only when someone visits the site after the scheduled time has passed. In a normal cron job, the action will be triggered without any interaction from users. 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 called `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 to provide the results of the cron. You should use a unique name as the hook. The last parameter defines the arguments to the hook, which we can keep blank in most cases.



The `wp_schedule_event` function initializes recurred function executions, and hence, should be avoided inside a hook such as `init`, which gets executed on every request. Ideally, scheduling events should be done inside the plugin activation handler.

First, we have to update the `activate_portfolio_manager` function to call the scheduling function on plugin activation. Consider the following code for an updated `activate_portfolio_manager` function:

```
public function activate_portfolio_manager(){
    global $wpwa_user;
    $wpwa_user->add_application_user_roles();
    $wpwa_user->remove_application_user_roles();
    $wpwa_user->add_application_user_capabilities();
    $this->flush_application_rewrite_rules();
    $this->create_custom_tables();
    $this->create_event_schedule();
}
```

Let's schedule subscriber notifications by implementing the `create_event_schedule` function inside the main plugin file as follows:

```
public function create_event_schedule() {
    wp_schedule_event(time(), 'everytenminutes',
    'notification_sender');
}
```

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 called `everytenminutes` to execute the task at 10-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 called `notification_sender` for executing a 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 kinds of functionalities in the utility class or in a file. Here, we will include it inside the `wpwa-actions-filters.php` file. Let's begin with the implementation of a custom interval:

```
function everytenminutes($schedules) {
    $schedules['everytenminutes'] = array(
        'interval' => 60*10,
        'display' => __('Once Ten Minutes', 'wpwa')
    );
    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 10- minute schedule using 600 seconds as the time interval. Now, the schedules list will have four values including the 10-minute interval. Next, we have to implement the `notification_sender` hook for sending notifications to subscribers.

## Notifying subscribers through e-mails

The process of notifying subscribers through e-mails 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 notification sending:

- 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 a way to track the posts that have been already notified and those that are yet to be notified. Therefore, we will use a custom post meta value to track the notified status. I hope you 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 will update the `send_subscriber_notifications` function to suit the new process, as shown in the following code:

```
public function send_subscriber_notifications($post) {
    update_post_meta($post->ID, "notify_status", "0");
}
```

Here, we have removed the e-mail sending functionality and updated a post meta value called `notify_status` to contain a value of `0`. The `notify_status` parameter of `0` means the post is new and subscribers haven't been notified.

Next, we will look at the implementation of the `notification_sender` hook inside the `wpwa-actions-filters.php` file.

```
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", "wpwa");
    $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", "wpwa");
    // Remaining code
}
```

We have defined the `notification_sender` action with a function called `notification_send`. The first part of the function contains the necessary code for initializing the `PHPMailer` class for sending e-mails. Afterwards, we have to grab the posts with the `notify_status` parameter of `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 the `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", "wpwa") . "<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 the browser. Afterwards, we send the e-mail with the new updates. This process will be continued for each and every post with a `notify_status` parameter of `0`. Once the e-mail is sent, we update the `notify_status` parameter to `1` to prevent duplicate notification in the next schedule.



WordPress scheduling works in a similar way to the cron jobs in Linux-based systems. However, we have a limitation compared to the 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 has 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 a 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 website, and I hope you will follow the rest of the development.

In the next section, we will talk about a few features of WordPress that we have left out so far, but are important to web application development.

## Time for action

Throughout this book, we have developed various practical scenarios to learn the art of web application development. Here, we have the final set of actions before we complete the portfolio application for 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 for getting experienced with the process:

- Find out different ways of structuring WordPress for web applications
- Improve the AJAX-based list to contain more filtering options

## 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. However, 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 a general web application development. The WordPress architecture is different from the 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.

In this chapter, we completed the development of the demo application for this book. So, feel free to discuss your own application structures and the techniques you have used for WordPress applications on the website of this book at <http://www.innovativephp.com/wordpress-web-applications>.

## Summary

We began this chapter by looking at the issues of the application plugins developed throughout the first nine chapters of the book. Our portfolio application lacked a proper structure in some parts. So, we restructured the application to keep the consistency across all its features.

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 in order to understand how to work with the restructured application.

Here, we are completing the demo application for this book with the basic foundation of the portfolio application. Make sure that you follow the guides on the website for this book to understand the more complex theories and techniques of developing web applications while completing the portfolio management system.

You can now take a look at the next chapter for supplementary modules for WordPress applications. This will be a theoretical chapter explaining the concepts to improve your application with utility features.



# 11

## Supplementary Modules for Web Development

In web application development, we mainly focus and plan our business logic. Throughout the first ten chapters of this book, we developed features that are directly related to the portfolio management application. However, there are supplementary features that are not related to the business requirements of the application, and yet play a vital part in the success of a project.

Multi-language support, caching, and security are important features of any web application. WordPress provides built-in features to support these type of non-application-related tasks. Apart from this, WordPress also offers some features that can be used to improve the flexibility and user experience of applications. Throughout this chapter, we will give a brief introduction to these supplementary modules so that you can use them when the opportunity arises.

In this chapter, we will cover the following topics:

- Internationalization
- Working with the media grid and image editor
- Introduction to the post editor
- Lesser-known WordPress features
- Introduction to multisite
- Time for action
- Final thoughts

Let's get started.



## Internationalization

Internationalization is the process of making your application ready for translating to other languages. WordPress itself provides translation for its core by default. In most cases, we will be developing web applications using plugins or themes. So, it's essential to make the theme and plugins translatable for an improved flexibility. We can develop web applications as standalone projects for a specific client or as projects for any client that you wish to use for. Internationalization is more important in the latter as a product is used by many clients compared to a project. Even with projects, it's an important aspect of development, when your client wants a non-English application.

In this section, we will look at internationalization support in WordPress and how to translate and manage plugins.

## Introduction to WordPress translation support

WordPress allows translation support for plugins using the GetText Portable Object. We need to have three things to enable translation support in WordPress:

- Defining the text domain
- Enabling translations on strings using WordPress functions
- Loading the text domain

WordPress uses the text domain to define all the text strings that belong to a certain plugin. In our scenario, we choose `wpwa` as the text domain for all our plugins. This should be a unique identifier for your plugin and can have alphanumeric characters and dashes. This text domain needs to be placed in all translation functions, just as we did throughout the last ten chapters.

Then, we need to enable translations by using the WordPress translate functions. We have to enable translations on every string that gets displayed on the site. The following code contains a sample translation function extracted from our main plugin:

```
$template_data['service_availability_label'] = __('Service  
availability', 'wpwa');  
$template_data['service_price_type_label'] = __('Service Prize  
Type', 'wpwa');
```

As you can see, we have enabled translation using the `__()` function and we have used `wpwa` as the text domain. Let's consider the translation functions in WordPress.

## The translation functions in WordPress

WordPress provides a lot of translation functions for different scenarios. Among those functions, `__` and `_e` are the most frequently used functions. Both these functions retrieve translated strings using the WordPress `translate` function. The difference between the two functions is that the first one returns the translated string, while the latter directly displays the translated string. Generally, we can use `__()` for variables and business logic while `_e()` is used for templates. Apart from these two functions, there are many other functions such as `_n()`, `_x()`, `esc_attr__()`, and so on. These functions will be used in advanced use cases. You can learn more about these functions inside the codex using [http://codex.wordpress.org/Function\\_Reference/translate](http://codex.wordpress.org/Function_Reference/translate).

## Creating plugin translations

Most popular plugins are translation-ready, which means you can use the built-in translation files or create translation files for any language. However, this feature seems to be lacking in many less popular plugins, as well as application plugins that were built for a specific website. So, as a developer, it's important to understand how to add support for translation files and create translation files from scratch. Let's see how we can build a fully translatable plugin. Create a new plugin in a file called `wpwa-lang.php`. Then, add a shortcode with a translatable string, as shown in the following code:

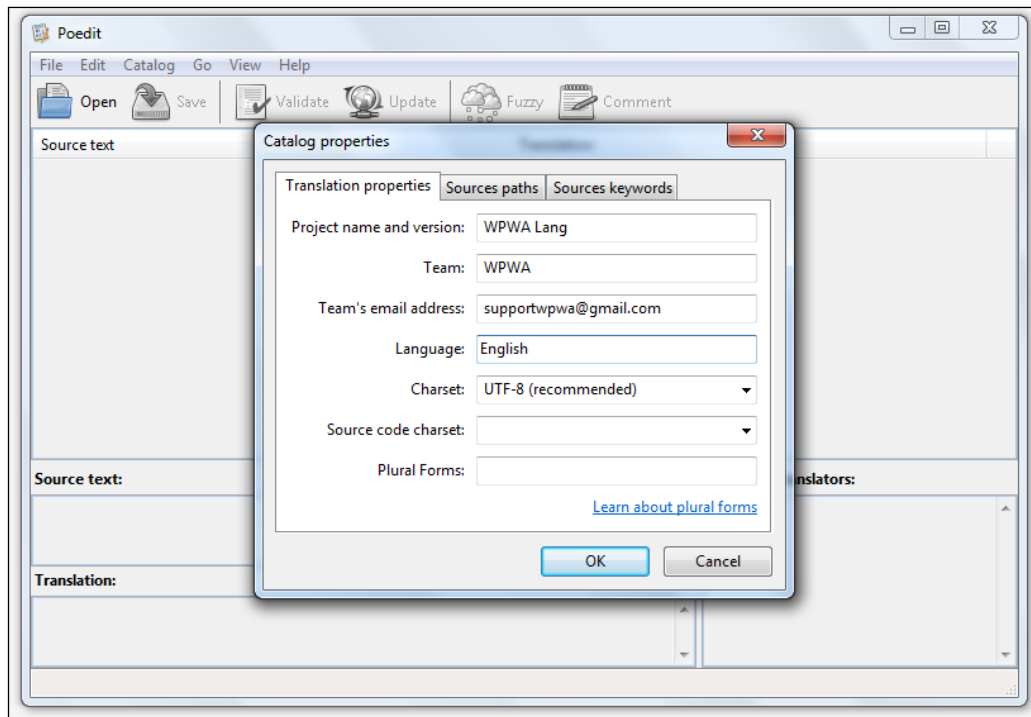
```
<?php
/*
 * Plugin Name: WPWA Language
 * Plugin URI:
 * Description: Learn translation management in WordPress.
 * Version: 1.0
 * Author: Rakhitha Nimesh
 * Author URI:
 * Text Domain: wpwalang
 */
add_shortcode('wpwa_lang_checker','wpwa_lang_checker');
function wpwa_lang_checker(){
    $app_name = "<h1>" . __('My Application','wpwalang') .
"</h1>";
    return $app_name;
}
```

Here, we have a basic shortcode that displays the text `My Application` inside a header tag. We have enabled translation using the `__()` function and we have used `wpwalang` as the plugin text domain. Now, we need to create translation files for this plugin.

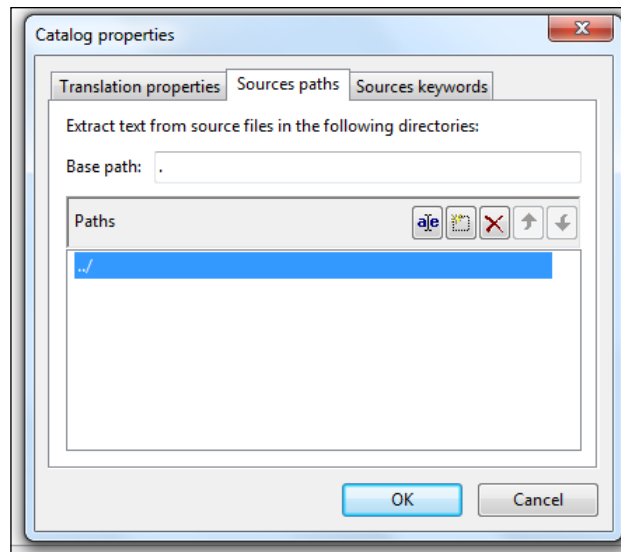
## Creating the POT file using PoEdit

First, we have to create the **Portable Object Template (POT)** file for the plugin. This file is considered as the base for translations. If you want to create a translation for a new language, you have to use this file and generate the translation file. Apart from the .pot file, we also have .mo and .po files. The **Portable Object (PO)** file is what we get as the result by translating the .pot file. The **Machine Object (MO)** file contains a compiled version of the .po file and is used to apply the translations. Having got ten a basic introduction to these file types, now we can move into creating those fields.

We can use the PoEdit software to generate and manage translations. You can download PoEdit from <http://sourceforge.net/projects/poedit/>. I have downloaded the version for Windows. You can download PoEdit for your OS. Once installed, we have to go to **File | New Catalog** to create a new catalog for our project, as shown in the following screenshot:

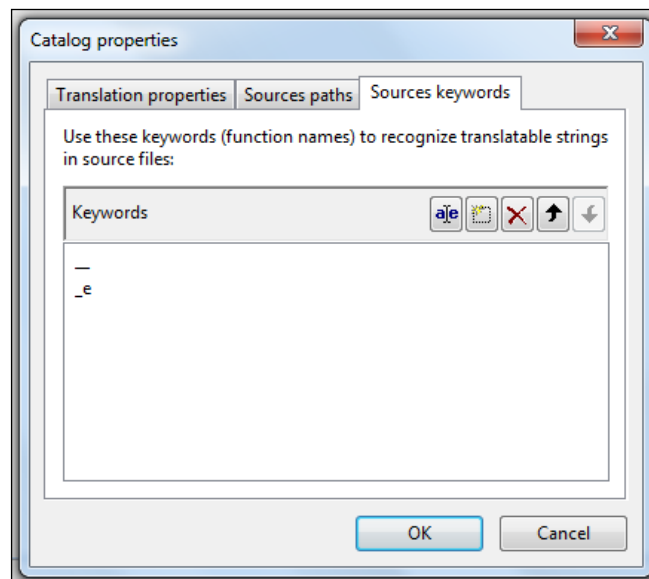


We can fill the fields for project name, team name, e-mail, and language, as shown in the preceding screenshot. Next, we have to add the source paths to our plugin. Consider the following screenshot for adding source paths:



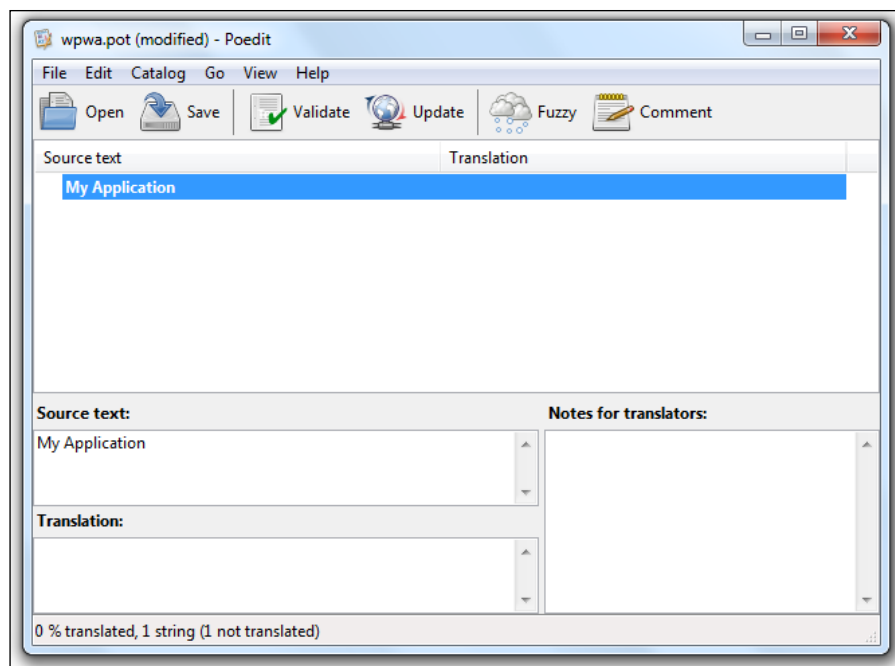
In this section, we have to define the paths to identify the source files. The source files within these paths will be explored by PoEdit to find translatable strings. We are planning to place our language files inside a folder called `lang`. So, all source files will be available one folder above the `lang` folder, and hence, we use `../` to define the source path. You can add as many paths as you wish using the **New Item** button.

Finally, we have to define the **Source keywords**, as shown in the following screenshot:



Here, we have added `__` and `_e` as the source function names for our project. If you are using other WordPress translation functions, you will have to add them using the **New Item** button as well. Based on current settings, PoEdit will only look for `__()` and `_e()` functions inside the project files.

Once we click on the **OK** button, PoEdit will search the source files and ask you to save the file. We have to name the file with the `.pot` extension before saving. Make sure that you save the `.pot` file inside a new folder called `lang` in your plugin folder. Once saved, PoEdit will show the strings available for translation, as shown in the following screenshot:



Since we only have one string, PoEdit will only display that string. If you apply this technique to our main portfolio plugin, you will get a huge list of translatable strings. Now, our next task is to create the actual translation files from the `.pot` file. Let's start with the translation file for the English language.

By default, we have the `.pot` file in English. So, we don't need to add any translations. You can just choose the **save as** option and save the file as `wpwalang-en_US.po` in the `lang` folder. In here, we use `wpwalang` as the text domain and then `en_US` as the language. Once the `.po` file is saved, it will automatically create the compiled `.mo` file.

Now, let's create another file for the French language. In this case, we have to add translations. So, get the translated text for `My Application` and add it to the **Translation** section, as shown in the bottom of the screenshot. We have to repeat this task for all other translations. Then, choose **save as** and save the file as `wpwalang-fr_FR.po` in the `lang` folder. Now, we have two translation files for the English and French languages.

## Loading language files

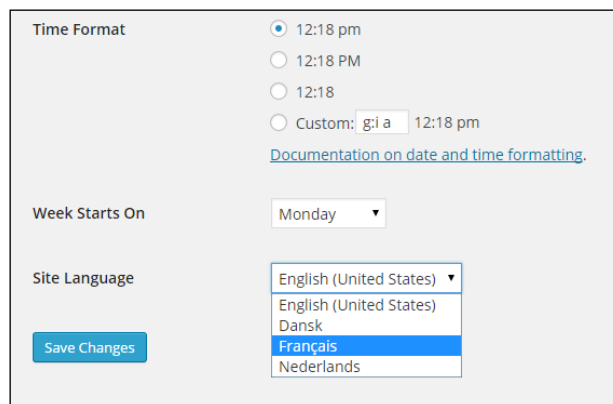
Now, we have to let WordPress know that there are translation files available for this plugin. WordPress provides a function called `load_plugin_textdomain` to define the translation file path. Let's add the following code to our plugin for the loading plugin text domain:

```
add_action('init', 'wpwa_lang_textdomain');
function wpwa_lang_textdomain() {
    load_plugin_textdomain('wpwalang', false, dirname(
        plugin_basename( __FILE__ ) ) . '/lang');
}
```

We have added the path to our language files folder with the plugin text domain. Now, WordPress will look for translation files with the prefix `wpwalang` followed by the language code. Now, everything is set up for the translation support for our new plugin.

## Changing the WordPress language

WordPress Version 4.0 and later allows us to change the site language from the admin section. Once the language is changed, the text in our plugin will be displayed in their respective language. Let's change the language to French using **Site Language** in the WordPress **Settings** | **General** section, as shown in the following screenshot:



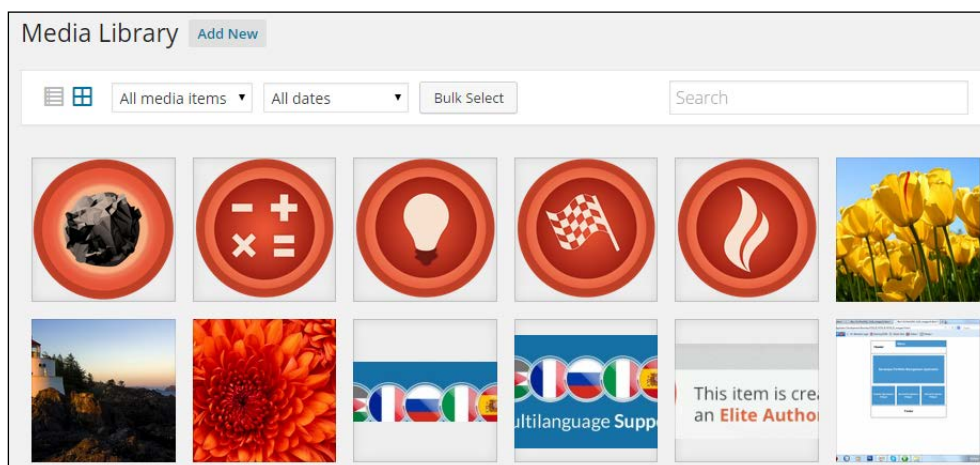
The screenshot shows the WordPress Settings | General section. The 'Time Format' section has four radio button options: '12:18 pm' (selected), '12:18 PM', '12:18', and 'Custom: g:i a' (with '12:18 pm' entered in the adjacent text field). Below these is a link to 'Documentation on date and time formatting.'. The 'Week Starts On' section has a dropdown menu set to 'Monday'. The 'Site Language' section has a dropdown menu with the following options: 'English (United States)' (selected), 'English (United States)', 'Dansk', 'Français' (highlighted in blue), and 'Nederlands'. A 'Save Changes' button is located at the bottom left of the settings area.

Once the language is changed, you will see the French version of the WordPress admin section. Now, create a new page/post and add our shortcode. You will see that `My Application` is converted to French as defined in our translation file.

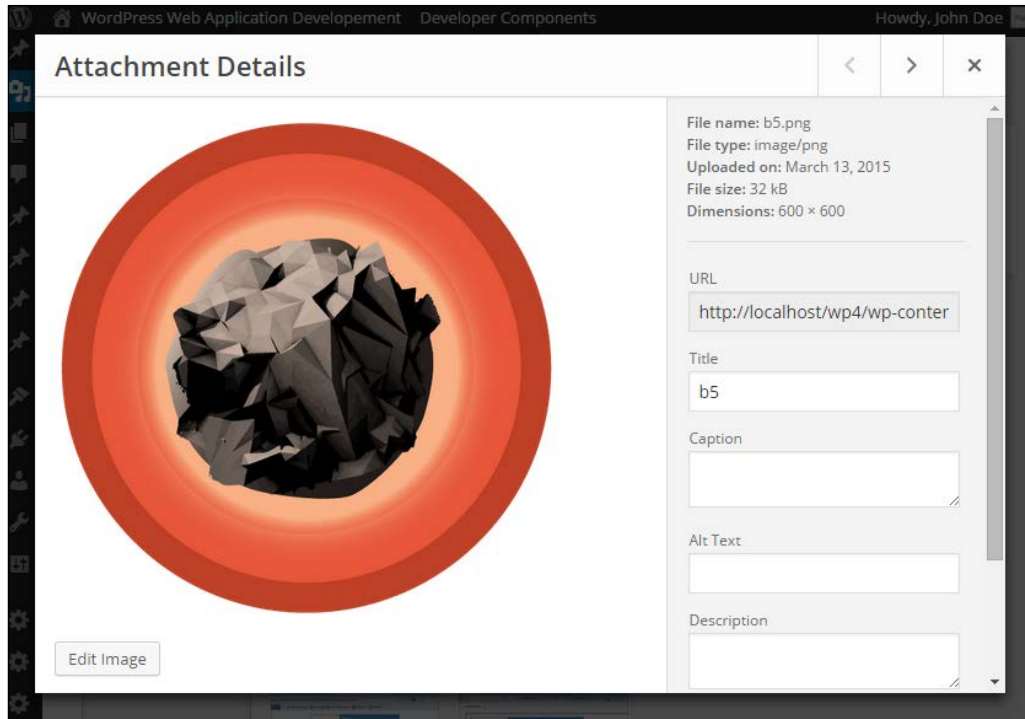
The next thing we need to know is how to update translation files. Assume that we have added new functionality to the plugin with new strings. Then, we need to update our translation files to include the new strings. So, first, we open the `.pot` file using PoEdit. Then, we click the **Update** button on the menu at the top. PoEdit will search for all the translations again and update the list. Once it's completed, you can save the `.pot` file again. You have to repeat this process for all other translation files. The only difference is that you will have to add translations to new strings before saving them again. By following this process, you can make the plugins ready for translation to any language you want.

## Working with media grid and image editor

In normal web applications, we let users upload images and files using the HTML file upload field. These files are uploaded into a specific server location. However, what if the admin wants to view the files or edit them? We have to manually download the files, make the modifications and upload them again. WordPress offers media management section by default. This section is improved in WordPress 4.0+ versions to include the media grid and image editor. This is a great feature for managing media as administrators. You can use this to let your users upload images and edit them before saving them in the WordPress backend. Since it's a built-in feature, you don't have to develop any functionality for image uploading in WordPress applications. Let's see the default display of the new media grid using the following screenshot:



Once you click on a single item, all the details will be displayed in a popup menu with the ability to navigate between the other media items. Also, for images, it lets us edit them instantly without needing any tool. The following screenshot previews the media item information screen:



As a developer, you have to get the support of the WordPress media uploader and image editor to let users upload and manage images, especially in the admin section.

## Introduction to the post editor

The WordPress post editor provides amazing features to edit the content of your site. It's improving with every version and now we have come to a stage where we are considering the frontend direct content editing. The default post editor provides lots of built-in items to format your content using HTML tags. Also, we use the post editor for adding shortcodes to content through buttons.

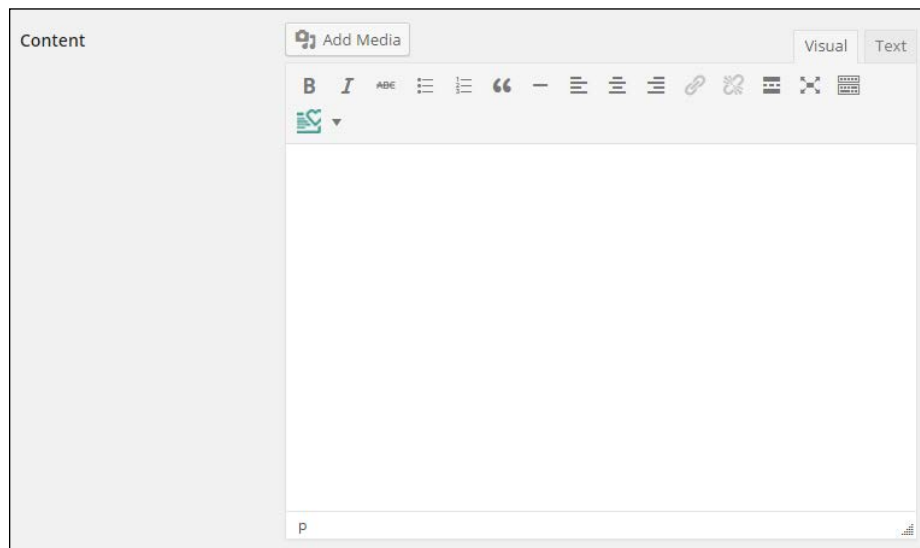


## Using the WordPress editor

In web applications, we usually need to use text areas to get descriptive information from the user. In such scenarios, we just put the default HTML textarea field or use a comprehensive editor such as TinyMCE. Adding and configuring this types of editor usually takes a lot of time. In WordPress, we have the ability to use a built-in editor anywhere you wish. Normally, we see it as the post/page editor. However, we can use it on both the frontend and backend of the application by just including the `wp_editor` function. This will create a nice content editor similar to the WordPress post editor. The following code shows the syntax of the `wp_editor` function:

```
wp_editor( $content, $editor_id, $settings = array() );
```

Once this line is used, you will get a great looking content editor, as shown in the following screenshot:



Once it's enabled, users can edit and format the content without any issues. Also, it offers the **Add Media** button, allowing users to upload images instantly and place them inside the content. In a normal application, we have to upload the file separately, get the URL manually, and add it to the content. So, developers should use `wp_editor` whenever possible instead of default text areas or manually adding editors such as TinyMCE.

## Video embedding

Video embedding is also a quite useful feature in developing applications. These days, most applications use videos to provide content. The WordPress editor allows you to just paste the URL as text and get the embedded video as the output. Without this feature, we need to upload videos or get the link, find the embedded code, and display the video manually. In an application where you need to embed videos, this feature becomes handy.

This feature doesn't work with any URL. WordPress provides a list of predefined providers with the support URL embedding. The following are some of the popular providers that are supported through this feature:

- Vimeo
- YouTube
- Twitter
- Flickr
- Instagram
- SlideShare

You can find the complete providers list at <http://codex.wordpress.org/Embeds>.

## 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 among the developers community. Let's get a brief introduction to 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 result 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 called `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 `$group` parameter defines the group name of the cached data. It's somewhat similar to namespaces, where we are allowed to create the same class inside multiple namespaces. By defining the `$group` parameter, we allow the possibility of creating duplicate cache keys in different groups. The fourth and final parameter defines the expiration 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 caching functionality for simple use cases. However, this might not be the best solution for larger web applications.



The nonpersistent nature of WordPress cache is a limitation where you lose all the cached data on page refresh.

More information about WordPress caching can be accessed from the codex at [http://codex.wordpress.org/Class\\_Reference/WP\\_Object\\_Cache](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 for providing caching inside WordPress:

- W3 Total Cache: <http://wordpress.org/plugins/w3-total-cache/>
- WP Super Cache: <http://wordpress.org/plugins/wp-super-cache/>

These are the most popular caching plugins, exceeding over seven million downloads in combination inside the WordPress plugin directory. 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 the caching functions by providing database-level cache for temporary time intervals. Compared to caching, transients are used by many developers for working with large web applications. Transient functions work in a manner similar to caching functions, where we have functions for settings 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 the transient functions is similar to the caching function with the exception of the group parameter. 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 that you use transients instead of caching functions.

## Testing

Application testing is another critical task that is used to identify potential defects before releasing to the live environment. Testing is mainly separated into two areas called unit testing and integration testing. Unit testing is used to test each small component independent 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 about existing test cases as it's not available inside the core. You can access a complete list of test cases at <http://unit-tests.svn.wordpress.org/trunk/tests/>. Make sure that you get the knowledge about testing WordPress by going through the existing 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. However, not many people know that the reason behind the hacking of most WordPress sites is due to 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 called *Hardening WordPress* for defining the necessary security constraints. You can read this guide at [http://codex.wordpress.org/Hardening\\_WordPress](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 the unused plugins and themes
- Check third-party plugins for malicious code before usage
- Move the `wp-config.php` file from the default directory
- Restrict the 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 and/or whitelisted IPs

There can be unlimited ways of breaking 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 for securing our applications. Here is a list of most popular security plugins provided in the WordPress plugin directory:

- iThemes Security: <http://wordpress.org/plugins/better-wp-security/>

- WP Security Scan: <http://wordpress.org/plugins/wp-security-scan/>
- BulletProof Security: <http://wordpress.org/plugins/bulletproof-security/>
- Secure WordPress: <http://wordpress.org/plugins/secure-wordpress/>

## Introduction to multisite

WordPress provides a module called multisite where you can create multiple networks of WordPress sites using a single installation. All the sites in the network share the same files. These sites can be installed as sub folders of the main site or sub domains. We need to know how WordPress multisite is used and how it can support web application development.

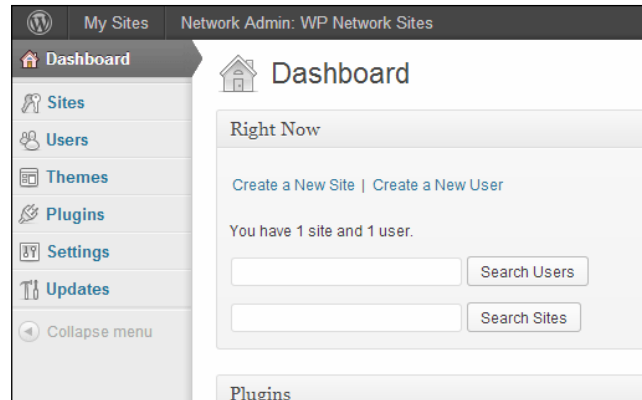
Let's identify the common usages of WordPress multisite:

- It lets users create their own blog, website, or product selling website within your site. This is a widely used technique where you let your users purchase a membership to manage their site within your network.
- It manages multiple products. WordPress theme and plugin developers use this technique to create demo sites for their plugins and themes using a single installation.
- It manages the branches of a large scale organization. Many large organizations have branches in multiple countries. So, multisite allows them to manage a separate site for each country within the network. Since all branches have similar features, multisite makes it very easy to manage.

Having looked at the practical use cases of WordPress multisite, now we have to know why multisite should be preferred over several single sites. Let's consider the advantages of using multisite:

- WordPress themes and plugins are shared across multiple sites
- You can manage multiple sites within one hosting account
- You can add/upgrade/delete plugins and themes once without duplicating these tasks for all sites
- You can control all the sites and users as a single super admin
- Upgrading and customizing the site is less time-consuming and effective
- You can dynamically create a new site any time within minutes, rather than wasting time on installing separate sites

Based on the practical scenarios and advantages, we can clearly see that we need to have multiple sites that use similar sets of features to get the most out of multisite features. Consider the following screenshot for the superadmin section of a multisite network:



As you might notice, this screen is completely different to the normal admin section of WordPress. Here, we have a menu item called **Sites**, which is used to add/edit/delete new sites inside the network. The superadmin of the network can manage these sites while an admin can manage a single site within the network.

Also, there are important sections called **Themes**, **Plugins**, **Settings**, and so on. These sections are used globally across all plugins. A plugin or theme activated from this section will act as a network-activated theme or plugin. If you deactivate them, it will affect all the sites within the network.

Managing WordPress multisite is an advanced topic that is beyond the scope of this book. What we need to know is how and when to use multisite in web application development. In general, WordPress web applications will be developed for direct clients, and hence, multisite may not play a vital role. However, if you are developing an application that provides a service to the user, then multisite could be an awesome option.

## Time for action

Throughout this book, we developed various practical scenarios to learn the art of web application development. Here, we have the final set of actions before we complete this book. By now, you should have all the knowledge of getting started with WordPress web development. After reading this chapter, you need to try the following set of actions for getting experienced with the process:

- Translate the Portfolio Manager plugin using the technique discussed in this chapter
- Figure out practical scenarios for implementing Cache and Transients

## Summary

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. However, 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.

We started this book by discussing how WordPress can be adapted to web applications and developed a simple question-answer interface using existing features. Then, we looked at user and database management capabilities while initiating the development of our portfolio management plugin. Also, we looked at the possibilities of extending core features as well as techniques for developing extensible plugins.

Then, we looked at customizing the backend features as well as the frontend features with the use of widgets and themes. The use of open source plugins and libraries is a key component in web application development, and hence, we had an in-depth look at the integration of open source libraries. We developed a simple API to understand the importance of APIs in web development. Finally, we integrated all the developed components into our application and looked at the possibilities of securing and improving the performance of web applications.

In this book, we developed an application structure considering the best practices of general web application development. The 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 that 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 that you follow the website content as it will be updated regularly with resources related to WordPress web application development. Also, we will transform 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.





# Configurations, Tools, and Resources

In this appendix, we will set up and configure WordPress and necessary tools to follow the demo application in this book. You can find a list of resources and tutorials on libraries and plugins used in this book. Let's start by configuring and setting up WordPress.

## Configure and set up WordPress

WordPress is a CMS that can be installed in a few minutes with an easy setup guide. Throughout this book, we are implementing a personal portfolio management application with advanced users. This short guide is intended to help you set up your WordPress installation with necessary configurations to be compatible with the features of our application. Let's get started!

### Step 1 – downloading WordPress

We are using WordPress 4.2.2 as the latest version available at the time of writing this book, so we have to download version 4.0 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 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 that you provide write permission for the `wp-config.php` file before starting the installation. Generally, we can use 755 permissions for directories and 644 permissions for files. You can learn more about WordPress file permissions at [http://codex.wordpress.org/Hardening\\_WordPress#File\\_Permissions](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 the local web server. There are ways of working in the local environment:

- Create a virtual host for running the application
- Use a localhost for running 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, we can match a custom URL to refer to our application. This method is preferred in web application development as the migration from local to 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 an actual online website.

By using an actual server URL for virtual host, we can directly export the local database into the server without changes.

The following resources will help you to set up virtual hosts on different operating systems:

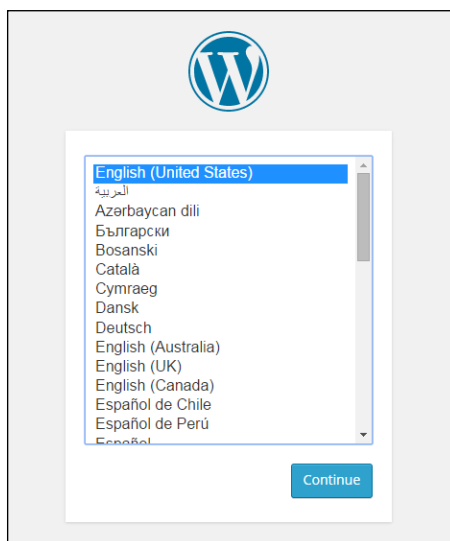
- **Windows (Wamp)** (<http://www.kristengrote.com/blog/articles/how-to-set-up-virtual-hosts-using-wamp>)
- **Mac** (<http://goo.gl/mZfVCi>)
- **Fedora** (<http://www.techchorus.net/setting-apache-virtual-hosts-fedora>)
- **Ubuntu** (<https://www.digitalocean.com/community/articles/how-to-set-up-apache-virtual-hosts-on-ubuntu-12-04-lts>)

## Using a localhost

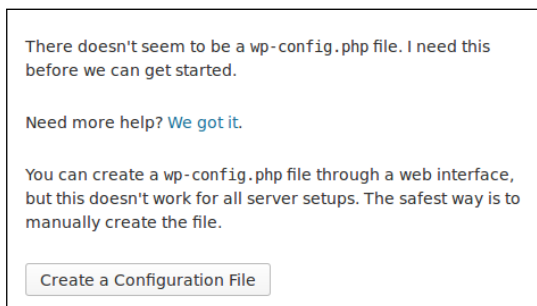
The second and commonly used method is to use a localhost as the URL to access the web application. Once the application folder is created inside the web root, we can use `http://localhost/application_folder_name` to access the application.

## Step 4 – installing WordPress

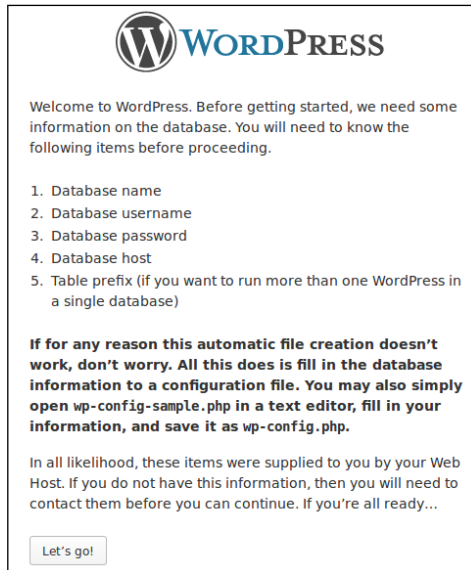
Open a web browser and enter your preceding application URL to get the initial screen of the WordPress installation process, as illustrated in the following screenshot:



We have to select the language for the site as the initial step from WordPress Version 4.0. Select the language as **English (United States)** for this installation and click the **Continue** button. Then 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.



Next, click the **Create a Configuration File** button to load the screen shown in the following image:



The image shows the WordPress database configuration screen. At the top is the WordPress logo. Below it, a welcome message states: "Welcome to WordPress. Before getting started, we need some information on the database. You will need to know the following items before proceeding." A numbered list follows: 1. Database name, 2. Database username, 3. Database password, 4. Database host, 5. Table prefix (if you want to run more than one WordPress in a single database). A bolded note says: "If for any reason this automatic file creation doesn't work, don't worry. All this does is fill in the database information to a configuration file. You may also simply open wp-config-sample.php in a text editor, fill in your information, and save it as wp-config.php." Another note says: "In all likelihood, these items were supplied to you by your Web Host. If you do not have this information, then you will need to contact them before you can continue. If you're all ready..." At the bottom is a button labeled "Let's go!"

The preceding screen displays all the information needed to continue with the installation. Click the **Let's go!** button after reading the contents to get the next screen, as shown in the following image:



The image shows the WordPress database configuration form. At the top is the WordPress logo. Below it, a message states: "Below you should enter your database connection details. If you're not sure about these, contact your host." The form has five input fields, each with a label and a description: "Database Name" (wpapplications) with description "The name of the database you want to run WP in.", "User Name" (username) with description "Your MySQL username", "Password" (password) with description "...and your MySQL password.", "Database Host" (localhost) with description "You should be able to get this info from your web host, if localhost does not work.", and "Table Prefix" (wp\_) with description "If you want to run multiple WordPress installations in a single database, change this." At the bottom is a button labeled "Submit".

Here, we have to enter the details for connecting to the database. Use the details in the database creation process for defining 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, such as a random string, for your tables to improve the security of your application. Once all the details are entered, hit the **Submit** button to get the next screen, as shown in the following image:

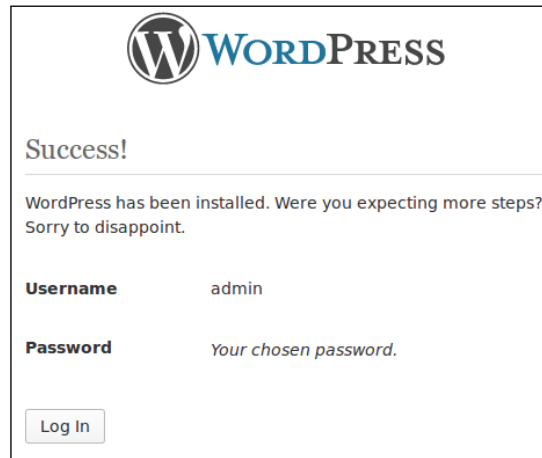


Also, we can use the WP Better Security plugin to generate a random prefix and update the database.



Click the **Run the install** button to get the next screen, as shown in the following image:


Fill the form with the requested details. By default, WordPress provides a blank field for the username. Ideally, you should be using a custom username as the admin role, instead of admin as the username, to improve the security of the application. Once all the details are filled, submit the form to complete the installation, and get the following screen:



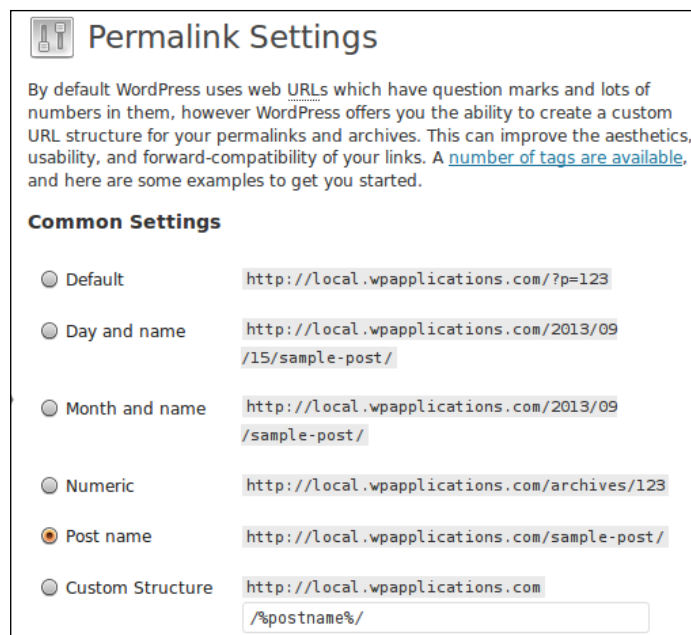
Details of your admin account will be displayed in this screen. Click the **Log In** button to get the login form and log into the admin area. Now, we are ready to go!

## Step 5 – setting up permalinks

Permalinks allow you to define the custom URL structure for your posts, pages, and custom URLs using `mod_rewrite`.

[  Your Apache installation must have `mod_rewrite` installed and turned on for permalinks to work. ]

By default, WordPress uses query parameters to load posts and pages through the ID. Usually, we change the existing URL structure to provide a pretty URL. So, navigate to the **Settings | Permalinks** section 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:



## Step 6 – downloading the Responsive theme

We are using 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 **Appearance | 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:

- **The Responsive theme:** This theme is developed by CyberChimps. You can find this at <http://goo.gl/Uf9Mp1>.
- **The Members plugin:** This plugin is developed by Justin Tadlock. This can be found at <http://goo.gl/HuhDax>.
- **The Rewrite Rules Inspector plugin:** This plugin is developed by Daniel Bachhuber and Automattic. You can find this at <http://goo.gl/oBVJmL>.
- **The Posts 2 Posts plugin:** This plugin is developed by Alex Ciobica and scribu. This is available at <http://goo.gl/8pQGmT>.
- **Pods – Custom Content Types and Fields:** You can find this framework at <http://goo.gl/ixMspf>.
- **The Custom List Table Example plugin:** This plugin is developed by Matt Van Andel, and you can find this at <http://goo.gl/3tnfmf>.
- **Backbone.js:** This library can be found at <http://goo.gl/VyhED1>.
- **Underscore.js:** This library is available at <http://goo.gl/aZ42YD>.
- **PHPMailer:** This library can be found at <http://goo.gl/VX90ym>.

## Online resources and tutorials

Web application development with WordPress has still not matured, so you will find various perspectives from various people about using WordPress as an application development framework. This section provides various tutorials and articles for understanding various perspectives on using WordPress for web application development:

- *Wordpress As An Application Platform*, Tom McFarlin (<http://goo.gl/gONP3i>)

- *WordPress For Application Development*, Tom McFarlin (<http://goo.gl/ubDasf>)
- *My Thoughts on Building Web Applications with WordPress*, Tom McFarlin (<http://goo.gl/fTUqQf>)
- *Why WordPress Isn't Viewed as an Application Framework*, Tom McFarlin (<http://goo.gl/Ophmak>)
- *Using WordPress as a Web Application Framework*, Harish Chouhan (<http://goo.gl/BFHqVB>)
- *Build Powerful Websites and Applications with WordPress*, Piklist (<http://goo.gl/WYqRNh>)
- *Build an App With WordPress – The compulsory todo list*, Harley Alexander (<http://goo.gl/rwMB6c>)



# Index

## A

### **action hooks**

- planning, for layouts 247, 248
- used, for extending home page template 244

### **active/inactive status, checking**

- constants 148
- functions 147
- plugin status, checking 148

### **admin dashboard**

- about 9
- appearance 10
- defining 168
- pages 10
- posts 10
- settings 10
- users 10

### **admin list tables**

- available custom columns, listing 197
- checkbox for records, displaying 197
- column default handlers, implementing 196
- comments list 192
- custom class, defining 194
- custom column handlers, implementing 195, 196
- custom list, adding as menu page 199
- generated list, displaying 199-202
- implementing 194
- initial configurations, creating 195
- instance variables, defining 194
- list data, retrieving 198
- list of bulk actions, creating 198
- post list 184
- sortable columns of list, defining 198

- user list 191
- using 183, 194
- working with 183

### **admin theme plugins**

- Bootstrap Admin 204
- Slate Admin theme 204

### **admin toolbar**

- customizing 168, 169
- items, managing 170-172
- removing 169

### **AJAX-based star rating system**

- including 210

### **Amazon Product Advertising API**

- URL 296

### **API**

- about 296
- advantages 296

### **API access tokens**

- integrating 306-311

### **API client**

- about 298
- building 298-301

### **API documentation**

- providing 311, 312

### **API server 298**

### **API user authentication**

- integrating 303-305

### **application data tables 80**

### **application frontend menu**

- generating 233
- navigation menu, creating 233-235
- user specific menus, displaying 236

### **application layouts**

- widgetizing 224

### **Application Programming**

Interface. *See* API

- application testing** 351
- application URL**
  - configuring 358
  - localhost, using 359
  - virtual host, creating 358
- application users**
  - registering 43, 44

## B

- Backbone.js** 255
  - about 254
  - advantages 256
  - URL 254
- Bootstrap Admin**
  - URL 204
- built-in delete functions**
  - delete\_post\_meta 85
  - delete\_user\_meta 85
  - wp\_delete\_post 85
- built-in insert functions**
  - add\_option 84
  - wp\_insert\_comment 84
  - wp\_insert\_post 84
- built-in update functions**
  - update\_post\_meta 84
  - update\_user\_meta 84
  - wp\_update\_term 84
- BulletProof Security**
  - URL 353

## C

- caching**
  - about 350
  - URL 350
- comment-related tables**
  - about 74
  - wp\_commentmeta 74
  - wp\_comments 74
- comments list** 192
- components, WordPress**
  - about 7
  - admin dashboard 9
  - application layout, customizing 9
  - plugins 10

- widgets 11
- WordPress page layout 8
- WordPress themes 8
- custom API**
  - creating 302, 303
- custom content types**
  - about 96
  - articles 99
  - books 99
  - class-wpwa-custom-post-manager.php 100
  - class-wpwa-template-loader.php 101
  - configurations, implementing 103, 104
  - custom fields, with meta boxes 113, 114
  - custom taxonomies, creating 108-111
  - implementing, for portfolio
    - application 100-102
  - models 101
  - permissions, assigning to projects 107, 108
  - permissions, assigning to project
    - type 111-113
  - planning, for applications 97
  - Pods framework 132-135
  - projects 97
  - projects class, creating 104-107
  - role, in web applications 96
  - services 98
  - templates 101
- custom e-mail sending functionality**
  - creating 274
- custom field data**
  - persisting 123-126
- Custom List Table Example plugin**
  - URL 364
- custom pages**
  - custom menu pages 175
  - options pages 175
- custom post type messages**
  - customizing 127-129
- custom post type relationships** 129-132
- custom routing technique**
  - application-specific theme, creating 222
  - direct template inclusion 221
  - pure PHP templates, using 220
  - templates, using in WordPress 220
  - theme, versus plugins based templates 222
  - using, in custom templates 219

## D

### database, querying

- custom tables, querying 85, 86
- existing tables, querying 84
- posts, working with 86
- WordPress query classes 88
- WP\_Query class, extending for applications 87

### data selecting functions

- get\_option 85
- get\_posts 85
- get\_users 85

### default post statuses, post life cycle

- Auto-Draft 188
- Draft 188
- Future 187
- Inherit 188
- Pending 188
- Private 188
- Published 187
- Trash 188

### default roles

- admin 40
- author 40
- contributor 40
- editor 40
- subscriber 40
- superadmin 40

### demo application

- URL 337

### developer profile page, with Backbone.js

- about 257-260
- events, integrating to Backbone.js
  - views 268, 269
- models, creating for server 269
- models, creating in server 270-272
- models, validating for server 269
- projects, creating from frontend 266-268
- projects list, displaying on page
  - load 262-266
- structuring, with Backbone.js 260-262
- structuring, with Underscore.js 260-262

### developer's list page

- URL 323

### development plan, portfolio management application

- about 13
- application goals 13
- features, planning 15
- functions, planning 15
- planning 14
- target audience 13
- user roles 15

## E

### entity relationship diagram, WordPress

- reference 75

### existing tables

- about 70
- categorizing 70
- comment-related tables 74
- post-related tables 72
- term-related tables 73
- user-related tables 71
- wp\_links table 75
- wp\_options table 75

### existing tables, adapting into web applications

- about 75
- other tables 78
- post-related tables 76
- term-related tables 77
- user-related tables 76

### existing tables, querying

- about 84
- records, deleting 85
- records, inserting 84
- records, selecting 85
- records, updating 84

### Exploit Scanner

- URL 292

### extended lists

- building 193

### extensible plugins

- about 148
- allowed types of images, customizing 155-157

- extensible file uploader plugin,
  - creating 150, 151
- file fields, converting with jQuery 151, 152
- file uploader, planning for portfolio application 149
- file uploader plugin, extending 155
- media uploader, integrating to
  - buttons 152-154
- project screens, loading 157-159
- project screens, saving 157-159
- used, with custom actions and filters 159, 160
- used, with WordPress core actions 149
- used, with WordPress filters 149

## F

### Facebook Graph API

- URL 296

### Fedora

- URL 358

### Front Controller pattern 218

#### frontend login

- about 62
- activation status, checking 66, 67
- creating 62-64
- login form, displaying 64-66

#### frontend registration

- automatic login after registration, implementing 60
- custom template implementation 46
- custom templates, creating 52
- do\_action function, using 51
- functions access, controlling 50
- implementing 44
- page template implementation 45
- registration form, designing 52-54
- registration form submission,
  - handling 55-57
- registration process, planning 54, 55
- registration success path,
  - exploring 58-60
- shortcode implementation 44
- simple router, building for user module 46
- system users, activating 61, 62

### functions, portfolio management application

- developer profile management 16
- frontend login and registration 16
- notification service 16
- responsive design 16
- settings panel 16
- third party libraries 16
- XML API 16

### functions, WordPress options API

- add\_option 181
- delete\_option 181
- get\_option 181
- update\_option 181

## G

### get\_template\_part function 117

### Google Maps API

- URL 296

## H

### high quality plugins

- developing 165

### home page template

- designing 231, 232
- extending, with action hooks 244
- widgets, customizing to enable extendable locations 245-247

## I

### image editor

- working with 346, 347

### input control types

- URL 239

### instance variables

- base\_path 103
- projects 103
- template\_parser 103

### internationalization

- about 340
- plugin translations, creating 341
- WordPress translation support 340

### iThemes Security

- URL 352

## L

### layout creation techniques, web application

- about 218
- page templates 219
- shortcodes 219

### LinkedIn API

- URL 282

### login strategies

- configuring 279, 280
- LinkedIn account authentication, implementing 281-283
- LinkedIn account, verifying 283, 284

## M

### Mac

- URL 358

### Machine Object (MO) file 342

### main navigation menu

- customizing 173, 174
- features, adding with custom pages 175
- new menu items, creating 174

### master tables 80

### media grid

- working with 346, 347

### Members plugin

- URL 43

### multisite

- about 353, 354
- advantages 353
- scenarios 354
- usages 353

## O

### OAuth library

- URL 280

### OpenAuth 277

### open-closed principle

- about 140
- URL 140

### open source JavaScript libraries, in WordPress

- about 253, 254
- Backbone.js 254, 255
- Backbone.js, integrating 256, 257

- code structuring, defining 255, 256
- developer profile page, creating with Backbone.js 257-260
- Underscore.js, integrating 256, 257

### open source libraries

- advantages 252
- online references 364
- selecting 252
- used, in WordPress core 252, 253

### options pages

- application options panel, building 178-180
- building 175, 176
- creating 176
- custom layout, creating for 176, 177
- WordPress options API, using 181, 182

## P

### page templates

- about 45
- cons 45
- pros 45

### permalinks

- about 362
- setting up 362

### PHPMailer

- custom functions, creating 274-276
- custom version, creating of pluggable wp\_mail function 274
- loading 274-276
- URL 273
- used, for custom e-mail sending 273
- used, within WordPress 273

### PHPUnit

- URL 352

### pluggable functions

- wp\_logout 161
- wp\_mail 161
- wp\_new\_user\_notification 161

### pluggable plugins

- about 161-163
- pluggable functions, using 164

### pluggable templates

- creating 242
- creating, in WordPress 242, 243



## **plugin dependencies**

handling 145-148

## **plugins**

about 10

online references 364

using 292

## **plugin translations**

creating 341

language files, loading 345

POT file, creating with PoEdit 342-344

WordPress language, changing 345, 346

## **Pods framework**

about 132

features 132

for custom content types 132-135

selecting, for web development 135, 136

URL 133

## **PoEdit**

about 342

URL 342

used, for creating POT file 342-344

## **Portable Object (PO) file 342**

## **Portable Object Template (POT) file 342**

## **portfolio application**

custom tables, creating 81-83

development plan 13

home page, building 223

implementing 336

integrating 316

structuring 316

tables, planning 80

types of tables, in web applications 80

widget 223, 224

## **post editor 347**

## **post filters**

reference 88

## **post list**

about 184

custom actions, creating for custom posts 184, 185

custom filters, creating for custom post types 186, 187

custom list columns, displaying 189, 190

custom post status links, creating 187-189

## **post-related tables**

about 72, 76

hotel reservation system scenario 77

online shopping cart scenario 77

project management application scenario 77

wp\_postmeta table 72

wp\_posts table 72

## **Posts 2 Posts plugin**

URL 364

# **Q**

## **question-answer interface**

building 18

comments template, customizing 22, 23

prerequisites 18

question list, generating 30-32

questions, creating 19-22

status of answers, changing 23-27

status of answers, saving 28, 29

## **questions plugin**

design of questions, customizing 32

features, enhancing 32

questions, approving 33

questions, categorizing 33

questions, rejecting 33

star rating, adding to answers 33

# **R**

## **Representational State Transfer (REST) 254**

## **responsive nature, of admin**

dashboard 209, 210

## **Responsive theme**

activating 363

downloading 363

URL 364

## **RESTful architecture**

URL 254

## **restructured application, portfolio application**

AJAX-based filtering, enabling 323-326

developer list template, designing 322, 323

developer model, building 321

working with 320

## **Retina Press**

URL 204

## **reusable libraries**

- creating, with plugins 142
- plugin dependencies, handling 145-148
- template loader plugin, planning 142-144
- template loader plugin, using 144, 145

## **Rewrite Rules Inspector plugin**

URL 364

## **router**

- building, for user module 46
- query variables, adding 47
- requirements 46
- rewriting rules, flushing 48, 49
- routing rules, creating 47

# **S**

## **Secure WordPress**

URL 353

## **security 352**

## **shortcode**

- cons 45
- implementing 44
- pros 45

## **Slate Admin theme**

URL 204

## **subscriber notifications**

- scheduling 331-333
- sending, through e-mails 333-336

# **T**

## **tables in web applications**

- about 80
- application data tables 80
- master tables 80
- transaction tables 80

## **table creation query**

- reference 82

## **template engine**

- about 115
- comparing, with template loader 122
- first template, creating 118-121
- simple custom template loader, building 116, 117

templates without parts 117

templates with parts 117

## **template execution hierarchy**

- Archive pages 215
- Single pages 216
- Single posts 216
- URL 215

## **template loader**

- integrating, into user manager 318-320

## **template loader dependencies**

- adding 317

## **template loader plugin**

- building 142
- planning 142-144
- using 144, 145

## **term-related tables**

- about 73
- wp\_term\_relationships 73
- wp\_terms 73
- wp\_term\_taxonomy 73

## **Theme Authenticity Checker (TAC)**

URL 292

## **Theme Check**

URL 293

## **theme customizer**

- custom options, adding 238-240
- used, for managing options 237
- used, for managing widgets 237
- widgets, handling 240-242

## **third-party libraries**

- using 292

## **Timber plugin**

URL 116

## **transaction tables 80**

## **transients 351**

## **Twig documentation**

URL 116

## **Twitter REST API**

URL 296

# **U**

## **Ubuntu**

URL 358

## **Underscore.js**

URL 364

**user authentication, with OpenAuth**

- implementing 277-279
- library, initializing 287, 288
- LinkedIn app, building 285, 286
- login strategies, configuring 279, 280
- strategies, requesting 287
- to application 289-292

**user capabilities**

- about 41
- creating 42
- default capabilities 42, 43

**user list 191****user management**

- about 36
- plugin, preparing 36, 37

**user profile**

- field values, updating 328-331
- updating, with additional fields 326-328

**user-related tables**

- about 71, 76
- wp\_usermeta table 71
- wp\_users table 71

**user roles**

- about 37
- adding 38, 39
- application installation, using 38
- application user roles, creating 38
- default and custom roles, selecting
  - between 40, 41
- default roles 40
- existing user roles, removing 41
- options, for implementing 38, 39
- plugin activation, using 38
- URL 38

**user roles, portfolio application**

- admin 15
- developer 15
- members 15
- subscribers 15

**V****vhosts 358****video embedding 349****visual presentation, for admin screens**

- about 202
- admin theme, creating 205-208

existing themes, using 203

third-party admin themes, using 203-205

**W****W3 Total Cache**

URL 350

**web application development, with built-in features**

- about 4
- actions 7
- admin dashboard 7
- caching 6
- database management 5
- filters 7
- media management 5
- plugins 6
- routing 5
- scheduling 6
- template management 5
- themes 6
- user management 5
- widgets 6
- XMR-RPC API 6

**web application registration process**

- detailed information, requesting 44
- user accounts, activating 44
- user-friendly interface 43

**widget**

- \_\_construct function 226
- about 11, 12, 223
- creating 225-231
- form function 226
- update function 226
- widget function 226

**widgets, application layouts**

- after\_title 225
- after\_widget 224
- before\_title 225
- before\_widget 224
- id 224
- name 224

**Windows (Wamp)**

URL 358

**WordPress**

- about 167, 251
- application folder, creating 358

- application URL, configuring 358
- application, using 364
- as CMS 2
- as web application framework 3
- auto saving 92
- components, identifying 7
- configuring 357
- considerations 90
- downloading 357
- final thoughts 337
- guidelines 17
- installing 359-362
- limitations 17, 18, 90
- meta tables, using 92
- MVC, versus event-driven architecture 4
- structure, page layout 8
- permalinks, setting up 362
- pluggable templates, creating 242, 243
- plugin, activating 363
- post revisions 91
- reference 4
- Responsive theme, activating 363
- Responsive theme, downloading 363
- revisions, disabling 92
- revisions, enabling 91
- themes 8
- transaction support 91
- WordPress application frontend**
  - about 214
  - automating 249
  - extendable layout, creating 249
  - template execution hierarchy 215-217
  - web application frameworks, template execution process 217, 218
  - WordPress theme, file structure 214, 215
- WordPress codex**
  - URL 112
- WordPress database**
  - about 70
  - existing tables, exploring 70
  - extending, with custom tables 79
  - querying 84
- WordPress features**
  - about 349
  - caching 350
  - security 352
  - testing 351
  - transients 351
- WordPress, for web application development**
  - online resources and tutorials 364
- WordPress options API**
  - URL 181
- WordPress plugins**
  - about 139
  - architecture 140, 141
  - defining 140
  - extensible plugins 148
  - pluggable plugins 161-163
  - reusable libraries, creating with 142
  - used, for web development 141, 142
- WordPress transient API 351**
- WordPress translation support**
  - about 340
  - translation functions 341
- WordPress web applications**
  - reference 92
  - URL 191
- WordPress XML-RPC API**
  - for web applications 297
  - URL 297
- WP\_Date\_Query class 90**
- wpdb class**
  - \$wpdb->get\_results("select query") 85
  - \$wpdb->get\_row('query') function 86
  - \$wpdb->query('query') function 85
  - reference 84
- wp\_mail function**
  - about 273
  - wp\_mail\_charset 274
  - wp\_mail\_content\_type 274
  - wp\_mail\_from 273
  - wp\_mail\_from\_name 274
- WP\_Meta\_Query class 90**
- WP\_Query class**
  - extending, for applications 87, 88
  - reference 86
- WP Security Scan**
  - URL 352
- WP Super Cache**
  - URL 351
- WP\_Tax\_Query class 90**

**WP\_User\_Query class**

about 89

filtering methods 89

reference 89

**WP\_UWP\_Comment\_Query class**

about 89

reference 90

**Y**

**YouTube API**

URL 296



## Thank you for buying **WordPress Web Application Development** *Second Edition*

### 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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

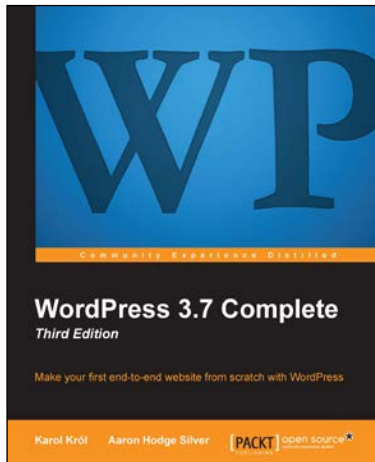
### About Packt Open Source

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 licenses, 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](mailto: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, then please 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.



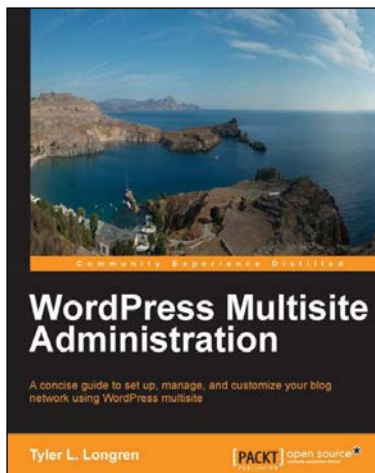
## WordPress 3.7 Complete *Third Edition*

ISBN: 978-1-78216-240-7

Paperback: 404 pages

Make your first end-to-end website from scratch with WordPress

1. Learn how to build a WordPress site quickly and effectively.
2. Find out how to create content that's optimized to be published on the Web.
3. Learn the basics of working with WordPress themes and playing with widgets.



## WordPress Multisite Administration

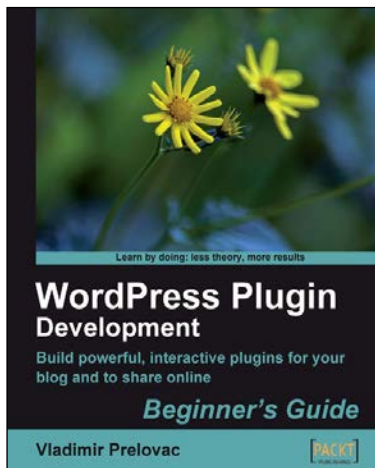
ISBN: 978-1-78328-247-0

Paperback: 106 pages

A concise guide to set up, manage, and customize your blog network using WordPress multisite

1. Learn how to configure a complete, functional, and attractive WordPress Multisite.
2. Customize your sites with WordPress themes and plugins.
3. Set up, maintain, and secure your blog network.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



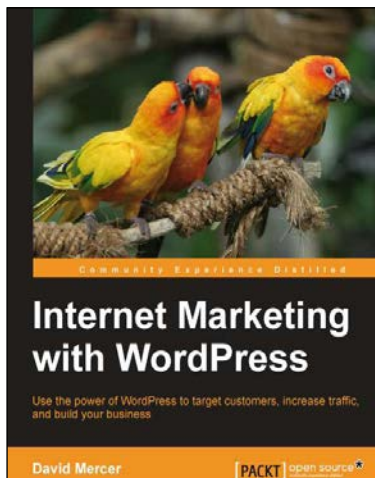
## WordPress Plugin Development: Beginner's Guide

ISBN: 978-1-84719-359-9

Paperback: 296 pages

Build powerful, interactive plugins for your blog and to share online

1. Everything you need to create and distribute your own plug-ins following WordPress coding standards.
2. Walk through the development of six complete, feature-rich, real-world plug-ins that are being used by thousands of WP users.
3. Written by Vladimir Prelovac, WordPress expert and developer of WordPress plug-ins such as Smart YouTube and Plugin Central.



## Internet Marketing with WordPress

ISBN: 978-1-84951-674-7

Paperback: 112 pages

Use the power of WordPress to target customers, increase traffic, and build your business

1. Get practical experience in key aspects of online marketing.
2. Accurately identify your business objectives and target audience to maximize your marketing efficiency.
3. Create and deliver awesome SEO-enhanced, targeted content to drive large numbers of visitors through your blog.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles