

O'REILLY®

Second  
Edition

# Building Web Apps with WordPress

WordPress as an Application Framework



Brian Messenlehner &  
Jason Coleman  
Foreword by Chris Lema

## 1. Foreword

### 2. Preface

- a. Who This Book Is For
- b. Who This Book Is Not For
- c. What You'll Learn
- d. About the Code
- e. Conventions Used in This Book
- f. Using Code Examples
- g. O'Reilly Online Learning
- h. How to Contact Us
- i. Acknowledgments

### 3. 1. Building Web Apps with WordPress

- a. What Is a Website?
- b. What Is an App?
- c. What Is a Web App?
  - i. Features of a Web App
  - ii. Mobile Apps
  - iii. Progressive Web Apps
- d. Why Use WordPress?
  - i. You Are Already Using WordPress
  - ii. Content Management Is Easy with WordPress

iii. User Management Is Easy and Secure with WordPress

iv. Plugins

v. Flexibility Is Important

vi. Frequent Security Updates

vii. Cost

viii. Responses to Some Common Criticisms of WordPress

e. When Not to Use WordPress

i. You Plan to License or Sell Your Site’s Technology

ii. Another Platform Will Get You “There” Faster

iii. Flexibility Is Not Important to You

iv. Your App Needs to Be Highly Real Time

f. WordPress as an Application Framework

i. WordPress Versus Model-View-Controller Frameworks

g. Anatomy of a WordPress App

i. What Is SchoolPress?

ii. SchoolPress Runs on a WordPress Multisite Network

iii. The SchoolPress Business Model

iv. Membership Levels and User Roles

- v. Classes Are BuddyPress Groups
- vi. Assignments Are a CPT
- vii. Submissions Are a (Sub)CPT for Assignments
- viii. Semesters Are a Taxonomy on the Class CPT
- ix. Departments Are a Taxonomy on the Class CPT
- x. SchoolPress Has One Main Custom Plugin
- xi. SchoolPress Uses a Few Other Custom Plugins
- xii. SchoolPress Uses the Memberlite Theme

#### 4. 2. WordPress Basics

##### a. WordPress Directory Structure

- i. Root Directory
- ii. /wp-admin
- iii. /wp-includes
- iv. /wp-content

##### b. WordPress Database Structure

- i. wp\_options
- ii. Functions Found in /wp-includes/option.php
- iii. wp\_users

iv. Functions Found in /wp-includes/...

v. wp\_usermeta

vi. wp\_posts

vii. Functions Found in /wp-includes/post.php

viii. wp\_postmeta

ix. Functions Found in /wp-includes/post.php

x. wp\_comments

xi. Functions Found in /wp-includes/comment.php

xii. wp\_commentsmeta

xiii. Functions Found in /wp-includes/comment.php

xiv. wp\_terms

xv. Functions Found in /wp-includes/taxonomy.php

xvi. wp\_termmeta

xvii. wp\_term\_taxonomy

xviii. /wp-includes/taxonomy.php

xix. wp\_term\_relationships

c. Hooks: Actions and Filters

i. Actions

ii. Filters

d. Development and Hosting Environments

i. Working Locally

ii. Choosing a Web Host

iii. Development, Staging, and Production  
Environments

e. Extending WordPress

5. 3. Using WordPress Plugins

a. The General Public License, Version 2, License

b. Installing WordPress Plugins

c. Building Your Own Plugin

d. File Structure for an App Plugin

i. /adminpages/

ii. /classes/

iii. /css/

iv. /js/

v. /images/

vi. /includes/

vii. /includes/lib/

viii. /pages/

ix. /services/

x. /scheduled/

xi. /schoolpress.php

e. Add-Ons to Existing Plugins

f. Use Cases and Examples

i. The WordPress Loop

ii. WordPress Global Variables

g. Free Plugins

i. Admin Columns

ii. Advanced Custom Fields

iii. BadgeOS

iv. Posts 2 Posts

v. Members

vi. W3 Total Cache

vii. Yoast SEO

h. Premium Plugins

i. Gravity Forms

ii. BackupBuddy

iii. WP All Import

i. Community Plugins

i. BuddyPress

6. 4. Themes

a. Themes Versus Plugins

i. Where to Place Code When Developing Apps

ii. When Developing Plugins

### iii. Where to Place Code When Developing Themes

b. The Template Hierarchy

c. Page Templates

i. Sample Page Template

ii. Using Hooks to Copy Templates

iii. When Should You Use a Theme Template?

d. Theme-Related WordPress Functions

i. Using locate\_template in Your Plugins

e. Style.css

i. Versioning Your Theme's CSS Files

f. functions.php

g. Themes and CPTs

h. Popular Theme Frameworks

i. WordPress Theme Frameworks

ii. Non-WordPress Theme Frameworks

i. Creating a Child Theme for Memberlite

j. Including Bootstrap in Your App's Theme

k. Menus

i. Navigation Menus

ii. Dynamic Menus

l. Responsive Design

- i. Device and Display Detection in CSS
- ii. Device and Feature Detection in JavaScript
- iii. Device Detection in PHP
- iv. Final Note on Browser Detection

## 7. 5. Custom Post Types, Post Metadata, and Taxonomies

- a. Default Post Types and CPTs
  - i. Page
  - ii. Post
  - iii. Attachment
  - iv. Revisions
  - v. Navigation Menu Item
  - vi. Custom CSS
  - vii. Changesets
  - viii. oEmbed Cache
  - ix. User Requests
  - x. Reusable Blocks

### b. Defining and Registering CPTs

- i. `register_post_type( $post_type, $args );`

### c. What Is a Taxonomy and How Should I Use It?

- i. Taxonomies Versus Post Meta
- ii. Creating Custom Taxonomies

- iii. `register_taxonomy( $taxonomy,  
$object_type, $args )`
- iv. `register_taxonomy_for_object_type( $taxonomy, $object_type )`

#### d. Using CPTs and Taxonomies in Your Themes and Plugins

- i. The Theme Archive and Single Template Files
- ii. Good Old WP\_Query and `get_posts()`

#### e. Metadata with CPTs

- i. `add_meta_box( $id, $title, $callback,  
$screen, $context, $priority,  
$callback_args )`
- ii. Using Meta Boxes with the Block Editor

#### f. Custom Wrapper Classes for CPTs

- i. Extending WP\_Post Versus Wrapping It
- ii. Why Use Wrapper Classes?
- iii. Keep Your CPTs and Taxonomies Together
- iv. Keep It in the Wrapper Class
- v. Wrapper Classes Read Better

### 8. 6. Users, Roles, and Capabilities

#### a. Getting User Data

- b. Add, Update, and Delete Users
- c. Hooks and Filters
- d. What Are Roles and Capabilities?
  - i. Checking a User's Role and Capabilities
  - ii. Creating Custom Roles and Capabilities
- e. Extending the WP\_User Class
- f. Adding Registration and Profile Fields
- g. Customizing the Users Table in the Dashboard
- h. Plugins
  - i. Theme My Login
  - ii. Hide the Admin Bar from Nonadministrators
  - iii. Paid Memberships Pro
  - iv. PMPro Register Helper
  - v. Members
  - vi. WP User Fields

## 9. 7. Working with WordPress APIs, Objects, and Helper Functions

- a. Shortcode API
  - i. Shortcode Attributes
  - ii. Nested Shortcodes
  - iii. Removing Shortcodes

iv. Other Useful Shortcode-Related Functions

b. Widgets API

i. Before You Add Your Own Widget

ii. Adding Widgets

iii. Defining a Widget Area

iv. Embedding a Widget Outside of a Dynamic Sidebar

c. Dashboard Widgets API

i. Removing Dashboard Widgets

ii. Adding Your Own Dashboard Widget

d. Settings API

i. Do You Really Need a Settings Page?

ii. Could You Use a Hook or Filter Instead?

iii. Use Standards When Adding Settings

iv. Ignore Standards When Adding Settings

e. Rewrite API

i. Adding Rewrite Rules

ii. Flushing Rewrite Rules

iii. Other Rewrite Functions

f. WP-Cron

- i. Adding Custom Intervals
- ii. Scheduling Single Events
- iii. Kicking Off Cron Jobs from the Server
- iv. Using Server Cron Only

#### g. WP Mail

- i. Sending Nicer Emails with WordPress

#### h. File Header API

- i. Adding File Headers to Your Own Files
- ii. Adding New Headers to Plugins and Themes

#### i. Heartbeat API

### 10. 8. Secure WordPress

#### a. Why It's Important

#### b. Security Basics

- i. Update Frequently
- ii. Don't Use the Username "admin"
- iii. Use a Strong Password
- iv. Examples of Bad Passwords
- v. Examples of Good Passwords

#### c. Hardening WordPress

- i. Don't Allow Admins to Edit Plugins or Themes
- ii. Change Default Database Tables Prefix

- iii. Move wp-config.php
- iv. Hide Login Error Messages
- v. Hide Your WordPress Version
- vi. Don't Allow Logins via wp-login.php
- vii. Add Custom .htaccess Rules for Locking Down wp-admin

#### d. SSL Certificates and HTTPS

- i. Installing an SSL Certificate on Your Server
- ii. WordPress Login and WordPress Administrator over SSL
- iii. Debugging HTTPS Issues
- iv. Avoiding SSL Errors with the “Nuclear Option”

#### e. Back Up Everything!

#### f. Scan, Scan, Scan!

#### g. Useful Security Plugins

- i. Spam-Blocking Plugins
- ii. Backup Plugins
- iii. Firewall/Scanner Plugins
- iv. Login and Password-Protection Plugins

#### h. Writing Secure Code

- i. Check User Capabilities
- ii. Custom SQL Statements

iii. Data Validation, Sanitization, and Escaping

iv. Nonces

**11. 9. JavaScript Frameworks and Workflow**

a. What Is ECMAScript?

b. What Is ES6?

c. What Is ES9?

d. What Is ESNext?

e. What Is Ajax?

f. What Is JSON?

g. jQuery and WordPress

i. Enqueuing Other JavaScript Libraries

ii. Where to Put Your Custom JavaScript

h. Ajax Calls with WordPress and jQuery

i. Managing Multiple Ajax Requests

j. Heartbeat API

k. WordPress Limitations with Asynchronous Processing

l. JavaScript Frameworks

i. Backbone.js

ii. React

**12. 10. WordPress REST API**

a. What Is a REST API?

i. API

ii. REST

iii. JSON

iv. HTTP

b. Why Use the WordPress REST API?

c. Using the WordPress REST API V2

i. Discovery

ii. Authentication

iii. Routes and Endpoints

iv. Requests

v. Responses

d. Adding Your Own Routes and Endpoints

i. register\_rest\_route( \$namespace,  
\$route, \$args, \$override );

ii. Setting Up the WordPress Single Sign-On Plugin

iii. Adding the /wp-sso/v1/check Route

iv. Bundling Basic Authentication with Our Plugin

v. Using the Endpoint We Set Up to Check User Credentials

e. Popular Plugins Using the WordPress REST API

i. WooCommerce

- ii. BuddyPress
- iii. Paid Memberships Pro

## 13. 11. Project Gutenberg, Blocks, and Custom Block Types

- a. The WordPress Editor
- b. The Classic Editor Plugin
- c. Using Blocks for Content and Design
- d. Using Blocks for Functionality
- e. Creating Your Own Blocks
  - i. Minimal Block Example
- f. Using Custom Blocks to Build App Experiences
  - i. Enabling the Block Editor in Your CPTs
  - ii. Block Categories
  - iii. The Homework Blocks
  - iv. Limiting Blocks to Specific CPTs
  - v. Limiting CPTs to Specific Blocks
  - vi. Block Templates
  - vii. Saving Block Data to Post Meta
- g. Tips
  - i. Enable WP\_SCRIPT\_DEBUG
  - ii. Use filemtime() for the Script Version
  - iii. More Tips
  - iv. Learn JavaScript, Node.js, and React More Deeply

## **14. 12. WordPress Multisite Networks**

- a. Why Multisite?**
- b. Why Not Multisite?**
- c. Multisite Alternatives**
  - i. Multiple Authors or Categories on the Same WordPress Site**
  - ii. Custom Post Types**
  - iii. Totally Separate Sites**
  - iv. Use a WordPress Maintenance Service**
  - v. Multitenancy**
- d. Setting Up a Multisite Network**
- e. Managing a Multisite Network**
  - i. Dashboard**
  - ii. Sites**
  - iii. Users**
  - iv. Themes**
  - v. Plugins**
  - vi. Settings**
  - vii. Updates**
- f. Multisite Database Structure**
  - i. Networkwide Tables**
  - ii. Individual Site Tables**
  - iii. Shared Site Tables**

g. Domain Mapping

h. Random Useful Multisite Plugins

i. Gravity Forms User Registration Add-On

ii. Member Network Sites Add-On for Paid Memberships Pro

iii. More Privacy Options

iv. Multisite Global Media

v. Multisite Plugin Manager

vi. Multisite Global Search

vii. Multisite Robots.txt Manager

viii. NS Cloner: Site Copier

ix. WP Multi Network

i. Basic Multisite Functionality

i. \$blog\_id

ii. is\_multisite()

iii. get\_current\_blog\_id()

iv. switch\_to\_blog( \$new\_blog )

v. restore\_current\_blog()

vi. get\_blog\_details( \$fields = null,  
\$get\_all = true )

vii. update\_blog\_details( \$blog\_id,  
\$details = array() )

viii. get\_blog\_status( \$id, \$pref )

- ix. update\_blog\_status( \$blog\_id, \$pref, \$value )
- x. get\_blog\_option( \$id, \$option, \$default = false )
- xi. update\_blog\_option( \$id, \$option, \$value )
- xii. delete\_blog\_option( \$id, \$option )
- xiii. get\_blog\_post( \$blog\_id, \$post\_id )
- xiv. add\_user\_to\_blog( \$blog\_id, \$user\_id, \$role )
- xv. wpmu\_delete\_user( \$user\_id )
- xvi. create\_empty\_blog( \$domain, \$path, \$weblog\_title, \$site\_id = 1 )
- xvii. Functions We Didn't Mention

## 15. 13. Localizing WordPress Apps

- a. Do You Even Need to Localize Your App?
- b. How Localization Is Done in WordPress
- c. Defining Your Locale in WordPress
- d. Text Domains
  - i. Setting the Text Domain
- e. Prepping Your Strings with Translation Functions
  - i. \_\_( \$text, \$domain = "default" )
  - ii. \_e( \$text, \$domain = "default" )

- iii. `_x( $text, $context, $domain = "default" )`
- iv. `_ex( $title, $context, $domain = "default" )`
- v. Escaping and Translating at the Same Time

#### f. Creating and Loading Translation Files

- i. Our File Structure for Localization
- ii. Generating a .pot File
- iii. Creating a .po File
- iv. Creating a .mo File

#### g. GlotPress

- i. Using GlotPress for Your WordPress.org Plugins and Themes
- ii. Creating Your Own GlotPress Server

### 16. 14. WordPress Optimization and Scaling

- a. Terms
- b. Origin Versus Edge
- c. Testing
  - i. What to Test
  - ii. Chrome Debug Bar
  - iii. The WordPress Site Health Tool
  - iv. Apache Bench
- v. Siege

d. W3 Total Cache

i. Page Cache Settings

ii. Minify

iii. Database Caching

iv. Object Cache

v. CDNs

vi. GZIP Compression

e. Hosting

i. WordPress-Specific Hosts

ii. Rolling Your Own Server

f. Selective Caching

i. The Transient API

ii. Multisite Transients

g. Using JavaScript to Increase Performance

h. Custom Tables

i. Bypassing WordPress

17. 15. Ecommerce

a. Choosing a Plugin

i.  WooCommerce

ii.  Paid Memberships Pro

iii.  Easy Digital Downloads

b. Payment Gateways

- c. Merchant Accounts
- d. Setting Up SaaS with Paid Memberships Pro
- e. The SaaS Model
  - i. Step 0: Establishing How You Want to Charge for Your App
  - ii. Step 1: Installing and Activating Paid Memberships Pro
  - iii. Step 2: Setting Up the Level
  - iv. Step 3: Setting Up Pages
  - v. Step 4: Choosing Payment Settings
  - vi. Step 5: Choosing Email Settings
  - vii. Step 6: Choosing Advanced Settings
  - viii. Step 7: Locking Down Pages
  - ix. Step 8: Customizing Paid Memberships Pro

## 18. 16. Mobile Apps Powered by WordPress

- a. Mobile App Use Cases
- b. Native and Hybrid Mobile Apps
  - i. What Is a Native Mobile App?
  - ii. What Is a Hybrid Mobile App?
  - iii. Why Hybrid over Native?
  - iv. Cordova
  - v. Ionic Framework

vi. App Wrapper

vii. AppPresser

## 19. 17. PHP Libraries, Web Service Integrations, and Platform Migrations

### a. PHP Libraries

i. Image Generation and Manipulation

ii. PDF Generation

iii. Geolocation and Geotargeting

iv. File Compression and Archiving

v. Developer Tools

### b. External APIs and Web Services

i. Elasticsearch

ii. ElasticPress by 1oup

iii. Google Vision

iv. Google Maps

v. Google Translate

vi. Twilio

vii. Other Popular APIs

### c. Migrations

i. Host Migrations

ii. Platform Migrations

iii. Create a Data Mapping Guide

## 20. 18. The Future

- a. Where We've Been
- b. The REST API
  - i. WordPress Plugins Will Focus More on APIs
  - ii. Headless WordPress
  - iii. GraphQL
- c. Gutenberg
  - i. The Administrator Interface Will Move to React/Gutenberg
  - ii. Gutenberg Will Power a Frontend Editing Experience for WordPress
  - iii. Block Templates Will Replace Themes
  - iv. Blocks Will Replace Plugins
- d. WordPress Market Share Will Increase and Decrease
- e. WordPress Will Become a More Popular Platform for Mobile Development
- f. WordPress Will Continue to Be Useful for Developing Apps of All Kinds

## 21. Index

## Praise for *Building Web Apps with WordPress*, Second Edition

*“WordPress is more than just software, it’s a movement that is becoming the de facto operating system of the web. More than just a blog or a CMS, when you learn how to use WordPress as an application platform you’ll be at the forefront of the third wave of its growth.”*

—Matt Mullenweg, Cofounder of WordPress

*“Brian and Jason have grown side by side with WordPress for years, and successfully demonstrate how, for the right kind of app, developers can leverage that engine to build more secure, more performant applications in half the time.”*

—Jake Goldman, President and Founder of 10up

*“Building Web Apps with WordPress is a great resource for developers looking to learn or currently working with the PHP-oriented approach to creating WordPress web apps.”*

—Dave Mackey, Web/Software Developer, Liquid Church

# **Building Web Apps with WordPress**

WordPress as an Application Framework

SECOND EDITION

**Brian Messenlehner and Jason Coleman**

# **Building Web Apps with WordPress**

by Brian Messenlehner and Jason Coleman

Copyright © 2020 Brian Messenlehner and Jason Coleman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Acquisitions Editor: Jennifer Pollock

Development Editor: Alicia Young

Production Editor: Deborah Baker

Copyeditor: Octal Publishing, LLC

Proofreader: Rachel Monaghan

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

April 2014: First Edition

December 2019: Second Edition

## **Revision History for the Second Edition**

- 2019-12-11: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491990087> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Web Apps with WordPress*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-99008-7

[LSI]



## Foreword

For the past several years, I've spent 15 to 24 weekends each year visiting software developers around the country. After something like 25,000 conversations, I've become convinced of three things:

### *Web applications are everywhere*

It's difficult to imagine life without web applications in it. Today, we do our banking online. We book hotels and vacations online using web applications; we do our taxes online in what certainly feels like an application; we check in on our phone to Supercuts before we arrive for our appointment using what feels like a simple form, but it, too, is an application. Web applications are everywhere.

### *Web developers feel completely unable to build them*

If you talk to nondevelopers, they'll tell you that their developer friend is a genius who can do anything with a computer. But if you spend time talking to developers directly, you'll discover that they see the world very differently.

They look at the web applications that they consider "real" and the companies behind them, and they have convinced themselves that the people that work "over there" know things about building web applications that no one else knows.

They're convinced that they can build only small applications—that they don't know what they need to know to do the stuff that will result in wildly popular web applications.

### *What's missing is a little bit of information and a lot more confidence*

When you sit with developers to talk about web applications, you quickly find out that the missing element is confidence more than

anything else. Writing the code isn't hard once you know what you need to do. But that doesn't mean that developers don't need a bit more insight into the process.

These conversations have taken place across the country at local conferences called WordCamps. These are community events that typically have 200 to 500 participants—equally split between developers and nondevelopers. What they have in common is the use or the intention to use the web publishing product called WordPress.

WordPress, if you've never heard of it, powers a third of the internet sites on the planet. It's primarily used as a content publishing platform. But the truth is that you can use WordPress to kickstart your web application.

That's what you'll find in the pages of this book.

What I love about this book is that it doesn't skimp in the early chapters. It helps readers slowly navigate the early questions. It gives you confidence. And I know how valuable you'll find that. It creates a roadmap for how to get from where you are to where you want to be.

But what I also love about this book is that it pushes you deeper into the details of every line of code you need to write. Some of it will be new to you. Some of it won't. But what you won't find is high-level writing that doesn't explain itself and leaves you wondering how to apply your newfound knowledge.

Instead, Brian and Jason introduce you to every part of WordPress that you need to know to use it as a web application framework. There are coding schools today that need to teach from this material. Thankfully, you don't need to pay for those schools, which cost far more than this book.

I can't recommend this book with any more excitement than in this foreword.

Until now you may have thought that there were two kinds of developers. The first is the kind who builds hardcore applications that you never could. The other is the kind like you, who could build small or simple applications.

When you've finished reading this book, there is one thing that I know for sure. When you're done, when you've read the words and played with the code, when you've put it into action—you will know that there's only one kind of developer: the kind that can build anything they can imagine, and you're that kind of developer.

The fact that you're reading these words now makes that already true. Now it's time for you to discover that reality.

***Chris Lema***  
***VP of Products & Innovation at Liquid Web***

# Preface

---

As we write this, WordPress powers 32% of all sites on the internet, and that number is growing. Many developers want to do more with their WordPress sites but feel that they need to jump ship to a more traditional application framework like Ruby on Rails, Symfony, Yii, or Laravel to build “real” web apps. This sentiment is wrong, and we’re here to fix it.

Despite starting out as blogging software and currently existing primarily as a content management system, WordPress has grown into a flexible and capable platform for building web apps. This book will show you how to use WordPress as an application framework to build *any* web app, large or small.

## Who This Book Is For

This book will be most useful for WordPress developers looking to work on heavier applications, and PHP developers with some WordPress experience looking for a PHP-based application framework.

Commercial plugin and theme developers, or anyone working on large distributed WordPress projects, will also find the concepts and techniques of this book useful.

If you are a PHP or language-agnostic developer using another framework and jealous of the large library of WordPress plugins and themes, you may be surprised to learn how well WordPress can work as a general application framework. Reading and applying the lessons in this book could change your work life for the better.

We assume that readers have an intermediate understanding of general PHP programming. You should also have a basic understanding of HTML and CSS, and familiarity with MySQL and SQL queries. Basic understanding of JavaScript and jQuery programming will help with Chapter 9 and any related examples.

## Who This Book Is Not For

This book is not for people who want to learn how to use WordPress as an end user. There will be brief introductions to standard WordPress functionality, but we assume that you have already experienced WordPress from a user's perspective.

This book is not meant for nonprogrammers. While it is possible to build very functional web applications by simply combining and configuring the many plugins available for WordPress, this book is written for developers building their own plugins and themes to power new web apps.

This book will not teach you how to program but rather how to program “the WordPress way.”

## What You'll Learn

Our hope with this book is that you will learn the programming and organizational techniques and best practices for developing complex applications using WordPress.

### *Chapter 1, Building Web Apps with WordPress*

Defines what we mean by “web app” and also covers why or why not to use WordPress for building web apps and how to compare WordPress to other application frameworks. We also introduce SchoolPress, the WordPress app that we use as an example throughout the book.

### *Chapter 2, WordPress Basics*

Covers the basics of WordPress. We go over the various directories of the core WordPress installation and what goes where. We also explain each database table created by WordPress, what data each holds, and which WordPress functions map to those tables. Even if you are an experienced WordPress developer, you can learn something from this chapter, and we encourage you to read it.

### *Chapter 3, Using WordPress Plugins*

Is all about plugins. What are they? How do you make your own plugins? How should you structure your app's main plugin? When should you leverage third-party plugins or roll your own?

### *Chapter 4, Themes*

Is all about themes. How do themes work? How do themes map to views in a typical model-view-controller (MVC) framework? What code should go into your theme, and what code should go into plugins? We also cover using theme frameworks and UI frameworks and the basics of responsive design.

### *Chapter 5, Custom Post Types, Post Metadata, and Taxonomies*

Covers custom post types and taxonomies. We go over the default post types built into WordPress, why you might need to build your own, and then how to go about doing that. We also cover post meta and taxonomies, what each is appropriate for, and how to build custom taxonomies and map them to your post types. Finally, we show how to build wrapper classes for your post types to organize your code utilizing object-oriented programming (OOP).

### *Chapter 6, Users, Roles, and Capabilities*

Covers users, roles, and capabilities. We show how to add, update, and delete users programmatically and how to work with user meta, roles, and capabilities. We also show how to extend the

`WP_User` class for your user archetypes like “customers” and “teachers” to better organize your code using OOP techniques.

### *Chapter 7, Working with WordPress APIs, Objects, and Helper Functions*

Covers a few of the more useful WordPress APIs and helper functions that didn’t fit into the rest of the book but are still important for developers building web apps with WordPress.

### *Chapter 8, Secure WordPress*

Is all about securing your WordPress apps, plugins, and themes.

### *Chapter 9, JavaScript Frameworks and Workflow*

Covers using JavaScript and Ajax in your WordPress application. We go over the correct way to enqueue JavaScript into WordPress and how to build asynchronous behaviors in your app.

### *Chapter 10, WordPress REST API*

Covers the REST API for WordPress and how to use it to integrate WordPress with outside apps.

### *Chapter 11, Project Gutenberg, Blocks, and Custom Block Types*

Covers the block editor and how to make your own blocks.

### *Chapter 12, WordPress Multisite Networks*

Covers WordPress multisite networks, including how to set them up and things to keep in mind when developing for multisite.

### *Chapter 13, Localizing WordPress Apps*

Covers localizing your WordPress plugins and themes, including how to prepare your code for translation and how to create and use translation files.

### *Chapter 14, WordPress Optimization and Scaling*

Covers how to optimize and scale WordPress for high-volume web apps. We go over how to test the performance of your WordPress app and the most popular techniques for speeding up and scaling sites running WordPress.

### *Chapter 15, Ecommerce*

Covers ecommerce. We go over the various types of ecommerce plugins available and how to choose between them. We then go into detail on how you can use WordPress to handle payments and account management for software as a service (SaaS) web apps.

### *Chapter 16, Mobile Apps Powered by WordPress*

Covers how to use WordPress to power native apps on mobile devices by creating app wrappers for iOS and Android.

### *Chapter 17, PHP Libraries, Web Service Integrations, and Platform Migrations*

Covers some third-party PHP libraries, services, and APIs that are often used in web apps as well as how to integrate them with WordPress, including full migrations.

### *Chapter 18, The Future*

Discusses the future of WordPress, what kinds of apps we expect to see running on WordPress, what kinds of updates we see coming for WordPress, and which tools and frameworks to keep an eye on for the future.

## **About the Code**

You can find all examples in this book at <https://github.com/bwawwp>.

Please note that these code examples were written to most clearly convey the concepts we cover in the book. To improve readability, we often ignored best practices for coding standards, security, and

localization (which we cover in Chapters 8 and 13) or ignored certain edge cases. You will want to keep this in mind before using any examples in production code.

You can find the sample app SchoolPress at <https://schoolpress.me>, with any open sourced code for that site available at this book's GitHub site.

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

### *Constant width*

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, datatypes, environment variables, statements, and keywords.

### ***Constant width bold***

Shows commands or other text that should be typed literally by the user.

### *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.

## NOTE

This element signifies a tip, suggestion, or general note.

## WARNING

This element indicates a warning or caution.

# Using Code Examples

As mentioned in “About the Code”, you can find all code examples in this book at <https://github.com/bwawwp>.

If you have a technical question or a problem using the code examples, please send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Building Web Apps with WordPress*, Second Edition, by Brian Messenlehner and Jason Coleman (O’Reilly). Copyright 2020 Brian Messenlehner and Jason Coleman, 978-1-491-99008-7.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## O'Reilly Online Learning

### NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata and any additional information. You can access this page at  
[\*https://oreil.ly/building-apps-wp2\*](https://oreil.ly/building-apps-wp2).

Email [\*bookquestions@oreilly.com\*](mailto:bookquestions@oreilly.com) to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at [\*http://www.oreilly.com\*](http://www.oreilly.com).

Find us on Facebook: [\*http://facebook.com/oreilly\*](http://facebook.com/oreilly)

Follow us on Twitter: [\*http://twitter.com/oreillymedia\*](http://twitter.com/oreillymedia)

Watch us on YouTube: [\*http://www.youtube.com/oreillymedia\*](http://www.youtube.com/oreillymedia)

## Acknowledgments

*From Brian:* Thanks to Jason Coleman and Matt Mullenweg; I could not have written this book without them! A very special thanks to Alicia Young for staying on top of things at O'Reilly Media. A special thanks to our technical reviewers for making sure everything in the book is legit. Thanks to Scott Bolinger from AppPresser.com and Jaffe Worley from AlphaWeb.com for putting up with me. A shout out to family and friends who have always been there for me and never stopped believing in me. Most of all, I'm thankful for my children, Dalya; Brian, Jr.; Nina; Cam; and Aksel Messenlehner—they give me a purpose, and without them I probably wouldn't even know what WordPress was.

*From Jason:* Thanks to my coauthor Brian for asking me to write this book with him. Thanks to our original editors Meghan and Allyson for keeping us on track and helping us to stay true to our original vision. Thanks to Alicia Young for editing the second edition of this book and sitting through our explanations of “WP Drama.” Thanks to all of the great technical editors we’ve had across both editions of the book: Sam Hotchkiss, Peter MacIntyre, Pippin Williamson, John James Jacoby, and Andrew Lima. Thanks to Frederick Townes for his feedback and contributions to our chapter on optimization and scaling. Thanks to Chris Lema for the wonderful forward to this book, his feedback on the book, and advice in general. Thanks to everyone in the WordPress community who answered all of my random tweets and may or may not have known they were helping me to write this book. Thanks to my wife, Kim, for supporting me as always during yet another adventure in our life. Thanks to my

daughter, Marin, for missing me when I was away to write, and my son, Isaac, for constantly asking me if I had “finished the book yet.” Last but not least, thanks to my family, who have always supported my writing: Mom, Dad, Jeremy, and Nana Men are all excited to be the first nonprogrammers to read *Building Web Apps with WordPress*.

# Chapter 1. Building Web Apps with WordPress

---

This book will help you build anything with WordPress: websites, themes, plugins, web services, and web apps. We chose to focus on web apps because you can view them as super websites that make use of all of the techniques we will cover.

There are many people who believe that WordPress isn't powerful enough or meant for building web apps; we'll get into that more later on. We've been building web apps with WordPress for many years and know that it's absolutely possible for you to use it to build scalable applications.

In this chapter, we start by defining what web app's are and then cover why WordPress is a great framework for building them. We also describe some situations in which using WordPress *wouldn't* be the best way to build your web app.

## What Is a Website?

You know what a website is: a set of one or more web pages, containing information, accessed via a web browser.

## What Is an App?

We like the [Wikipedia definition](#): “Application software (app for short) is software designed to perform a group of coordinated functions, tasks, or activities for the benefit of the user.”

## What Is a Web App?

A web app is just *an app run through a web browser*.

Note that with some web apps, the browser technology is hidden—for example, when you’re integrating your web app into a native Android or iOS app, running a website as an application in Google Chrome, or running an app using Adobe AIR. However, inside these applications, there is still a system parsing HTML, CSS, and JavaScript.

You can also think of a web app as *a website, plus more application-like stuff*. There is no exact dividing line where a website becomes a web app. It’s one of those cases where you just know it when you see it.

What we *can* do is explain some of the features of a web app, give you some examples, and then try to come up with a shorthand definition so that you know generally what we are talking about as we use the term throughout the book.

## NOTE

You will see references to SchoolPress while reading this book. SchoolPress is a web application we are building to help schools and educators manage their students and curricula. All of the code examples are geared toward functionality that may exist in SchoolPress. We talk more about the overall concept of SchoolPress later in this chapter.

## Features of a Web App

The following are some features typically associated with web apps and applications in general. The more of these features there are in a website, the more appropriate it is to upgrade its label to a web app.<sup>1</sup>

### *Interactive elements*

A typical website experience involves navigating through page loads, scrolling, and clicking hyperlinks. Web apps can have links and scrolling, too, but they tend to use other methods of navigating through the app.

Websites with forms offer transactional experiences. An example would be a contact form on a website or an application form on the careers page of a company's career website. Forms allow users to interact with a site using something more than a click.

Web apps will have even more interactive user interface (UI) elements. Examples include toolbars, drag-and-drop elements, rich text editors, and sliders.

### *Tasks rather than content*

Remember, web apps are “designed to help the user to perform specific tasks.” Google Maps users get driving directions. Gmail

users write emails. Trello users manage lists. SchoolPress users comment on class discussions.

Some apps are still content focused. A typical session with a Facebook or Twitter app involves about 90% reading. However, the apps themselves present a way of browsing content different from the typical web browsing experience.

### *Logins*

Logins and accounts allow a web app to save information about its users. This information is used to facilitate the main tasks of the app and enable a persistent experience. When logged in, SchoolPress users can see which discussions are unread. They also have a username that identifies their activity within the app.

Web apps can also have tiers of users. SchoolPress will have administrators controlling the inner workings of the app, teachers setting up classes, and students participating in class discussions.

### *Device capabilities*

Web apps running on your phone can access your camera, your address book, internal storage, and GPS location information.

Web apps running on the desktop may access a webcam or a local hard drive. The same web app might respond differently depending on the device accessing it. Web apps will adjust to different screen sizes, resolutions, and capabilities.

### *Work offline*

Whenever possible, it's a good idea to make your web apps work offline. Sure, the interactivity of the internet is what defines that “web” part of web app, but a site that still works when you drive through a tunnel will feel more like an app.

With Gmail, you can draft emails offline. Evernote allows you to draft notes offline and then synchronize them to the internet when connectivity is restored.

## *Mashups*

Web apps can tie one or more web apps together. A web app can utilize various web services and APIs to push and pull data. You could have a web app that pulls location-based information like longitude and latitude from Twitter and Foursquare and posts it to a Google Map.

## **Mobile Apps**

Since the first edition of this book was published back in 2012, web apps—and mobile apps in particular—have taken off. On most websites, mobile devices have now overtaken desktop computers as the largest source of traffic (Source: Perficient, Inc.).

In 2012, the quintessential web app looked something like Basecamp, project management software accessed through your desktop web browser. In 2019, the quintessential web app looks like Twitter, a communication app accessed through your iOS or Android phone.

Because in most cases, the majority of your users will be accessing your websites and apps on a mobile device, we support a “mobile first” mindset when designing and developing web apps. We cover how to get your WordPress apps running natively on mobile in [Chapter 16](#). We cover the basics of responsive design and having your websites show up properly for any screen size in [Chapter 4](#).

## **Progressive Web Apps**

*Progressive web apps* (PWAs) are websites that take advantage of modern browser features to behave as native apps in Android, iOS, or

on the desktop. Specifically, websites that use *service workers* to function while offline have a *web app manifest* file to define the app to the operating system (OS), and meet a few other requirements so they can be launched as apps directly from the browser.

PWAs have been championed by the Google Chrome team, but are now supported on iOS and most modern web browsers. A feature plugin for PWA support is in development to support the primary features of PWAs in WordPress core. You can use this plugin turn your WordPress site into a PWA, and that is a good idea, but in reality coding a PWA is more of a mindset than a simple conversion. Similar to the “features of a web app” we just described, the main PWA site at Google has a checklist of features expected for most PWAs, including these baseline features:

- Site is served over HTTPS.
- Pages are responsive on tablets and mobile devices.
- All app URLs load while offline.
- Metadata is provided for Add to Home screen.
- First load is fast even on 3G.
- Site works cross-browser.
- Page transitions don’t feel like they block on the network.
- Each page has a URL.

In addition to the baseline features, there is a checklist of items for “exemplary” PWAs that covers user experience (UX) and performance. Google’s Lighthouse tool provides automated tests and

reports for meeting the PWA criteria. Even fully native apps or apps built for the browser can benefit from some of the suggestions in the PWA checklists and Lighthouse reports.

## Why Use WordPress?

No single programming language or software tool will be right for every job. We'll explore why you may *not* want to use WordPress in a bit, but for now, let's go over some situations in which using WordPress to build your web app *would* be a good choice.

### You Are Already Using WordPress

If you already use WordPress for your main site, you might be just a quick plugin away from adding the functionality you need. WordPress has great plugins for ecommerce (WooCommerce), forums (bbPress), membership sites (Paid Memberships Pro), social networking functionality (BuddyPress), and gamification (BadgeOS).

Building your app into your existing WordPress site will save you time and make things easier on your users. So, if your application is fairly straightforward, you can create a custom plugin on your WordPress site to program the functionality of your web app.

If you are happy with WordPress for your existing site, don't be confused if people say that you need to upgrade to something else to add certain functionality to your site. It's probably not true. You don't need to throw out all the work you've already done on WordPress, and what follows are great reasons to stick with WordPress.

## **Content Management Is Easy with WordPress**

Developed first as a blogging platform, WordPress has evolved through the years, and with the introduction of custom post types (CPTs) in version 3.0, into a fully functional content management system (CMS). Any page or post can be edited by administrators via the dashboard, which can be accessed through your web browser. You will learn about working with CPTs in [Chapter 5](#).

WordPress makes adding and editing content easy via a WYSIWYG (What You See Is What You Get) editor, so you don't have to use web designers every time you want to make a simple change to your site. You can also create custom menus and navigation elements for your site without touching any code.

If your web app focuses on bits of content (e.g., our SchoolPress app is focused on assignments and discussions), the Custom Post Types API for WordPress (covered in [Chapter 5](#)) makes it easy to quickly set up and manage this custom content.

Even apps that are more task oriented will typically have a few pages for information, documentation, and sales. Using WordPress for your app will give you one place to manage your app and all of your content.

## **User Management Is Easy and Secure with WordPress**

WordPress has everything you need for adding both administrative users and end users to your site.

In addition to controlling access to content, the Roles and Capabilities system in WordPress is extensible and allows you to control what *actions* are available for certain groups of users. For example, by default, users with the contributor role can *add* new posts, but can't *publish* them. Similarly, you can create new roles and capabilities to manage who has access to your custom functionality.

You can use plugins like Paid Memberships Pro to extend the built-in user management to allow you to designate members of different levels and control what content users have access to. For example, you can create a level to give paying members access to premium content on your WordPress site.

## Plugins

There are more than 55,000 free plugins in the [WordPress repository](#). There are many more plugins, both free and premium, on various sites around the internet. When you have an idea for an extension to your website, there is a good chance that there's a plugin for that, which will save you time and money.

There are a handful of indispensable plugins that we end up using on almost every site and web application we build.

For most websites you create, you'll want to cache output for faster browsing, use tools like Google Analytics for visitor tracking, create sitemaps, and tweak page settings for search engine optimization (SEO), along with a number of other common tasks.

There are many well-supported plugins for all of these functions. We suggest our favorites throughout this book; you can find a list of them on this book's website.

## Flexibility Is Important

WordPress is a full-blown framework capable of many things. Additionally, WordPress is built on PHP, JavaScript, and MySQL technology, so anything you can build in PHP/MySQL (which is pretty much anything) can be bolted into your WordPress application easily enough.

WordPress and PHP/MySQL in general aren't perfect for every task, but they are well suited for a wide range of tasks. Having one platform that will grow with your business can allow you to execute and pivot faster. For example, here is a typical progression for the website of a Lean startup running on WordPress:

1. Announce your startup with a one-page website.
2. Add a form to gather email addresses.
3. Add a blog.
4. Focus on SEO and optimize all content.
5. Push blog posts to Twitter and Facebook.
6. Add forums.
7. Use the Paid Memberships Pro plugin to allow members to pay for access.
8. Add custom forms, tools, and application behaviors for paying members.

9. Update the UI using JavaScript techniques and frameworks.
10. Tweak the site and server to scale.
11. Localize the site/app for different countries and languages.
12. Add Progressive Web App support.
13. Launch iOS and Android wrappers for the app.

The neat thing about moving through this path is that at every step along the way, you have *the same database of users* and are using *the same development platform*.

## Frequent Security Updates

The fact that WordPress is used on millions of sites makes it a target for hackers trying to break through its security. Some of those hackers have been successful in the past; however, the developers behind WordPress are quick to address vulnerabilities and release updates to fix them. It's like having millions of people constantly testing and fixing your software, because that's exactly what is happening.

The underlying architecture of WordPress makes applying these updates a quick and painless process that even novice web users can perform. If you are smart about how you set up WordPress and upgrade to the latest versions when they become available, WordPress is a far more secure platform for your site than anything else available. We discuss security in more detail in [Chapter 8](#).

## Cost

WordPress is free. PHP is free. MySQL is free. Most plugins are free.

Servers and hosting cost money, but depending on how big your web application is and how much traffic you get, it can be relatively inexpensive. If you require custom functionality not found in any existing plugins, you may need to pay a developer to build it. Or if you are a developer yourself, it will cost you some time.

## **Responses to Some Common Criticisms of WordPress**

Some highly vocal critics of WordPress might say that it isn't a good framework for building web apps, or that it isn't a framework at all. With all due respect to those with these opinions, we'd like to go over why we disagree. The following are some common criticisms.

### **WORDPRESS IS JUST FOR BLOGS**

Many people believe that because WordPress was first built to run a blog, it is good only for running blogs.

Statements like this were true a few years ago, but WordPress has since implemented strong CMS functionality, making it useful for other content-focused sites. WordPress is now the most popular CMS in use, with more than 60% market share.<sup>2</sup> Figure 1-1 shows a slide from Matt Mullenweg's "State of WordPress" presentation from WordCamp San Francisco 2013. The upside-down pyramid on the left represents a circa 2006 WordPress, with most of the code devoted to the blog application and a little bit of CMS and platform code holding it up. The pyramid on the right represents the current state of

the WordPress platform, where most of the code is in the platform itself, with a CMS layer on top of that, and the blog application running on top of the CMS layer. WordPress is a much more stable platform than it was just a few years ago.



*Figure 1-1. Diagrams from Matt Mullenweg's 2013 "State of WordPress" presentation. WordPress wasn't always so stable.*

You can use the Custom Post Types API to tweak your WordPress installation to support other content types besides blog posts or pages. We cover this in detail in [Chapter 5](#).

## WORDPRESS IS JUST FOR CONTENT SITES

Similar to the “just for blogs” folks, some will say that WordPress is just for content sites.

First, even if WordPress were applicable for only content-based sites and apps, that would account for a large number of apps. The home screen of your phone probably includes a large number of content-based apps like Netflix, Twitter, Facebook, Reddit, and Evernote. These are very popular apps maintained by giant companies. Now,

we're not suggesting these apps *should* run on WordPress, but we *are* suggesting that you *could* build an app similar to these on using WordPress as an application framework.

Second, as we'll go over in detail in this book, WordPress is a great framework for building more interactive web applications as well. The main feature allowing WordPress to be used as a framework is the Plugin API, which allows you to hook into how WordPress works by default and change things. Not only can you use the thousands of plugins available in the WordPress repository and elsewhere on the internet, you can also use the Plugin API to write your own custom plugins to make WordPress do anything possible in PHP/MySQL.

## WORDPRESS DOESN'T SCALE

Some will point to a default WordPress installation running on low-end hosting and note how the site slows down or crashes under heavy load, and thus conclude that WordPress doesn't scale.

Or maybe when we suggested you could build a site like Facebook using WordPress, you rightly scoffed at the idea.

### NOTE

If you are intending to build an app at Facebook scale, this is not the book for you. Ask your CTO what part of their billion-dollar budget is allocated to your app and which engineers you need to headhunt from Google and Amazon to build your custom solution.

In reality, many high-traffic sites run on WordPress. WordPress.com runs on the same basic software as any WordPress site and is one of the most highly trafficked websites in the world.

As use of your app scales up, you will need to upgrade and swap out individual components to meet that scale. The issues with scaling WordPress are the same issues you have scaling any application: caching pages and data, handling database calls more rapidly, and improving network performance. Large sites like WordPress.com, TechCrunch, and the *New York Times* blogs have scaled on WordPress. Similarly, most of the lessons learned scaling PHP/MySQL applications in general apply to WordPress as well. We cover scaling WordPress apps in detail in [Chapter 14](#).

## WORDPRESS IS INSECURE

Like any open source product, there will be a trade-off with regard to security when using WordPress.

On the one hand, because WordPress is so popular, it will be the target of hackers looking for security exploits. And because the code is open source, these exploits will be easier to discover.

On the other hand, because WordPress is open source, you will hear about it when these exploits become public, and someone else will probably fix the exploit for you.

We feel more secure knowing that there are lots of people out there trying to exploit WordPress and just as many people working to make WordPress secure against those exploits. We don't believe in

“security through obscurity” except as an additional measure. We’d rather have the security holes in our software come out in the open rather than go undetected until the worst possible moment.

[Chapter 8](#) covers security issues in more detail, including a list of best practices to harden your WordPress install and how to code in a secure manner.

## WORDPRESS PLUGINS ARE CRAP

The Plugin API in WordPress and the thousands of plugins that have been developed using it are the secret sauce and, in our opinion, the number-one reason that WordPress has become so popular and is so successful as a website platform.

Some people will say, “Sure, there are thousands of plugins, but they are all crap.” OK, some of the plugins out there *are* crap.

But there are a lot of plugins that are most definitely not crap—among them, AppPresser, developed by coauthor Brian Messenlehner. If you like using WordPress to manage your written content or ecommerce store, the AppPresser plugin and platform is the fastest way to get that content or store into a mobile app.

Paid Memberships Pro, developed by coauthor Jason Coleman, is also not crap. Using Paid Memberships Pro to handle your member billing and management will allow you to focus your development efforts on your app’s core competency instead of how to integrate your site with a payment gateway.

A lot of plugins do something very simple (e.g., hiding the admin bar from nonadmins), work exactly as advertised, and don't really have room for being crap.

Themes and plugins found in the WordPress.org repository are heavily vetted by volunteers for security and code quality. The [WordPress.org Theme Review](#) process is notoriously stricter and more comprehensive than processes at other marketplaces. The [Tide project](#) is working to add automated tests to the plugin and theme repositories that will result in higher-quality plugins and updates while also detecting compatibility and security issues faster.

Even the crappy plugins can be fixed, rewritten, or borrowed from to work better. You may find it easier sometimes to rewrite a bad plugin instead of fixing it. However, you are still further ahead than you would be if you had to write everything yourself from scratch.

No one is forcing you to use WordPress plugins without vetting them yourself. If you are building a serious web app, you're going to check out the plugin code yourself, fix it up to meet your standards, and move on with development.

## When Not to Use WordPress

WordPress isn't the solution for every application. This section describes a few cases in which you *wouldn't* want to use WordPress to build your application.

### You Plan to License or Sell Your Site's Technology

WordPress uses the GNU General Public License, version 2 (GPLv2), which has restrictions on how you distribute software that you build with it. Namely, you cannot restrict what people do with your software once you sell or distribute it.

This is a complicated topic, but the basic idea is if you are only selling or giving away *access* to your application, you won't need to worry about the GPLv2. However, if you are selling or distributing the underlying source code of your application, the GPLv2 will apply to the code you distribute.

For example, if we host SchoolPress on our own servers and sell accounts to access the app, that doesn't count as distribution, and the GPLv2 doesn't impact our business at all.

However, if we wanted to allow schools to install the software to run on their own servers, we would need to share the source code with them. This would count as an act of distribution. Our customers would be able to legally give away our source code for free even if we had initially charged them for the software. We must use the GPLv2 license, which doesn't allow us to restrict what users do with the code after they download it.

## **Another Platform Will Get You “There” Faster**

If you have a team of experienced Ruby developers, you should use Ruby to build your web app. If there is a platform, framework, or bundle that includes 80% of the features you need for your web app

and WordPress doesn't have anything similar, you should probably use that other platform.

## Flexibility Is Not Important to You

One of the greatest features of a WordPress site is the ability to quickly change parts of your website to better fit your needs. For example, if Facebook "likes" stop driving traffic, you can uninstall your Facebook Connect plugin and install one for Pinterest.

Generally, updating your theme or swapping plugins on a WordPress site will be faster than developing features from scratch on another platform. However, for cases in which optimization and performance are more important than being able to quickly update the application, programming a native app or programming in straight PHP is going to be the better choice.

If your app is going to do *one simple thing*, you will want to build your app at a lower level. For example, the Paid Memberships Pro license server is basically a single JSON file of add-on information and a small script to check license keys and deliver zipped files. Jason built that license server in straight PHP, with heavy amounts of caching. The license server runs on a \$10/month DigitalOcean Sroplet and serves more than 80,000 sites running Paid Memberships Pro.

Similarly, if you have Facebook's resources, you can afford to build everything by hand and use custom PHP-to-C compilers and native

iOS components to shave a few milliseconds off your website and app load times.

## Your App Needs to Be Highly Real Time

One potential downside of WordPress, which we will get into later, is its reliance on the typical web server architecture. In the typical WordPress setup, a user visits a URL, which communicates with a web server (like Apache) over HTTP, kicks off a PHP script to generate the page, and then returns the full page to the user.

There are ways to improve the performance of this architecture using caching techniques and/or optimized server setups. You can make WordPress asynchronous by using Ajax calls or accessing the database with alternative clients. However, if your application needs to be real time and fully asynchronous (e.g., a chatroom-like app or a multiplayer game), you have our blessing to think twice about using WordPress.

Many WordPress developers, including Matt Mullenweg, the founder and spiritual leader of WordPress, understand this limitation. More and more of the functionality of WordPress is being moved into JavaScript, where computation can be pushed off to the browser and frameworks like REACT can be used to create highly interactive experiences. The new Gutenberg editor added in WordPress 5.0 is the best example of this move and is indicative of things to come, but for now you'll face an uphill battle trying to get WordPress to work asynchronously with the same performance as a native app, or

something built entirely in Node.js or other technologies specifically suited to real-time applications.

## WordPress as an Application Framework

Content management systems like WordPress, Drupal, and Joomla are often left out of the framework discussion, but in reality, WordPress (in particular) is really great for what frameworks are supposed to be about: quickly building applications.

Within minutes, you can set up WordPress and have a fully functional app with user signups, session management, content management, and a dashboard to monitor site activity.

The various APIs, common objects, and helper functions covered throughout this book allow you to code complex applications faster without having to worry about lower-level systems integration.

Figure 1-2 shows that righthand triangle from Mullenweg's 2013 "State of WordPress" presentation depicting a stable WordPress platform with a CMS layer built on top and a blogging application built on top of the CMS layer.

The reality is that the majority of the current WordPress codebase supports the underlying application platform. Think of each WordPress release as an application framework bundled with a sample blogging app.

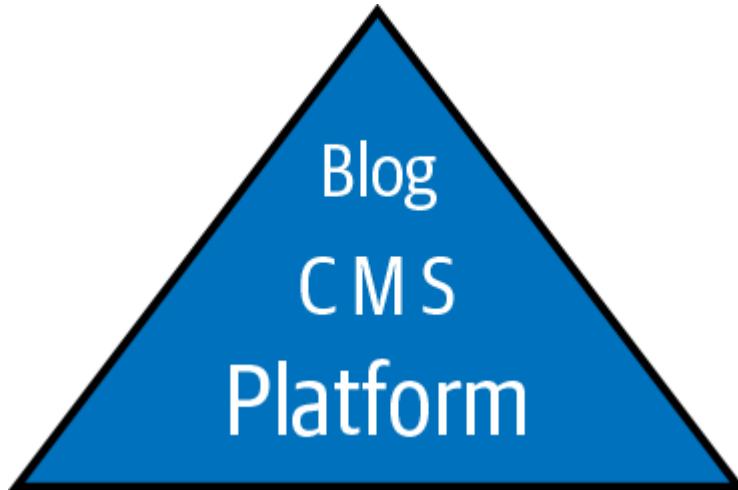


Figure 1-2. The WordPress platform

## WordPress Versus Model-View-Controller Frameworks

Model-view-controller (MVC) is a common design pattern used in many software development frameworks. The main benefits of using an MVC architecture are code reusability and separation of concerns (SoC). WordPress doesn't use an MVC architecture, but does in its own way encourage code reuse and SoC.

Here, we'll briefly explain the MVC architecture, and how it maps to a WordPress development process. If you're familiar with MVC-based frameworks, this section should help you understand how to approach WordPress development in a similar way.

Figure 1-3 describes a typical MVC-based application. The end user uses a *controller*, which manipulates the application state and data via a *model*, which then updates a *view* that is shown to the user. For example, in a blog application, a user might be looking at the recent posts page (a view). The user would click a post title, which would

take the user to a new URL (a controller) that would load the post data (in a model) and display the single post (a different view).

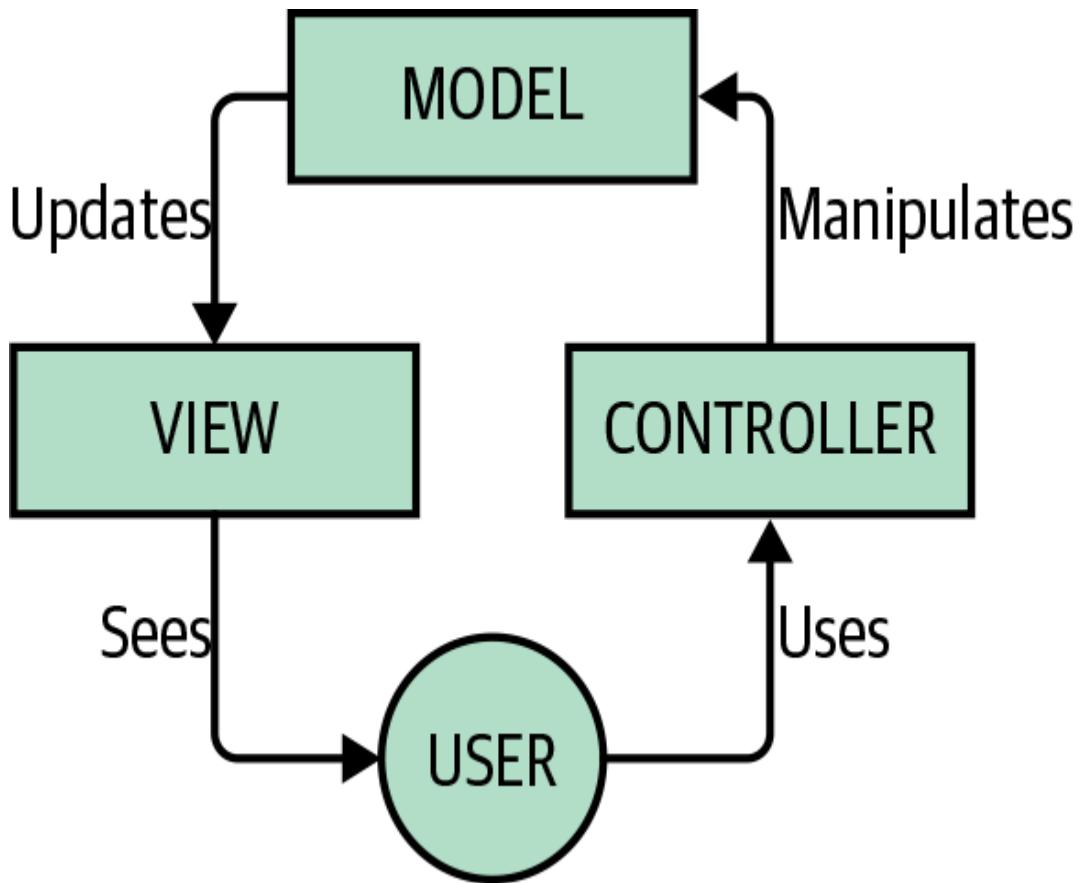


Figure 1-3. How MVC works

The MVC architecture supports code reusability by allowing the models, views, and controllers to interact. For example, both the recent posts view and the single posts view might use the same post model when displaying post data. The same models might be used in the frontend to display posts and in the backend to edit them. The MVC architecture supports SoC by allowing designers to focus their attention on the views while programmers focus their attention on the models.

You could try to use an MVC architecture within WordPress. There are a number of projects to help you do just that; however, we think trying to strap MVC onto WordPress could lead to issues unless the WordPress core were to officially support MVC. Instead, we suggest following the “WordPress Way,” as outlined in this book.

Still, if you are interested, the [WP MVC plugin](#) is in active development and helps you to use an MVC framework to create WordPress plugins. If you don’t want or need to go full MVC, there are a couple of ways to map an MVC process to WordPress.

## MODELS = PLUGINS

In an MVC framework, the code that stores the underlying data structures and business logic is found in the models. This is where the programmers will spend the majority of their time.

In WordPress, plugins are the proper place to store new data structures, complex business logic, and custom post type definitions.

This comparison breaks down in a couple of ways. First, many plugins add view-like functionality and contain design elements—take any plugin that adds a widget to be used in your pages. Second, forms and other design components used in the WordPress dashboard are generally handled in plugins as well.

One way to make the SoC more clear when adding view-like components to your WordPress plugins is to create a *templates* or *pages* folder and put your frontend code into it. Common practice is to allow templates to override the template used by the plugin. For

example, when using WordPress with the Paid Memberships Pro plugin, you can place a folder called *paid-memberships-pro/pages* into your active theme to override the default page templates. (This technique for overriding plugin templates is covered in Chapter 4.)

## VIEWS = THEMES

In an MVC framework, the code to display data to the user is written in the views. This is where designers and frontend developers will spend the majority of their time.

In WordPress, themes are the proper place to store templating code and logic.

Again, the comparison here doesn't map one to one, but "views = themes" is a good starting point.

## CONTROLLERS = TEMPLATE LOADER

In an MVC framework, the code to process user input (in the form of URLs or `$_GET` or `$_POST` data), and decide which models and views to use to handle a request, is stored in the controllers.

Controller code is generally handled by a programmer and often set up once and then forgotten. The meat of the programming in an MVC application happens in the models and views. Even so, controllers are an important part of how an app works.

In WordPress, all page requests (unless they are accessing a cached `.html` file) are processed through the *index.php* file and processed by WordPress according to the *template hierarchy*. The template loader

figures out which file in the template should be used to display the page to the end user. For example, use *search.php* to show search results, *single.php* to show a single post, and so on.

The default behavior can be further customized via the `WP_Rewrite` API (covered in [Chapter 7](#)) and other hooks and filters. You can find information on the [template hierarchy](#) in the WordPress Theme Handbook; we cover the template hierarchy in more depth in [Chapter 4](#).

For a better understanding of how MVC frameworks work, the PHP framework [Yii](#) has a great resource explaining in detail how to best use its MVC architecture.<sup>3</sup>

For more on how to develop web applications using WordPress as a framework, continue reading this book.

## Anatomy of a WordPress App

In this section, we describe the app we built as a companion for this book: SchoolPress. We'll cover the intended functionality of SchoolPress, how it works and who will use it, and—most important for this book—how each piece of the app is built in WordPress.

Don't be alarmed if you don't understand some of the following terminology. In later chapters, we review everything introduced here in more detail. Whenever possible, we point to the chapter that corresponds to the feature discussed.

## NOTE

This book is not meant to be a “how to re-create the SchoolPress app” book or step-by-step walkthrough guide. When it makes sense, we use SchoolPress in our code examples throughout the book so you don’t have to spend time understanding the context of every individual example.

## What Is SchoolPress?

SchoolPress is a web app that makes it easy for teachers to interact with their students outside of the classroom. Teachers can create *classes* and invite their students to them. Each class has a forum for ad hoc discussion and also a more structured system for teachers to post *assignments* and have students turn in their work.

You can find the working app on the [SchoolPress website](#). The application’s source code can be found in the [SchoolPress GitHub repo](#).

## SchoolPress Runs on a WordPress Multisite Network

SchoolPress runs a multisite version of WordPress. The main site hosts free accounts where teachers can sign up and start managing their classes. It also has all of the marketing information for separate school sites on the network, including the page to sign up and check out for a paid membership level.

Schools can create a unique subdomain that will house classes for their teachers. This setup offers finer control and reporting for all

classes across the entire school. Details on using a multisite network with WordPress can be found in [Chapter 12](#).

## The SchoolPress Business Model

SchoolPress uses the Paid Memberships Pro, PMPro Register Helper, and PMPro Network plugins to customize the registration process and accept credit card payments for schools signing up.

Schools can purchase a unique subdomain for their school for an annual fee. No other SchoolPress users pay for access. When school administrators sign up, they can specify a school name and slug for their subdomain *<ourschool>.schoolpress.me*. A new network site is set up for them and they are given access to a streamlined version of the WordPress dashboard for their site.

The school admin then invites teachers into the system. Teachers can also request an invitation to a school that must be approved by the school admin. Teachers can invite students to the classes they create. Students can also request an invitation to a class that must be approved by the teacher.

Teachers can also sign up free of charge to host their classes at *schoolpress.me*. Pages hosted on this subdomain may run ads or other monetization schemes. Details on how to set up ecommerce with WordPress are discussed in [Chapter 15](#).

## Membership Levels and User Roles

Teachers are given a Teacher membership level (through Paid Memberships Pro) and a custom role called “Teacher” that gives them access to create and edit their classes, moderate discussions in their class forums, and create and manage assignments for their classes.

Teachers do not have access to the WordPress dashboard. They create and manage their classes and assignments through frontend forms created for this purpose.

Students are given a “Student” membership level and the default “Subscriber” role in WordPress. Students have access to view and participate only in classes to which they are invited by their teachers. Details on user roles and capabilities are explained in [Chapter 6](#), and [Chapter 15](#) covers using membership levels to control access.

## **Classes Are BuddyPress Groups**

When teachers create “classes,” they are really creating *BuddyPress groups* and inviting their students to the group. Using BuddyPress, we get class forums, private messaging, and a nice way to organize our users.

The class discussion forums are powered by the bbPress plugin. A new forum is generated for each class, and BuddyPress manages access to the forums. Details on leveraging third-party plugins like BuddyPress and bbPress can be found in [Chapter 3](#).

## **Assignments Are a CPT**

Assignments are a CPT that uses a frontend submission form for teachers to post new assignments. Assignments are just like the default blog posts in WordPress, with a title, body content, and attached files. The teacher posting the assignment is the post's author.

### NOTE

WordPress has built-in post types like posts and pages and built-in taxonomies like categories and tags. For SchoolPress, we are creating our own CPTs and taxonomies. Find more on creating custom post types and taxonomies in [Chapter 5](#).

## Submissions Are a (Sub)CPT for Assignments

Students can post comments on an assignment, and they can also choose to post their official submission for the assignment through another form on the frontend.

Submissions, like assignments, are also CPTs. Submissions are linked to assignments by setting the submission's `post_parent` field to the ID of the assignment to which it was submitted. Students can post text content and also add one or more attachments to a submission.

## Semesters Are a Taxonomy on the Class CPT

A custom taxonomy called *Semester* is set up for the group/class CPT. School admins can add new semesters to their sites. For example, a “fall 2019” semester could be created and teachers could assign this semester when creating their classes. Students then can easily browse a list of all fall 2019 classes.

## Departments Are a Taxonomy on the Class CPT

A custom taxonomy called *Department* is also set up for the group/class CPT. This is also available as a drop-down list for teachers when creating their classes, and allows students to browse the list of classes by department.

## SchoolPress Has One Main Custom Plugin

Behind the scenes, the custom bits of the SchoolPress app are controlled from a single custom plugin called SchoolPress. This—the main plugin—includes definitions for the various CPTs, taxonomies, and user roles. It also contains the code to tweak the third-party plugins SchoolPress uses like Paid Memberships Pro and BuddyPress.

The main plugin also contains classes for school admins, teachers, and students that extend the `WP_User` class and classes for classes, assignments, and submissions that wrap the `WP_Post` class. These (PHP) classes allow us to organize our code in an object-oriented way that makes it easier to control how our various customizations work together and to extend our code in the future. These classes are fun to work with and allow for the code shown in Example 1-1.

### Example 1-1. Possible user login events

```
if ($class->isTeacher($current_user))
{
    //this is the teacher, show them teacher stuff
    //...
}
elseif ($class->isStudent($current_user))
{
```

```

    //this is a student in the class, show them student stuff
    //...
}

elseif(is_user_logged_in())
{
    //not logged in, send them to the login form with a
    redirect back here
    wp_redirect(wp_login_url(get_permalink($class->ID)));
    exit;
}

else
{
    //not a member of this class, redirect them to the invite
    page
    wp_redirect($class->invite_url);
    exit;
}

```

Creating custom plugins is covered in [Chapter 3](#), and extending the [WP\\_User](#) class in [Chapter 6](#).

## SchoolPress Uses a Few Other Custom Plugins

Occasionally, a bit of code will be developed for a particular app that would also be useful on other projects. If the code can be contained enough that it can run outside of the context of the current app and main plugin, it can be built into a separate custom plugin.

An example of this would be the [Force First and Last Name as Display Name](#) plugin that was a requirement for this project. It didn't require any of the main plugin code to run and is useful for other WordPress sites outside of the context of the SchoolPress app. We created a separate plugin for this functionality and maintain it in the [WordPress.org](#) repository so others can use it and benefit from it.

## SchoolPress Uses the Memberlite Theme

The main *schoolpress.me* site runs on a customized Memberlite child theme. If a school admin signs up for a premium subdomain, they can choose from a variety of Memberlite child themes; they can also change any of the theme's colors, fonts, and logos to better fit school/class branding. Also, all themes use a responsive design that ensures the site will look good on mobile and tablet displays as well as desktop displays.

The code in the Memberlite theme is very strictly limited to display-related programming. The theme code obviously includes the HTML and CSS for the site's layout, but also contains some simple logic that integrates with the main SchoolPress plugin (like the preceding branching code). However, any piece of code that manipulates the custom post types or user roles or involves a lot of calculation is delegated to the SchoolPress plugin.

Now that we've established what an app is, discussed why you might want to build one with WordPress, presented the "WordPress" way of separating concerns, and described at a high level our SchoolPress example app, let's dig into the core of WordPress, what's included, and how it works.

---

<sup>1</sup> Many of the ideas in this section are influenced by the following blog posts: "[What is a Web Application?](#)" by Dominique Hazaël-Massieux, and "[What is a Web Application?](#)" by Bob Baxley.

<sup>2</sup> W3Tech has regular surveys on the use of different content management systems.

<sup>3</sup> Yii is an MVC-based PHP framework. Other PHP frameworks, like [Laravel](#), are more popular among WordPress developers and the PHP community in general, but the

MVC-related documentation on the Yii website is particularly well written.

# Chapter 2. WordPress Basics

---

WordPress was first developed in 2003 and was created primarily as blogging software. By the release of version 3.5, the image of WordPress had changed from blogging software to a versatile CMS and the word “blog” was actually removed from the description of the software and most places in the source code. Today, WordPress has evolved to become the largest platform on the web and is used on about 30% of all websites on the internet. This is pretty amazing if you think about it. More than a *half a billion* internet websites run on top of WordPress.

WordPress has gained so much popularity over the years for a couple of reasons. The first is that WordPress is open source software and has an entire community of people invested in improving it and continually contributing new code to extend its functionality. WordPress users, developers, and designers are always thinking of new creative ways to use WordPress and creating plugins for these new features, which can be made available to the community.

Another reason that WordPress has been so successful is the fact that it’s an extremely flexible CMS laced with hooks and filters so plugin and theme developers can have almost total control to build all different kinds of websites. Developers are constantly exploring innovative new ways to use the software, including building web and

mobile applications, which is the focus of this book. The use of hooks and filters is covered later in this chapter.

### NOTE

We are going to assume that you already know how to use WordPress and have already installed the latest version. If this is your first time using WordPress, check out [the WordPress home page](#) to familiarize yourself with it.

## WordPress Directory Structure

Let's take a quick top-level look at the folders and files that are included within a typical WordPress install.

### Root Directory

In the root directory, there are a few core WordPress files. Unless you are digging around in the core WordPress code looking for hooks to use or trying to learn how certain functionality is coded, the only core WordPress file you may need to ever access is *wp-config.php*. You should never, ever, ever, ever<sup>1</sup> alter any other core WordPress files. Hacking core files is a bad idea because upgrading to a new version of WordPress will override your changes. The only directory you should need to interact with is *wp-content* because it contains your plugins, themes, and uploaded files.

Any time you find yourself wanting to hack a core WordPress file, think again. There is probably a hook or filter you could use to accomplish the same goal. If there isn't a hook or filter available to

do what you need, add one and request to have it added to the core. The core WordPress developers are very responsive about adding in new hooks and filters.

There is one more file you may need to update in the WordPress root directory, depending on your setup and how you are using WordPress: the *.htaccess* file. It's not a WordPress core file, but an Apache file WordPress uses to handle directory configuration, permalinks, and redirects. This file is not there by default; it's created by WordPress automatically the first time you define your permalink structure. Check out all the *.htaccess* configuration options at your leisure on [WordPress's htaccess Support page](#).

## **/wp-admin**

This directory contains core directories and files for managing the WordPress admin dashboard interface. A key file in this directory is *admin-ajax.php*, which all Ajax requests should be run through. We cover Ajax in [Chapter 9](#).

## **/wp-includes**

This directory contains core directories and files for various WordPress functionality. We highly encourage you to look over the structure and code in this directory to better understand the inner workings of WordPress.

## **/wp-content**

This directory is where WordPress users and developers can make WordPress do whatever they want. It contains subdirectories for the plugins and themes you have installed on your website as well as any media files you upload to your website.

The *wp-content* directory includes several subdirectories, as described next.

## /WP-CONTENT/PLUGINS

Any WordPress plugin you install on your WordPress site will be located in this directory. By default, WordPress comes with the Hello Dolly and Akismet plugins.

Hello Dolly is included as a quick example of how a basic WordPress plugin is set up. The plugin itself just displays a random line from the song “Hello Dolly” in the upper right of the administrator dashboard.

The Akismet plugin helps stop spam comments by checking incoming comments against the database at <https://akismet.com>. This plugin and service greatly reduce the number of spam comments that make it onto the frontend of your website. The Akismet service is free (or name your own price) for personal use.

## /WP-CONTENT/THEMES

Any WordPress themes you install on your WordPress site will be located in this directory. By default, WordPress comes with a few standard themes all named for the year they were released (Twenty Seventeen, Twenty Nineteen, etc.).

## /WP-CONTENT/UPLOADS

Once you start uploading any photos or files to your media library, you will see this directory being populated with those uploaded files. All uploaded media is stored in the *uploads* directory. Some plugins will also create a subdirectory in the *uploads* directory to various files used or managed by the plugin.

## /WP-CONTENT/MU-PLUGINS

In WordPress, you can force the use of any plugin by creating a *mu-plugins* directory inside of the *wp-content* directory. This directory does not exist unless you create it. The “mu” stands for “must use,” and any plugin you put in the *mu-plugins* folder will automatically run without needing to be manually activated on the admin plugins page. In fact, you won’t even see any must-use plugins listed there.

Must-use plugins are especially useful on multisite installs of WordPress so you can use plugins that your individual network site admins won’t be able to deactivate.

It is a good idea to check for the *mu-plugins* folder on any existing site you start working on to see whether it contains any plugins, and if so, to determine what they do. So many times we have been debugging an issue and wondering why something unexpected was happening even though we had disabled all of the active plugins, only to find out that there was an overlooked mu-plugin responsible for the issue.

# WordPress Database Structure

WordPress runs on top of a MySQL database and creates its own tables to store data and content. Following is the database schema created by a default installation of WordPress. We have also included some basic information on built-in WordPress functions for interacting with these tables. If you can grasp the database schema and get comfortable with the list functions in this chapter, you can push and pull any data into and out of WordPress.

### NOTE

The following table names use the default prefix `wp_`. You can change this prefix during the WordPress installation; thus, the exact table names of your WordPress install may vary.

## **`wp_options`**

The `wp_options` table stores any sitewide data for you. This table stores the name, description, and admin email that you entered when running a typical installation. This table will also come prepopulated with a few records that store the various default settings within WordPress. [Table 2-1](#) shows the database structure for the `wp_options` table.

*Table 2-1. Database schema for wp\_options table*

Column	Type	Collation	Nu ll	Defa ult	Extra
option_id	bigint(20)		No	None	AUTO_INCREMENT
option_name	varchar(64)	utf8_general_ci	No		
option_value	longtext	utf8_general_ci	No	None	
autoload	varchar(20)	utf8_general_ci	No	Yes	

WordPress apps and plugins typically store their settings in the wp\_options table using the functions defined in the next section. The settings can be stored in separate rows while using a common prefix for the option names. In most cases, it is more performant to store all of the options in one array and save them into just one row in the wp\_options table.

## Functions Found in /wp-includes/option.php

The following functions can be found in */wp-includes/option.php*.

**ADD\_OPTION( STRING \$OPTION, MIXED \$VALUE = "",  
STRING \$DEPRECATED = "", STRING|BOOL  
\$AUTOLOAD = 'YES' )**

First checks whether an option\_name exists before inserting a new row:

*\$option*

A required string of the option\_name you would like to add.

*\$value*

An optional mixed variable of the option\_value you would like to add. If the variable passed is an array or object, the value will be serialized before storing in the database.

*\$deprecated*

This parameter, deprecated in version 2.3, is no longer used.<sup>2</sup>

*\$autoload*

An optional Boolean used to distinguish whether to load the option into cache when WordPress starts up. Set to yes or no. The default value is yes. If you are sure you are going to need this option on every page load, you can leave the value as the default yes. If you are only going to need to look up the option on specific pages, it's usually better to set autoload to no.

## **UPDATE\_OPTION( \$OPTION, \$NEWVALUE )**

Updates an existing option but will also add it if it doesn't already exist:

*\$option*

A required string of the option\_name you would like to update/add.

*\$newvalue*

An optional mixed variable of the option\_value you would like to update/add.

## **GET\_OPTION( \$OPTION, \$DEFAULT = FALSE )**

Retrieves the option\_value for a provided option\_name:

*\$option*

A required string of the option\_name you would like to get.

*\$default*

An optional mixed variable you would like to return if the option\_name you provided doesn't exist in the table. By default, this parameter is false.

## **DELETE\_OPTION( \$OPTION )**

Deletes an existing option from the database permanently:

*\$option*

A required string of the option\_name you would like to delete.

### **NOTE**

Most code examples in this book are not fully functional code, but basic theoretical examples of how to use the functions we are talking about. You can follow along with most of the code examples in a custom plugin or your theme's *functions.php* file.

Example 2-1 demonstrates some basic functions for interacting with the wp\_options table.

*Example 2-1. Adding, updating, getting, and deleting records in the wp\_options table*

---

```

<?php
// add option
$twitters = array( '@bwawwp', '@bmess', '@jason_coleman' );
add_option( 'bwawwp_twitter_accounts', $twitters );

// get option
$bwawwp_twitter_accounts = get_option(
'bwawwp_twitter_accounts' );
echo '<pre>';
print_r( $bwawwp_twitter_accounts );
echo '</pre>';

// update option
$twitters = array_merge(
    $twitters,
    array(
        '@alphaweb',
        '@pmproplugin'
    )
);
update_option( 'bwawwp_twitter_accounts', $twitters );

// get option
$bwawwp_twitter_accounts = get_option(
'bwawwp_twitter_accounts' );
echo '<pre>';
print_r( $bwawwp_twitter_accounts );
echo '</pre>';

// delete option
delete_option( 'bwawwp_twitter_accounts' );

/*
The output from the above example should look something like
this:
Array
(
    [0] => @bwawwp
    [1] => @bmess
    [2] => @jason_coleman
)
Array

```

```
(  
    [0] => @bwawwp  
    [1] => @bmess  
    [2] => @jason_coleman  
    [3] => @alphaweb  
    [4] => @pmproplugin  
)  
*/  
?>
```

## wp\_users

When you log in to WordPress with your username and password, you are referencing data stored in this table. All users and their default data are stored in the `wp_users` table. [Table 2-2](#) shows the database structure for the `wp_users` table.

*Table 2-2. Database schema for wp\_users table*

Column	Type	Collation	N u ll	Default	Extra
ID	bigint (20)		N o	None	AUTO_INCREMENT
user_login	varchar (60)	utf8_general_ci	N o		
user_pass	varchar (64)	utf8_general_ci	N o		
user_nicename	varchar (50)	utf8_general_ci	N o		
user_email	varchar (100)	utf8_general_ci	N o		
user_url	varchar (100)	utf8_general_ci	N o		
user_registered	datetime		N o	0000-00-00 00:00:00	
user_activation_key	varchar (60)	utf8_general_ci	N o		
user_status	int(11)		N o	0	
display_name	varchar (250)	utf8_general_ci	N o		

For many WordPress apps, you will use the administrator dashboard GUI to create and manage users. However, if you need to create users

in your code or update metadata about them, the functions defined in the next section will be useful.

## Functions Found in /wp-includes/...

These functions are found in */wp-includes/pluggable.php* and */wp-includes/user.php*.

### **WP\_INSERT\_USER( \$USERDATA )**

Inserts a new user into the database. This function can also be used to update a user if the user ID is passed in with the `$user_data`.

`$userdata` is a required array of field names and values. Accepted fields are as follows:

*ID*

An integer that will be used for updating an existing user.

*user\_pass*

A string that contains the plain-text password for the user.

*user\_login*

A string that contains the user's username for logging in.

*user\_nicename*

A string that contains a URL-friendly name for the user. The default is the user's username.

*user\_url*

A string containing the URL for the user's website.

*user\_email*

A string containing the user's email address.

*display\_name*

A string that will be shown on the site. Defaults to the user's username. It is likely that you will want to change this, for appearance.

*nickname*

The user's nickname. Defaults to the user's username.

*first\_name*

The user's first name.

*last\_name*

The user's last name.

*description*

A string containing content about the user.

*rich\_editing*

A string for whether to enable the rich editor. `false` if not empty.

*user\_registered*

The date the user registered. Format is Y-m-d H:i:s.

*role*

A string used to set the user's role.

**WP\_CREATE\_USER( \$USERNAME, \$PASSWORD, \$EMAIL )**

This function utilizes the prior function `wp_insert_user()` and makes it easier to add a new user based on the required columns:

*\$username*

A required string of the username/login of a new user.

*\$password*

A required string of the password of a new user.

*\$email*

A required string of the email address of a new user.

## **WP\_UPDATE\_USER( \$USERDATA )**

Use this function to update any field in the `wp_users` and `wp_usermeta` (covered next) tables tied to a specific user. Note that if a user's password is updated, all of their cookies will be cleared, logging them out of WordPress:

*\$userdata*

A required array of field names and values. The ID and at least one other field is required. These fields are the same ones accepted in the `wp_insert_post()` function.

## **GET\_USER\_BY( \$FIELD, \$VALUE )**

This function returns the `WP_User` object on success and `false` if it fails. The WordPress `User` class is found in `/wp-includes/capabilities.php` and basically queries the `wp_user` table like so:

```
SELECT * FROM wp_users WHERE $field = $value;
```

The `WP_User` class caches the results to avoid querying the database every time it is used. The class also figures out the roles and capabilities of a specific user, which we will go over in more detail in Chapter 6:

*\$field*

A required string of the field by which you would like to query the user data. This string can only be `id`, `slug`, `email`, or `login`.

*\$value*

A required integer or string of the value for a given `id`, `slug`, `email`, or `login`.

## **GET\_USERDATA( \$USERID )**

This function actually utilizes the previous function, `get_user_by()`, and returns the same `WP_User` object:

*\$userid*

A required integer of the user ID of the user for which you would like to get data.

## **WP\_DELETE\_USER( \$ID, \$REASSIGN = 'NOVALUE' )**

You guessed it: this function delete a user and can also reassign any of their posts or links to another user:

*\$id*

A required integer of the ID of the user you would like to delete.

*\$reassign*

An optional integer of the ID you would like to reassign any post or links from the deleted user to. [Example 2-2](#) demonstrates some of the basic functions for interacting with the `wp_users` table.

*Example 2-2. Working with the wp\_users table*

---

```
<?php
// insert user
$userdata = array(
    'user_login'      => 'brian',
    'user_pass'       => 'KO03gT7@n*',
    'user_nicename'   => 'Brian',
    'user_url'        => 'https://alphaweb.com/',
    'user_email'      => 'brian@alphaweb.com',
    'display_name'    => 'Brian',
    'nickname'        => 'Brian',
    'first_name'      => 'Brian',
    'last_name'        => 'Messenlehner',
    'description'     => 'This is a WordPress Administrator
account.',
    'role'            => 'administrator'
);
wp_insert_user( $userdata );

// create users
wp_create_user( 'jason', 'YR529G%*v@', 'jason@schoolpress.me'
);

// get user by login
$user = get_user_by( 'login', 'brian' );
echo 'email: ' . $user->user_email . ' / ID: ' . $user->ID .
'<br>';
echo 'Hi: ' . $user->first_name . ' ' . $user->last_name .
'<br>';

// get user by email
$user = get_user_by( 'email', 'jason@schoolpress.me' );
echo 'username: ' . $user->user_login . ' / ID: ' . $user->ID
. '<br>';
```

```

// update user-change username fields and change role to admin
$userdata = array(
    'ID'          => $user->ID,
    'first_name'  => 'Jason',
    'last_name'   => 'Coleman',
    'user_url'    => 'http://strangerstudios.com/',
    'role'        => 'administrator'
);
wp_update_user( $userdata );

// get userdata for brian
$user = get_userdata( $user->ID );
echo 'Hi: ' . $user->first_name . ' ' . $user->last_name .
'<br>';

// delete user-delete the original admin and set their posts
// to our new admin
// wp_delete_user( 1, $user->ID );

/*
The output from the above example should look something like
this:
email: brian@schoolpress.me / ID: 2
Hi: Brian Messenlehner
username: jason / ID: 3
Hi: Jason Coleman
*/
?>

```

## wp\_usermeta

Sometimes you may want to store additional data along with a user. WordPress provides an easy way to do this without having to add extra columns to the users table. You can store as much user metadata as you need to in the `wp_usermeta` table. Each record is associated to a user ID in the `wp_user` table by the `user_id` field. Table 2-3 shows the database structure for the `wp_usermeta` table.

*Table 2-3. Database schema for wp\_usermeta table*

Column	Type	Collation	Null	Default	Extra
umeta_id	bigint(20)		No	None	AUTO_INCREMENT
user_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_val	longtext	utf8_general_ci	Yes	NULL	

### **GET\_USER\_META( \$USER\_ID, \$KEY = "", \$SINGLE = FALSE )**

Gets a user's meta value for a specified key:

*\$user\_id*

A required integer of a user ID.

*\$key*

An optional string of the meta key of the value you would like to return. If blank, then all metadata for the given user will be returned.

*\$single*

A Boolean of whether to return a single value or not. The default is `false` and the value will be returned as an array.

There can be more than one meta key for the same user ID with different values. If you set `$single` to `true`, you will get the first key's value; if you set it to `false`, you will get an array of the values of each record with the same key.

### **UPDATE\_USER\_META( \$USER\_ID, \$META\_KEY, \$META\_VALUE, \$PREV\_VALUE = '' )**

This function will update user metadata but will also insert metadata if the passed-in key doesn't already exist:

`$user_id`

A required integer of a user ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to store. If this meta key already exists, it will update the current row's meta value; if not, it will insert a new row.

`$meta_value`

A required mixed value of an integer, string, array, or object. Arrays and objects will automatically be serialized.

`$prev_value`

An optional mixed value of the current metadata value. If a match is found, it will replace the previous/current value with the new value you specified. If left blank, the new meta value will replace the first instance of the matching key. If you have five rows of metadata with the same key and you don't specify which row to update with this value, it will update the first row and remove the other four.

## NOTE

This function relies on the `update_metadata()` function located in `/wp-includes/meta.php`. Check it out!

### **ADD\_USER\_META( \$USER\_ID, \$META\_KEY, \$META\_VALUE, \$UNIQUE = FALSE )**

Yup, this function will insert brand-new user meta into the `wp_usermeta` table. We don't use this function often anymore because we can just use `update_user_meta()` to insert new rows as well as update them. If you want to ensure that a given meta key is used only once per user, you should use this function and set the `$unique` parameter to `true`:

`$user_id`

A required integer of a user ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to store.

`$meta_value`

A required mixed value of an integer, string, array, or object.

`$unique`

An optional Boolean that, when set to `true`, will make sure the meta key can only ever be added once for a given ID.

### **DELETE\_USER\_META( \$USER\_ID, \$META\_KEY, \$META\_VALUE = '' )**

Deletes user metadata for a provided user ID and matching key. You can also specify a matching meta value if you want to delete only that value and not other metadata rows with the same meta key:

`$user_id`

A required integer of a user ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to delete.

`$meta_value`

An optional mixed value of the meta value. If you have more than one record with the same meta key, you can specify which one to delete by matching the meta value. It defaults to nothing, which will delete all meta rows with a matching `user_id` and `meta_key`.

Example 2-3 demonstrates some of the basic functions for interacting with the `wp_username` table.

---

*Example 2-3. Working with the `wp_username` table*

---

```
<?php
// get brian's id
$brian_id = get_user_by( 'login', 'brian' )->ID;

// add user meta - unique is set to true.
add_user_meta( $brian_id, 'bwawp_wife', 'Married to the
game', true );

// get user meta - returning a single value
$brians_wife = get_user_meta( $brian_id, 'bwawp_wife', true );
echo "Brian's wife: " . $brians_wife . "<br>";

// add user meta - 3rd parameter is a unique value
```

```
add_user_meta( $brian_id, 'bwawwp_kid', 'Dalya' );
add_user_meta( $brian_id, 'bwawwp_kid', 'Brian' );
add_user_meta( $brian_id, 'bwawwp_kid', 'Nina' );
add_user_meta( $brian_id, 'bwawwp_kid', 'Cam' );
add_user_meta( $brian_id, 'bwawwp_kid', 'Aksel' );

// update user meta - this will update brian to brian jr.
update_user_meta( $brian_id, 'bwawwp_kid', 'Brian Jr', 'Brian'
);

// get user meta - returning an array
$brians_kids = get_user_meta( $brian_id, 'bwawwp_kid' );
echo "Brian's kids:";
echo '<pre>';
print_r($brians_kids);
echo '</pre>';

// delete brian's user meta
delete_user_meta( $brian_id, 'bwawwp_wife' );
delete_user_meta( $brian_id, 'bwawwp_kid' );

// get jason's id
$jason_id = get_user_by( 'login', 'jason' )->ID;

// update user meta - this will create meta if the key doesn't
exist for the user.
update_user_meta( $jason_id, 'bwawwp_wife', 'Kimberly Ann
Coleman' );

// get user meta-returning an array
$jasons_wife = get_user_meta( $jason_id, 'bwawwp_wife' );
echo "Jason's wife:";
echo '<pre>';
print_r($jasons_wife);
echo '</pre>';

// add user meta - storing as an array
add_user_meta( $jason_id, 'bwawwp_kid', array( 'Isaac',
'Marin' ) );

// get user meta - returning a single value which happens to
be an array.
```

```

$jasons_kids = get_user_meta( $jason_id, 'bwawwp_kid', true );
echo "Jason's kids:";  

echo '<pre>';
print_r($jasons_kids);
echo '</pre>';

// delete jason's user meta
delete_user_meta( $jason_id, 'bwawwp_wife' );
delete_user_meta( $jason_id, 'bwawwp_kid' );

/*
The output from the above example should look something like
this:
Brian's wife: Married to the game
Brian's kids:
Array
(
    [0] => Dalya
    [1] => Brian Jr
    [2] => Nina
    [3] => Cam
    [4] => Aksel
)
Jason's wife:
Array
(
    [0] => Kimberly Ann Coleman
)
Jason's kids:
Array
(
    [0] => Isaac
    [1] => Marin
)
*/
?>

```

## wp\_posts

Ah, the meat of WordPress. The wp\_posts table is where most of your post data is stored. By default, WordPress comes with posts and pages. Both of these are technically posts and are stored in this table.

The `post_type` field is what distinguishes the type of a post—that is, whether it is a post, a page, a menu item, a revision, or any CPT that you may later create (CPTs are covered more in [Chapter 5](#)).

[Table 2-4](#) shows the database structure for the `wp_posts` table.

*Table 2-4. Database schema for wp\_posts table*

Column	Type	Collation	N u ll	Default	Extra
ID	bigint (20)	utf8_general_ci	NO	None	AUTO_INCREMENT
post_author	bigint (20)	utf8_general_ci	NO	0	
post_date	datetime	utf8_general_ci	NO	0000-00-00 00:00:00	
post_date_gmt	datetime	utf8_general_ci	NO	0000-00-00 00:00:00	
post_content	longtext	utf8_general_ci	NO	None	
post_title	text	utf8_general_ci	NO	None	
post_excerpt	text	utf8_general_ci	NO	None	
post_status	varchar (20)	utf8_general_ci	NO	Publish	
comment_status	varchar (20)	utf8_general_ci	NO	Open	
ping_status	varchar (20)	utf8_general_ci	NO	Open	
post_password	varchar (20)	utf8_general_ci	NO		
post_name	varchar (200)	utf8_general_ci	NO		

Column	Type	Collation	N u ll	Default	Extra
to_ping	text	utf8_general_ci	N o	None	
pinged	text	utf8_general_ci	N o	None	
post_modified	datetime		N o	0000-00-00 00:00:00	
post_modified_gmt	datetime		N o	0000-00-00 00:00:00	
post_content_filtered	longtext	utf8_general_ci	N o	None	
post_parent	bigint (20)		N o	0	
guid	varchar (255)	utf8_general_ci	N o		
menu_order	int(11)		N o	0	
post_type	varchar (20)	utf8_general_ci	N o	Post	
post_mime_type	varchar (100)	utf8_general_ci	N o		
comment_count	bigint (20)		N o	0	

## Functions Found in /wp-includes/post.php

The functions that follow are found in */wp-includes/post.php*.

## **WP\_INSERT\_POST( \$POSTARR, \$WP\_ERROR = FALSE )**

This function inserts a new post with provided post data:

*\$postarr*

An array or object of post data. Arrays are expected to be escaped; objects are not.

*\$wp\_error*

An optional Boolean that will allow for a `WP_Error` if returned `false`.

The defaults for the parameter `$postarr` are:

*post\_status*

Default is `draft`.

*post\_type*

Default is `post`.

*post\_author*

Default is current user ID (`$user_ID`). The ID of the user who added the post.

*ping\_status*

Default is the value in the `default_ping_status` option.  
Whether the attachment can accept pings.

*post\_parent*

Default is 0. Set this for the post it belongs to, if any.

*menu\_order*

Default is 0. The order in which the array is displayed.

*to\_ping*

Whether to ping.

*pinged*

Default is empty string.

*post\_password*

Default is empty string. The password to access the attachment.

*guid*

Global unique ID for referencing the attachment.

*post\_content\_filtered*

Post content filtered.

*post\_excerpt*

Post excerpt.

**WP\_UPDATE\_POST( \$POSTARR = ARRAY(),  
\$WP\_ERROR = FALSE )**

This function updates a post with provided post data.

*\$postarr*

A required array or object of post data. Arrays are expected to be escaped; objects are not.

*\$wp\_error*

An optional Boolean that will allow for a `WP_Error` if returned `false`.

## **GET\_POST( \$POST = NULL, \$OUTPUT = OBJECT, \$FILTER = 'RAW' )**

This function gets post data from a provided post ID or a post object:

*\$post*

An optional integer or object of the post ID or post object you want to retrieve. The default is the current post you are on inside of the post loop, which is covered later in this chapter.

*\$output*

An optional string of the output format. The default value is OBJECT (WP\_Post object) and the other values can be ARRAY\_A (associative array) or ARRAY\_N (numeric array).

*\$filter*

An optional string of how the context should be sanitized on output. The default value is raw, but other values can be edit, db, display, attribute, or js. Sanitization is covered in [Chapter 8](#).

## **GET\_POSTS( \$ARGS = NULL )**

This function returns a list of posts from matching criteria. This function uses the WP\_Query class, which you will see examples of throughout the book: \$args is an optional array of post arguments.

The defaults are:

*numberposts*

Default is 5. Total number of posts to retrieve. -1 is all.

*offset*

Default is 0. Number of posts to pass over.

*category*

What category to pull the posts from.

*orderby*

Default is post\_date. How to order the posts.

*order*

Default is DESC. The order to retrieve the posts.

*include*

A list of post IDs to include.

*exclude*

A list of post IDs to exclude.

*meta\_key*

Any metadata key.

*meta\_value*

Any metadata value. Must also use meta\_key.

*post\_type*

Default is post. Can be page, or attachment, or the slug for any custom CPT. The string any will return posts from all post types.

*post\_parent*

The parent ID of the post.

*post\_status*

Default is publish. Post status to retrieve.

## **WP\_DELETE\_POST( \$POSTID = 0, \$FORCE\_DELETE = FALSE )**

This function will trash any post or permanently delete it if \$force\_delete is set to true:

*\$postid*

A required integer of the post ID you would like to trash or delete.

*\$force\_delete*

An optional Boolean that, if set to true, will delete the post; if left blank, it will default to false and move the post to a deleted status.

Example 2-4 demonstrates some of the basic functions for interacting with the wp\_posts table.

### *Example 2-4. Working with the wp\_posts table*

---

```
<?php
// insert post - set post status to draft
$args = array(
    'post_title'    => 'Building Web Apps with WordPress',
    'post_excerpt'  => 'WordPress as an Application
Framework',
    'post_content'  => 'WordPress is the key to successful
cost effective
    web solutions in most situations. Build almost
anything on top of the
        WordPress platform. DO IT NOW!!!!',
    'post_status'   => 'draft',
    'post_type'     => 'post',
    'post_author'   => 1,
    'menu_order'   => 0
);
$post_id = wp_insert_post( $args );
```

```

echo 'post ID: ' . $post_id . '<br>';

// update post - change post status to publish
$args = array(
    'ID' => $post_id,
    'post_status' => 'publish'
);
wp_update_post( $args );

// get post - return post data as an object
$post = get_post( $post_id );
echo 'Object Title: ' . $post->post_title . '<br>';

// get post - return post data as an array
$post = get_post( $post_id, ARRAY_A );
echo 'Array Title: ' . $post['post_title'] . '<br>';

// delete post - skip the trash and permanently delete it
wp_delete_post( $post_id, true );

// get posts - return 100 posts
$posts = get_posts( array( 'numberposts' => '100' ) );
// loop all posts and display the ID & title
foreach ( $posts as $post ) {
    echo $post->ID . ': ' . $post->post_title . '<br>';
}

/*
The output from the above example should look something like
this:
post ID: 589
Object Title: Building Web Apps with WordPress
Array Title: Building Web Apps with WordPress
"A list of post IDs and Titles from your install"
*/
?>

```

## wp\_postmeta

At times you may want to store additional data along with a post. WordPress provides an easy way to do this without having to add

extra fields to the posts table. You can store as much post metadata as you need to in the `wp_postmeta` table. Each record is associated to a post through the `post_id` field. When editing any post in the backend of WordPress, you can add/update/delete metadata or custom fields via the UI. [Table 2-5](#) shows the database structure for the `wp_postmeta` table.

### NOTE

Metadata keys that start with an underscore are hidden from the Custom Fields UI on the edit post page. This is useful to hide certain meta fields that you don't want end users editing directly.

*Table 2-5. Database schema for wp\_postmeta table*

Column	Type	Collation	Null	Default	Extra
meta_id	bigint(20)		No	None	AUTO_INCREMENT
post_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_val	longtext	utf8_general_ci	Yes	NULL	

## Functions Found in /wp-includes/post.php

The following functions are found in */wp-includes/post.php*.

## **GET\_POST\_META( \$POST\_ID, \$KEY = "", \$SINGLE = FALSE )**

Get post metadata for a given post:

*\$post\_id*

A required integer of the post ID, for which you would like to retrieve post meta.

*\$key*

Optional string of the meta key name for which you would like to retrieve post meta. The default is to return metadata for all of the meta keys for a particular post.

*\$single*

A Boolean of whether to return a single value or not. The default is `false`, and the value will be returned as an array.

There can be more than one meta key for the same post ID with different values. If you set `$single` to `true`, you will get the first key's value; if it is set to `false`, you will get an array of the values of each record with the same key.

## **UPDATE\_POST\_META( \$POST\_ID, \$META\_KEY, \$META\_VALUE, \$PREV\_VALUE = "" )**

This function updates post metadata, but it also inserts metadata if the passed-in key doesn't already exist:

*\$post\_id*

A required integer of a post ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to store. If this meta key already exists, it will update the current row's meta value; if not, it will insert a new row.

`$meta_value`

A required mixed value of an integer, string, array, or object. Arrays and objects will automatically be serialized.

`$prev_value`

An optional mixed value of the current metadata value. If a match is found, it will replace the previous/current value with the new value you specified. If left blank, the new meta value will replace the first instance of the matching key. If you have five rows of metadata with the same key and you don't specify which row to update with this value, it will update the first row and remove the other four.

#### NOTE

This function relies on the `update_metadata()` function located in `/wp-includes/meta.php`. Check it out!

**`ADD_POST_META( $POST_ID, $META_KEY, $META_VALUE, $UNIQUE = FALSE )`**

This function inserts brand-new post meta into the `wp_postmeta` table. This function is used less these days because we can just use the previous function we talked about, `update_post_meta()`, to insert new rows as well as update them. If you want to ensure that a

given meta key is used only once per post, you should use this function and set the `$unique` parameter to `true`:

`$user_id`

A required integer of a post ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to store.

`$meta_value`

A required mixed value of an integer, string, array, or object.

`$unique`

An optional Boolean that, when set to `true`, will ensure that the meta key can be added only once for a given ID.

**`DELETE_POST_META( $POST_ID, $META_KEY, $META_VALUE = '' )`**

This function deletes post metadata for a provided post ID and matching key. You can also specify a matching meta value if you want to delete only that value and not other metadata rows with the same meta key:

`$post_id`

A required integer of a post ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to delete.

`$meta_value`

An optional mixed value of the meta value. If you have more than one record with the same meta key, you can specify which one to delete by matching this value. It defaults to nothing, which will delete all meta rows with a matching `post_id` and `meta_key`.

In [Example 2-5](#), we get the last post and add, update, and delete various post meta.

#### *Example 2-5. Working with post metadata*

---

```
<?php
// get posts - return the latest post
$posts = get_posts( array( 'numberposts' => '1', 'orderby' =>
    'post_date', 'order' => 'DESC' ) );
foreach ( $posts as $post ) {
    $post_id = $post->ID;

    // update post meta - public metadata
    $content = 'You SHOULD see this custom field when
editing your latest post.';
    update_post_meta( $post_id, 'bwawp_displayed_field',
$content );

    // update post meta - hidden metadata
    $content = str_replace( 'SHOULD', 'SHOULD NOT',
$content );
    update_post_meta( $post_id, '_bwawp_hidden_field',
$content );

    // array of student logins
    $students[] = 'dalya';
    $students[] = 'ashleigh';
    $students[] = 'lola';
    $students[] = 'isaac';
    $students[] = 'marin';
    $students[] = 'brian';
    $students[] = 'nina';
    $students[] = 'cam';
```

```

        // add post meta - one key with array as value, array
will be serialized
        // automatically
        add_post_meta( $post_id, 'bwawwp_students', $students,
true );

        // loop students and add post meta record for each
student
        foreach ( $students as $student ) {
            add_post_meta( $post_id, 'bwawwp_student',
$student );
        }

        // get post meta - get all meta keys
        $all_meta = get_post_meta( $post_id );
        echo '<pre>';
        print_r( $all_meta );
        echo '</pre>';

        // get post meta - get 1st instance of key
        $student = get_post_meta( $post_id, 'bwawwp_student',
true );
        echo 'oldest student: ' . $student;

        // delete post meta
        delete_post_meta( $post_id, 'bwawwp_student' );
    }

/*
The output from the above example should look something like
this:
Array
(
    [_bwawwp_hidden_field] => Array
        (
            [0] => You SHOULD NOT see this custom field when
editing your latest post.
        )

    [bwawwp_displayed_field] => Array
        (
            [0] => You SHOULD see this custom field when
editing your latest post.
        )
)

```

```

        )

[bwawwp_students] => Array
(
    [0] => a:7:
{i:0;s:5:"dalya";i:1;s:8:"ashleigh";i:2;s:4:"lola";i:3;s:5:
"isaac";i:4;s:5:"marin";i:5;s:5:"brian";i:6;s:4:"nina";i:6;s:5
:"cam";}
)

[bwawwp_student] => Array
(
    [0] => dalya
    [1] => ashleigh
    [2] => lola
    [3] => isaac
    [4] => marin
    [5] => brian
    [6] => nina
    [7] => cam
)
)
oldest student: dalya
*/
?>

```

## wp\_comments

Comments can be left on any post. The `wp_comments` table stores individual comments and associated comment data for any post.

Table 2-6 shows the database structure for the `wp_comments` table.

*Table 2-6. Database schema for wp\_comments table*

Column	Type	Collation	N u ll	Default	Extra
comment_ID	bigint (20)		N o	None	AUTO_INCREMENT
comment_post _ID	bigint (20)		N o	0	
comment_auth or	tinytex t	utf8_gen eral_ci	N o		
comment_auth or_email	varchar (100)	utf8_gen eral_ci	N o		
comment_auth or_url	varchar (200)	utf8_gen eral_ci	N o		
comment_auth or_IP	varchar (100)	utf8_gen eral_ci	N o		
comment_date	datetim e		N o	0000-00-00 00:00:00	
comment_date _gmt	datetim e		N o	0000-00-00 00:00:00	
comment_cont ent	text	utf8_gen eral_ci	N o	None	
comment_karm a	int(11)		N o	0	
comment_appr oved	varchar (20)	utf8_gen eral_ci	N o	1	
comment_agen t	varchar (20)	utf8_gen eral_ci	N o		

Column	Type	Collation	N u ll	Default	Extra
comment_type	varchar(20)	utf8_general_ci	N o		
comment_parent	bigint(20)		N o	0	
user_id	bigint(20)		N o	0	

## Functions Found in /wp-includes/comment.php

The functions that follow are found in */wp-includes/comment.php*.

### **GET\_COMMENT( \$COMMENT, \$OUTPUT = OBJECT )**

This function returns comment data from a comment ID or comment object. If the comment is empty, the global comment variable will be used if set:

`$comment`

An optional integer, string, or object of a comment ID or object.

`$output`

An optional string that defines what format the output should be in. Possible values are OBJECT, ARRAY\_A, and ARRAY\_N.

### **GET\_COMMENTS( \$ARGS = '' )**

This function retrieves a list of comments for specific posts or a single post. It calls the `WP_Comment_Query` class, which we cover

in the next chapter. \$args are an optional array or string of arguments to query comments. The default arguments are:

*author\_email*

A string of a comment author's email address.

*ID*

An integer of the ID of a comment.

*karma*

An integer of a comment's karma, which can be used by plugins for rating.

*number*

An integer of the number of comments to return. Default is all comments.

*offset*

An integer of the number of comments to pass over. Default is 0.

*orderby*

A string of the field in which to order the comment by. Allowed values are: comment\_agent, comment\_approved, comment\_author, comment\_author\_email, comment\_author\_IP, comment\_author\_url, comment\_content, comment\_date, comment\_date\_gmt, comment\_ID, comment\_karma, comment\_parent, comment\_post\_ID, comment\_type, and user\_id.

*order*

A string of how to order the selected `orderby` argument.  
Defaults to DESC and also accepts ASC.

*parent*

An integer of a comment's parent comment ID.

*post\_id*

An integer of the post ID to which a comment is attached.

*post\_author*

An integer of the post author ID to which a comment is attached.

*post\_name*

A string of the post name to which a comment is attached.

*post\_parent*

An integer of the post parent ID to which a comment is attached.

*post\_status*

A string of the post status to which a comment is attached.

*post\_type*

A string of the post type to which a comment is attached.

*status*

A string of the status of a comment. Optional values are hold, approve, spam, or trash.

*type*

A string of the type of a comment. Optional values are '', pingback, or trackback.

*user\_id*

An integer of the user ID of a comment.

*search*

A string of search terms on which you can search a comment.

Searches the following fields: `comment_author`,  
`comment_author_email`, `comment_author_url`,  
`comment_author_IP`, and `comment_content`.

*count*

A Boolean that will make the query return a count or results. The default value is `false`.

*meta\_key*

The comment meta key of comment meta to search on.

*meta\_value*

The comment meta value of comment meta to search on;  
`meta_key` is required.

## **WP\_INSERT\_COMMENT( \$COMMENTDATA )**

This function inserts a comment into the database:

*\$commentdata*

A required array of comment fields and values to be inserted.

Available fields to be inserted are: `comment_post_ID`,  
`comment_author`, `comment_author_email`,  
`comment_author_url`, `comment_author_IP`,  
`comment_date`, `comment_date_gmt`,  
`comment_content`, `comment_karma`,  
`comment_approved`, `comment_agent`, `comment_type`,  
`comment_parent`, and `user_id`.

## **WP\_UPDATE\_COMMENT( \$COMMENTARR )**

This function updates comment data and filters to make sure all required fields are valid before updating in the database:

*\$commentarr*

An optional array of arguments containing comment fields and values to be updated. These are the same field arguments just listed for the `wp_insert_comment()` function.

## **WP\_DELETE\_COMMENT( \$COMMENT\_ID, \$FORCE\_DELETE = FALSE )**

This function deletes a comment. By default, it will trash the comment unless you specify to permanently delete:

*\$comment\_id*

A required integer of the comment ID to trash/delete.

*\$force\_delete*

An optional Boolean that if set to `true` will permanently delete a comment. Example 2-6 demonstrates some of the basic functions for interacting with the `wp_comments` table.

Example 2-6 demonstrates managing comment data attached to a post.

*Example 2-6. Working with the wp\_comments table*

---

```
<?php
// insert post
$args = array(
    'post_title' => 'What should I do tonight?',
    'post_content' => 'Think of something cool to do and make a
comment about it!',
```

```

    'post_status' => 'publish'
);
$post_id = wp_insert_post( $args );
echo 'post ID: ' . $post_id . ' - ' . $args['post_title'] .
'<br>';

// make comments array
$comments[] = 'ICE CREAM!!!!';
$comments[] = 'Taco Bell';
$comments[] = 'Get a good night sleep';

//loop comments array
foreach ( $comments as $key => $comment ) {
    // insert comments
    $commentdata = array(
        'comment_post_ID' => $post_id,
        'comment_content' => $comments[$key],
    );
    $comment_ids[] = wp_insert_comment( $commentdata );
}

echo 'comments:<pre>';
print_r( $comments );
echo '</pre>';

// update comment
$commentarr['comment_ID'] = $comment_ids[0];
$commentarr['comment_content'] = 'Read this entire book';
wp_update_comment( $commentarr );

// insert comment - sub comment from parent id
$commentdata = array(
    'comment_post_ID' => $post_id,
    'comment_parent' => $comment_ids[0],
    'comment_content' => 'That is a pretty good idea...', 
);
wp_insert_comment( $commentdata );

// get comments - search taco bell
$comments = get_comments( 'search=Taco Bell&number=1' );
foreach ( $comments as $comment ) {
    // insert comment - sub comment of taco bell comment
}

```

```

id
$commentdata = array(
    'comment_post_ID' => $post_id,
    'comment_parent' => $comment->comment_ID,
    'comment_content' => '',
);
wp_insert_comment( $commentdata );
}

// get comment - count of comments for this post
$comment_count = get_comments( 'post_id=' . $post_id .
'&count=true' );
echo 'comment count: ' . $comment_count . '<br>';

// get comments - get all comments for this post
$comments = get_comments( 'post_id=' . $post_id );
foreach ( $comments as $comment ) {
    // update 1st comment
    if ( $comment_ids[0] == $comment->comment_ID ) {
        $commentarr = array(
            'comment_ID' => $comment->comment_ID,
            'comment_content' => $comment->comment_content . ' &
build some apps!',
        );
        wp_update_comment( $commentarr );
        // delete all other comments
    } else {
        // delete comment
        wp_delete_comment( $comment->comment_ID, true
    );
    }
}

// get comment - new comment count
$comment_count = get_comments( 'post_id=' . $post_id .
'&count=true' );
echo 'new comment count: ' . $comment_count . '<br>';

// get comment - get best comment
$comment = get_comment( $comment_ids[0] );
echo 'best comment: ' . $comment->comment_content;

/*
The output from the above example should look something like
this:
post ID: 91011 - What should I do tonight?
comments:

```

```

Array
(
    [0] => ICE CREAM!!!!
    [1] => Taco Bell
    [2] => Get a good night sleep
)
comment count: 5
new comment count: 1
best comment: Read this entire book & build some apps!
*/
?>

```

## wp\_commentsmeta

Just like the wp\_usermeta and wp\_postmeta table, this table stores any custom, additional data tied to a comment by the comment\_id fields. Table 2-7 shows the database structure for the wp\_commentsmeta table.

*Table 2-7. Database schema for wp\_commentsmeta table*

Column	Type	Collation	Null	Default	Extra
meta_id	bigint(20)		No	None	AUTO_INCREMENT
comment_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_val	longtext	utf8_general_ci	Yes	NULL	

## Functions Found in /wp-includes/comment.php

The following functions are found in */wp-includes/comment.php*.

**GET\_COMMENT\_META( \$COMMENT\_ID, \$KEY = "",  
\$SINGLE = FALSE )**

This function gets comment meta for a given comment ID:

*\$comment\_id*

A required integer of the comment ID for which you would like to retrieve comment meta.

*\$key*

Optional string of the meta key name for which you would like to retrieve comment meta. The default is to return metadata for all of the meta keys for a particular post.

*\$single*

A Boolean of whether to return a single value or not. The default is `false`, and the value will be returned as an array.

**ADD\_COMMENT\_META( \$COMMENT\_ID, \$META\_KEY,  
\$META\_VALUE, \$UNIQUE = FALSE )**

This function adds comment meta for given comment ID:

*\$comment\_id*

A required integer of a comment ID.

*\$meta\_key*

A required string of the meta key name for the meta value you would like to store.

*\$meta\_value*

A required mixed value of an integer, string, array, or object.

*\$unique*

An optional Boolean that, when set to `true`, will make sure the meta key can only ever be added once for a given ID.

**UPDATE\_COMMENT\_META( \$COMMENT\_ID,  
\$META\_KEY, \$META\_VALUE, \$PREV\_VALUE = "" )**

This function updates comment meta for a given comment ID:

*\$comment\_id*

A required integer of a comment ID.

*\$meta\_key*

A required string of the meta key name for the meta value you would like to store. If this meta key already exists, it will update the current row's meta value; if not, it will insert a new row.

*\$meta\_value*

A required mixed value of an integer, string, array, or object. Arrays and objects will automatically be serialized.

*\$prev\_value*

An optional mixed value of the current metadata value. If a match is found, it will replace the previous/current value with the new value you specified. If left blank, the new meta value will replace the first instance of the matching key. If you have five rows of metadata with the same key and you don't specify which row to update with this value, it will update the first row and remove the other four.

**DELETE\_COMMENT\_META( \$COMMENT\_ID,  
\$META\_KEY, \$META\_VALUE = "" )**

This function deletes comment metadata for a provided comment ID and matching key. You can also specify a matching meta value if you only want to delete that value and not other metadata rows with the same meta key:

`$comment_id`

A required integer of a comment ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to delete.

`$meta_value`

An optional mixed value of the meta value. If you have more than one record with the same meta key, you can specify which one to delete by matching this value. It defaults to nothing, which will delete all meta rows with a matching `post_id` and `meta_key`.

Example 2-7 demonstrates some of the basic functions for interacting with the `wp_commentsmeta` table.

---

*Example 2-7. Working with the `wp_commentsmeta` table*

---

```
<?php
// get comments - last comment ID
$comments = get_comments( 'number=1' );
foreach ( $comments as $comment ) {
    $comment_id = $comment->comment_ID;

    // add comment meta - meta for view date & IP address
    $viewed = array( date( "m.d.y" ),
    $_SERVER["REMOTE_ADDR"] );
    $comment_meta_id = add_comment_meta( $comment_id,
    'bwawp_view_date',
    $viewed, true );
```

```

echo 'comment meta id: ' . $comment_meta_id;

// update comment meta - change date format to format
like
// October 23, 2020, 12:00 am instead of 10.23.20
$viewed = array( date( "F j, Y, g:i a" ),
$_SERVER["REMOTE_ADDR"] );
update_comment_meta( $comment_id, 'bwawwp_view_date',
$viewed );

// get comment meta - all keys
$comment_meta = get_comment_meta( $comment_id );
echo '<pre>';
print_r( $comment_meta );
echo '</pre>';

// delete comment meta
delete_comment_meta( $comment_id, 'bwawwp_view_date'
);
}

/*
The output from the above example should look something like
this:
comment meta id: 16
Array
(
    [bwawwp_view_date] => Array
        (
            [0] => a:2:{i:0;s:24:"August 11, 2018, 4:16
pm";i:1;s:9:"127.0.0.1";}
        )
)
*/
?>

```

## wp\_terms

The wp\_terms table stores each category name or term name that you create. Each record is tied to its taxonomy in the wp\_term\_taxonomy table by the term\_id. So you're familiar

with post categories and tags? Well, each category or tag is stored in this table, and technically they are both taxonomies. Every term that is stored in the name column is a taxonomy term. We will be covering taxonomies in much more detail in [Chapter 5](#), so if you don't fully grasp what a taxonomy is, you will soon. [Table 2-8](#) shows the database structure for the wp\_terms table.

*Table 2-8. Database schema for wp\_terms table*

Column	Type	Collation	Null	Default	Extra
term_id	bigint(20)		No	None	AUTO_INCREMENT
name	varchar(200)		No		
slug	varchar(200)	utf8_general_ci	No		
term_group	bigint(10)		No	0	

## Functions Found in /wp-includes/taxonomy.php

The following functions are found in */wp-includes/taxonomy.php*.

### **GET\_TERMS( \$TAXONOMIES, \$ARGS = '' )**

This function gets the terms of a specific taxonomy or an array of taxonomies:

*\$taxonomies*

A required string or array of a taxonomy or list of taxonomies.

#### *\$args*

An optional string or array of arguments. Available arguments are:

#### *orderby*

Default is name. Can be name, count, term\_group, slug, or nothing, which will use term\_id. Passing a custom value other than these will cause the terms to be ordered on that custom value.

#### *order*

ASC or DESC. The default is ASC.

#### *hide\_empty*

The default value is true, which will only return terms that are attached to a post. If set to false, you can return all terms regardless of whether they are being used by a post or not.

#### *exclude*

An array or comma-separated or space-delimited string of term IDs to exclude from the query results. If include is being used, exclude will be ignored.

#### *exclude\_tree*

An array or comma-separated or space-delimited string of term IDs to exclude from the query results, including any child terms. If include is being used, exclude\_tree will be ignored.

#### *include*

An array or comma-separated or space-delimited string of term IDs to include in the query results.

*number*

The number of terms for the query to return. The default is all.

*offset*

The number by which to offset the terms query.

*fields*

You can specify if you want to return term IDs or names. The default is all, which returns an array of term objects.

*slug*

A string that will return any terms that have a matching slug.

*hierarchical*

Includes all child terms if they are attached to posts. The default is true, so to not return terms hierarchically, set this value to false.

*search*

A string that will return any terms whose names match the value provided. The search is case-insensitive.

*name\_like*

A string that will return any terms whose names begin with the value provided. Like search, this string is case-insensitive.

*pad\_counts*

If set to true, the query results will include the count of each term's children.

*get*

If set to `all`, returns terms regardless of ancestry or whether the terms are empty.

*child\_of*

When set to a term ID, the query results will contain all descendants of the provided term ID. The default is `0`, which returns everything.

*parent*

When set to a term ID, the query results will contain the direct children of the provided term ID. The default is an empty string.

*cache\_domain*

Enables a unique cache key to be produced when this query is stored in object cache.

## **GET\_TERM( \$TERM, \$TAXONOMY, \$OUTPUT = OBJECT, \$FILTER = 'RAW' )**

This function gets all term data for any given term:

*\$term*

A required integer or object of the term to return.

*\$taxonomy*

A required string of the taxonomy of the term to return.

*\$output*

An optional string of the output format. The default value is `OBJECT`, and the other values can be `ARRAY_A` (associative array) or `ARRAY_N` (numeric array).

*\$filter*

An optional string of how the context should be sanitized on output. The default value is `raw`.

## **WP\_INSERT\_TERM( \$TERM, \$TAXONOMY, \$ARGS = ARRAY() )**

This function adds a new term to the database:

*\$term*

A required string of the term to add or update.

*\$taxonomy*

A required string of the taxonomy to which the term will be added.

*\$args*

An optional array or string of term arguments to be inserted/updated. Available arguments are as follows:

*alias\_of*

An optional string of the slug of which the term will be an alias.

*description*

An optional string that describes the term.

*parent*

An optional integer of the parent term ID of which this term will be a child.

*slug*

An optional string of the slug of the term.

## **WP\_UPDATE\_TERM( \$TERM\_ID, \$TAXONOMY, \$ARGS = ARRAY() )**

This function updates an existing term in the database:

*\$term\_id*

A required integer of the term ID of the term you want to update.

*\$taxonomy*

A required string of the taxonomy with which the term is associated.

*\$args*

An optional array or string of term arguments to be updated.

These are the same arguments used in `wp_insert_term()`.

## **WP\_DELETE\_TERM( \$TERM, \$TAXONOMY, \$ARGS = ARRAY() )**

This function deletes a term from the database. If the term is a parent of other terms, the children will be updated to that term's parent:

*\$term*

A required integer of the term ID of the term you want to delete.

*\$taxonomy*

A required string of the taxonomy with which the term is associated.

*\$args*

An optional array to overwrite term field values.

## **wp\_termmeta**

Since WordPress 4.4, metadata can be stored for terms. The [Simple Taxonomy Ordering plugin by YIKES, Inc.](#) uses term meta to allow you to reorder how your categories and other taxonomies show up in lists and widgets.

*Table 2-9. Database schema for wp\_termmeta table*

Column	Type	Collation	Null	Default	Extra
meta_id	bigint(20)		No	None	AUTO_INCREMENT
term_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_value	longtext	utf8_general_ci	Yes	NULL	

The functions that follow work similarly to the variants for user meta, post meta, and comment meta.

**GET\_TERM\_META( \$TERM\_ID, \$KEY = "", \$SINGLE = FALSE )**

This function gets a term's meta value for a specified key:

*\$term\_id*

A required integer of a term ID.

*\$key*

An optional string of the meta key of the value you would like to return. If blank, all metadata for the given term will be returned.

*\$single*

A Boolean for whether to return a single value. The default is `false`; thus, the value will be returned as an array.

There can be more than one meta key for the same term ID with different values. If you set `$single` to `true`, you will get the first key's value; if you set it to `false`, you will get an array of the values of each record with the same key.

**UPDATE\_TERM\_META( \$TERM\_ID, \$META\_KEY, \$META\_VALUE, \$PREV\_VALUE = '' )**

This function updates term metadata but will also insert metadata if the passed-in key doesn't already exist:

*\$term\_id*

A required integer of a term ID.

*\$meta\_key*

A required string of the meta key name for the meta value you would like to store. If this meta key already exists, it will update the current row's meta value; if not, it will insert a new row.

*\$meta\_value*

A required mixed value of an integer, string, array, or object. Arrays and objects will automatically be serialized.

*\$prev\_value*

An optional mixed value of the current metadata value. If a match is found, it will replace the previous/current value with the new value you specified. If left blank, the new meta value will replace the first instance of the matching key. If you have five rows of metadata with the same key and you don't specify which row to update with this value, it will update the first row and remove the other four.

### NOTE

This function relies on the `update_metadata()` function located in `/wp-includes/meta.php`. Check it out!

## **ADD\_TERM\_META( \$TERM\_ID, \$META\_KEY, \$META\_VALUE, \$UNIQUE = FALSE )**

This function inserts brand-new term meta into the `wp_termmeta` table. Again, it is preferred to use `update_term_meta()` to insert new rows as well as update them. If you want to ensure that a given meta key is used only once per term, you should use this function and set the `$unique` parameter to `true`:

`$term_id`

A required integer of a term ID.

`$meta_key`

A required string of the meta key name for the meta value you would like to store.

`$meta_value`

A required mixed value of an integer, string, array, or object.

*\$unique*

An optional Boolean that, when set to `true`, will make sure the meta key can be added only once for a given ID.

**`DELETE_TERM_META( $TERM_ID, $META_KEY, $META_VALUE = '' )`**

This function deletes term metadata for a provided term ID and matching key. You can also specify a matching meta value if you want to delete only that value and not other metadata rows with the same meta key:

*\$term\_id*

A required integer of a term ID.

*\$meta\_key*

A required string of the meta key name for the meta value you would like to delete.

*\$meta\_value*

An optional mixed value of the meta value. If you have more than one record with the same meta key, you can specify which one to delete by matching the meta value. It defaults to nothing, which will delete all meta rows with a matching `term_id` and `meta_key`.

## **wp\_term\_taxonomy**

The `wp_term_taxonomy` table stores each taxonomy type you are using. WordPress has two taxonomy types built in, `category` and `post_tag`, but you can also register your own taxonomies. When a

new term is added in the `wp_terms` table, it is associated with its taxonomy in this table, along with that taxonomy term ID, description, parent, and count. [Table 2-10](#) shows the structure for the `wp_term_taxonomy` table.

*Table 2-10. Database schema for wp\_term\_taxonomy table*

Column	Type	Collation	Null	Default	Extra
<code>term_taxonomy_id</code>	<code>bigint(20)</code>		NO	None	<code>AUTO_INCREMENT</code>
<code>term_id</code>	<code>bigint(20)</code>		NO	0	
<code>taxonomy</code>	<code>varchar(32)</code>	<code>utf8_general_ci</code>	NO		
<code>description</code>	<code>longtext</code>	<code>utf8_general_ci</code>	NO	None	
<code>parent</code>	<code>bigint(20)</code>		NO	0	
<code>count</code>	<code>bigint(20)</code>		NO	0	

## /wp-includes/taxonomy.php

You can find the following functions in `/wp-includes/taxonomy.php`.

**`GET_TAXONOMIES( $ARGS = ARRAY(), $OUTPUT = 'NAMES', $OPERATOR = 'AND' )`**

This function returns a list of registered taxonomy objects or a list of taxonomy names:

*\$args*

An optional array of arguments to query what taxonomy objects are returned. There are a lot, and we cover all of them in [Chapter 5](#).

*\$output*

An optional string of either names or objects. The default is names, which returns a list of taxonomy names.

*\$operator*

An optional string of either and or or. The default is and, meaning all the arguments passed in must match. If set to or, any arguments passed in can match.

## **GET\_TAXONOMY( \$TAXONOMY )**

This function will first check that the parameter string given is a taxonomy object; if it is, it will return it:

*\$taxonomy*

A required string of the name of the taxonomy object to return.

## **REGISTER\_TAXONOMY( \$TAXONOMY, \$OBJECT\_TYPE, \$ARGS = ARRAY() )**

This function creates or updates a taxonomy object. Registering custom taxonomies can really extend WordPress because you can categorize your posts any way you see fit. We'll go over registering taxonomies in much more detail in [Chapter 5](#):

*\$taxonomy*

A required string of the name of the taxonomy.

`$object_type`

A required array or string of the object types (post types like post and page) to which this taxonomy will be tied.

`$args`

An optional array or string of arguments. There are a lot of these, and we cover all of them in [Chapter 5](#).

## **wp\_term\_relationships**

The `wp_term_relationships` table relates a taxonomy term to a post. Every time you assign a category or tag to a post, it's being linked to that post in this table. [Table 2-11](#) shows the structure for the `wp_term_relationships` table.

*Table 2-11. Database schema for wp\_term\_relationships table*

Column	Type	Collation	Null	Default	Extra
object_id	bigint(20)		No	0	
term_taxonomy_id	bigint(20)		No	0	
term_order	int(11)		No	0	

**GET\_OBJECT\_TAXONOMIES( \$OBJECT, \$OUTPUT = 'NAMES' )**

This function returns all taxonomies associated with a post type or post object:

*\$object*

A required array, string, or object of the name(s) of the post type(s) or post object(s).

*\$output*

An optional string of either names or objects. The default is names, which returns a list of taxonomy names.

### **WP\_GET\_OBJECT\_TERMS( \$OBJECT\_IDS, \$TAXONOMIES, \$ARGS = ARRAY() )**

This function returns terms associated with a supplied post object ID or IDs and a supplied taxonomy:

*\$object\_ids*

A required string or array of object IDs for the object terms you would like to return. Passing in a post ID would return terms associated with that post ID.

*\$taxonomies*

A required string or array of the taxonomy names from which you want to return terms. Passing in the taxonomy post\_tag would return terms of the post\_tag taxonomy.

*\$args*

An optional array or string of arguments that change how the data is returned. Here are the arguments you can change:

*orderby*

Defaults to name; also accepts count, slug, term\_group, term\_order, and none.

*order*

Defaults to ASC; also accepts DESC.

*fields*

Defaults to all; also accepts ids, names, slugs, and all\_with\_object\_id. This argument dictates what values are returned.

## **WP\_SET\_OBJECT\_TERMS( \$OBJECT\_ID, \$TERMS, \$TAXONOMY, \$APPEND = FALSE )**

This function adds taxonomy terms to a provided object ID and taxonomy. It has the ability to overwrite all terms or to append new terms to existing terms. If a term passed into this function doesn't already exist, it will be created and then related to the provided object ID and taxonomy:

*\$object\_id*

A required integer of the object ID (post ID) to which to relate your terms.

*\$terms*

A required array, integer, or string of the terms you want to add to an object (post).

*\$taxonomy*

A required array or string of the taxonomy or taxonomies to which you want to relate your terms.

*\$append*

An optional Boolean that defaults to `false` and will replace any existing terms related to an object ID with the new terms you provided. If it is set to `true`, your new terms will be appended to the existing terms.

### NOTE

There was once discussion of a future WordPress release to remove the `wp_terms` table. The `name` and `slug` columns of `wp_terms` would be moved into the `wp_terms_taxonomy` table, and a MySQL view created called `wp_terms`, that could be queried against, preserving backward compatibility for custom queries. This update was shelved, but it would be nice to clean up some tables around terms and taxonomies.

## Hooks: Actions and Filters

WordPress developers hook for a living! Hooks are great, and they make adding functionality into WordPress plugins and themes simple and easy.

There are two types of hooks—actions and filters—and they are two of the most powerful tools in WordPress. We spend a lot of time working with actions and filters. Understanding this section is important to your growth as a WordPress developer.

### Actions

Where an action hook (technically a `do_action()` function) exists in code running on WordPress, you can insert your code by calling

the `add_action()` function and passing in the action hook name and custom function with the code you want to run:

```
do_action( $tag, $arg );
```

Here are the available arguments:

`$tag`

The name of the action hook being executed.

`$arg`

One or more additional arguments that are passed through to the function called from the `add_action()` function referencing this `do_action()` function. Say what? Keep reading...

You can create your own hook in a theme or plugin by adding your own `do_action()` functions. However, most of the time you will be using established hooks in the WordPress core or other plugins.

For example, suppose you wanted to check whether a user was logged in when WordPress first loads up but before any output is displayed to the browser. You can use the `init` hook:

```
<?php
add_action( 'init', 'my_user_check' );

function my_user_check() {
    if ( is_user_logged_in() ) {
        // do something because a user is logged in
    }
}
?>
```

So what just happened? In the core of WordPress, there is an action hook, `do_action('init')`, and we are calling a function called `my_user_check()` from the `add_action()` function. At whatever point in time the code is being executed, when it gets to the `init` action hook, it will run our custom `my_user_check()` function to do whatever we want before continuing on.

### NOTE

Check out [WordPress's reference page](#) for a list of the most used WordPress hooks.

## Filters

Filters are kind of like action hooks in the sense that you can tap into them wherever they exist in WordPress. However, instead of inserting your own code where the hook or `do_action()` exists, you are filtering the returned value of existing functions that are using the `apply_filters()` function in WordPress core, plugins, and/or themes. In other words, by utilizing filters, you can hijack content before it is inserted into the database or before it is displayed to the browser as HTML:

```
apply_filters( $tag, $value, $var );
```

`$tag`

The name of the filter hook.

`$value`

The value on which the filter can be applied.

*\$var*

Any additional variables, such as a string or an array, passed into the filter function.

If you search the core WordPress files for `apply_filters()` you will find that this function is called all over the place, and like action hooks, it can also be added to and called from any theme or plugin. Anywhere in code running on your WordPress site that you see the `apply_filters()` function being called, you can filter the value being returned by that function. For our example, we are going to filter the title of all posts before they are displayed to the browser. We can hook into any existing filters using the `add_filter()` function:

```
add_filter( $tag, $function, $priority, $accepted_args );
```

*\$tag*

The name of the filter hook you want to filter. This should match the `$tag` parameter of the `apply_filters()` function call for which you want to filter the results.

*\$function*

The name of the custom function used to actually filter the results.

*\$priority*

This number sets the priority in which your `add_filter()` function will run compared to other places in the code that might

be referencing the same filter hook tag. By default, this value is 10.

### *\$accepted\_args*

You can set the number of parameters that your custom function that handles the filtering can accept. The default is 1, which is the `$value` parameter of the `apply_filters()` function.

OK, so how would real code for this look? Let's start by adding a filter to alter the title of any post returned to the browser. We know of a filter hook for `the_title` that looks like this:

```
apply_filters( 'the_title', $title, $id );
```

`$title` is the title of the post, and `$id` is the ID of the post:

```
<?php
add_filter( 'the_title', 'my_filtered_title', 10, 2 );

function my_filtered_title( $value, $id ) {
    $value = '[' . $value . ']';
    return $value;
}
?>
```

The preceding code should wrap any post titles in brackets. If your post title was “hello world,” it would now read “[hello world].” Note that we didn’t use the `$id` in our custom function. If we wanted to, we could have applied the brackets to only specific post IDs.

## NOTE

While `add_action()` is meant to be used with `do_action()` hooks, and `add_filter()` is meant to be used with `apply_filters()` hooks, the functions work the same way and are interchangeable. For readability, it is still a good idea to use the proper function depending on whether you intend to return a filtered result or just perform some code at a specific time.

# Development and Hosting Environments

WordPress needs a web server to run. This section covers the basics of setting up a local and remote environment for your WordPress app.

## Working Locally

Before we get into WordPress basics, we suggest that you set up a local development environment to run your WordPress installation. You can run WordPress on your desktop computer or laptop, which will make development more efficient and faster. You can run code locally versus using FTP or deploying code from a code repository to a web server. You can also write and test code without having an internet connection. There's a productive workday with your laptop and Mother Nature somewhere in your future.

There are a ton of online resources on how to run WordPress locally, so we aren't going to cover them all in detail, but based on your OS and preferences you will most likely be setting up a MAMP (Mac, Apache, MySQL, PHP), WAMP (Windows, Apache, MySQL, PHP), or LAMP (Linux, Apache, MySQL, PHP) stack. Also check out XAMPP and AMPPS, which are cross-platform stacks.

There are a few local web server tools developed specifically for WordPress. [DesktopServer by ServerPress](#) and [Local by Flywheel](#) are popular tools that streamline all of the semitedious configuration settings for setting up local WordPress installs via their UI.

Tools like VirtualBox, Vagrant, and Docker are also popular in the WordPress community. These tools are especially good for certain automated testing and deployment schemes. [WordPress.org](#) has a good document for [installing Varying Vagrant Vagrants \(VVV\)](#), which is popular among core contributors.

## Choosing a Web Host

If you don't already know what a web host is, you have some Googling to do. In short, a web host is where your website resides, specifically your WordPress directory and database.

We are assuming that most of you reading this already work or have worked with a web-hosting company, or maybe you work for a company that has its own internal web servers. Choosing a good host for your web application is key to ensuring it's fast, secure, and scalable. There are lots of web hosting companies out there, and pretty much all of them will be able to host WordPress, but not all of them are optimized specifically for WordPress.

A “Managed” WordPress hosting company would be the best choice for a high-scale WordPress-powered web application because its focus is on providing optimized WordPress hosting environments. For large-scale websites, a lot of work must go into the caching of the

data because of the way the WordPress database is structured with storing metadata. If you use WordPress as a large CMS with lots of posts with lots of post meta on an insufficiently powerful server, it will start to buckle and eat up server resources. Hosting companies with a focus on optimized WordPress have built-in caching systems that help optimize your content delivery.

You can always use caching plugins if your host doesn't do all the caching for you ([Chapter 14](#) covers caching and other scaling considerations if you are managing your own environment). Some managed WordPress hosts offer other advanced tools like automatic plugin updates and services like CDN integration geared toward optimizing your hosted WordPress environment.

Some examples of managed WordPress hosting companies include WP Engine, WordPress VIP, and Pagely. There are many other options for hosting. We maintain an up-to-date list of hosts for all sizes at [our website](#).

## **Development, Staging, and Production Environments**

A typical website project will have three code environments: Development, Staging, and Production. Ideally, all three environments are separate web servers, with separate databases, and separate installations of WordPress.

The Development environment (sometimes referred to as Dev or DEV) is where you do your new coding and maintenance work.

Typically, it's the local environment you set up using the information earlier in this chapter.

The Staging environment (sometimes referred to as Testing or TEST) is where you do your testing. The data on the Staging site should be as close as possible to the Production site. This means that you will sometimes be exporting data from the live site, scrubbing it to remove private user information, and importing it into the Staging site.

The Production environment (sometimes referred to as Live or PRD) is the real website your users visit and interact with.

If you can't have all three code environments, at the very least you want to have a separate Development and Production environment. You should never update code on a Production website without testing it thoroughly on a Development site.

Ideally you would do all of your development locally and then commit your code to a code repository and deploy it to a Development site that can have multiple developers working on it at the same time. Then, once it's ready to be thoroughly tested, maybe by your boss or client in an environment that is an exact copy of the Production site, you would deploy your code to the Staging site. Finally, once everything checks out on the Staging site, you deploy your code into Production. Make backups of the Production website before deploying any code in case you need to roll back any updates. And finally, never assume that your code will work without testing it.

Some managed WordPress hosting companies like WP Engine have automatic Staging sites for every WordPress site you have with it, which is a really cool feature.

### NOTE

If you were thinking about FTP when “deploying code” was mentioned earlier, you should up your game and use a source code repository like GitHub.

## Extending WordPress

Now you know the fundamentals of how WordPress is set up, how the data is stored, and the basic tools for manipulating that data. You’ve been introduced to action and filter hooks, which are a primary method for extending WordPress.

We cover more of the various built-in functions and methods used to interact with WordPress data throughout the book. Chapter 3 covers the WordPress Plugin API, including some of the key features of WordPress that make extending it easy, powerful, and consistent!

---

<sup>1</sup> ...ever, ever, ever...

<sup>2</sup> The third parameter for `add_option`, which was deprecated in 2.3, used to be a “description” string that was stored along with the option in the `wp_options` table.

# Chapter 3. Using WordPress Plugins

---

Plugins are awesome! If you didn't know, now you know! Plugins can help you deploy a full-blown web application with little to no knowledge of actual code. Whether you are using a free plugin, premium plugin, or building your own, plugins can extend WordPress to give you the functionality your application requires.

As we mentioned earlier, the great advantage of open source software is that members of the community are invested in improving WordPress and often build plugins to achieve a desired feature. The definition of a plugin provided in the WordPress Codex is “a program, or a set of one or more functions, written in the PHP scripting language, that adds a specific set of features or services to the WordPress weblog, which can be seamlessly integrated with the weblog using access points and methods provided by the WordPress Plugin Application Program Interface (API).” Plugins allow you to turn your site into anything you can think of, from a basic blog to an ecommerce site to a social network to a mobile iOS and Android app.

There are a couple of plugins that come standard with any new WordPress installation: Hello Dolly and Akismet. If you didn't know, the Hello Dolly plugin adds a random lyric from the song “Hello Dolly” (from the musical of the same name) to the top of your dashboard on each page load. It's not useful, but it's a good way to

see how to structure your own plugins. The Akismet plugin integrates with *Akismet.com* to automatically filter out spam comments from your blog. While Hello Dolly is useless outside of its educational value, Akismet is downright necessary on any site with commenting turned on. You always have the ability to deactivate these plugins or delete them altogether if you do not see any use for them on your site.

You can access more than 55,000 plugins through the official [WordPress plugin repository](#). As not all plugins are found there, do an internet search for additional functionality you need. Many plugin creators have their work downloadable through personal or business sites, and many charge a fee. Premium plugins, which you have to pay to use, are also available. Similar to mobile apps, there may be a scaled-down version of the premium plugin available for free, with a more involved version available for the fee. Most premium plugins also offer developer licenses, allowing developers building multiple sites to pay one price for the plugin files, which can then be installed on multiple WordPress installations.

### NOTE

Be careful when searching for premium plugins on Google. Always make sure you are getting a plugin from the plugin author's actual site or their official repositories. Lots of sites (also so-called "GPL Clubs") have cropped up offering "nulled," "cracked," or "warez" versions of premium plugins for free, or at drastically reduced prices. Most of these plugins include malware or other harmful code you obviously don't want.

## The General Public License, Version 2, License

No matter how you purchase or obtain a WordPress plugin, all WordPress plugins must use the General Public License, Version 2 (GPLv2), code license. This states that if the source code is *distributed* (made available, sold online, etc.), you can do anything with the code so long as any derivative work retains the GPLv2 license. Some themes and plugins may use a split license, meaning the HTML, CSS, JavaScript, and images are distributed under a different license than the PHP files. Some themes and plugins do not mention the GPLv2 license or flat out deny it applies. There is a bit of legal merit to their claims, but the authority figures in the WordPress.org community (namely Matt Mullenweg) state that all themes and plugins must be GPL compatible. Our stance is that if you want to do business in the WordPress community, you should follow their rules.

Overall, plugins are a great way to add enhanced functionality to your website without having to change any core WordPress files. If you are looking for a specific feature, you should first do a search to make sure that a plugin does not already exist for that functionality. If not, then you have two options: you can choose to download and modify an existing plugin, or build a new one from scratch.

## Installing WordPress Plugins

To install a WordPress plugin, simply log in to the WordPress administrator dashboard of your site, also known as the backend.

Click on the Plugins section, as shown in [Figure 3-1](#). You then have the option to search the WordPress plugin repository or upload one if you have already downloaded a plugin from the repository or another source. Once you have completed your search and found a plugin you are interested in, click to install it. With the plugin installed, you then have the option to activate it. If you do not activate the plugin, it remains deactivated on the “Plugins → Installed Plugins” page of your site. Also, please keep in mind that many plugins will need to be configured once activated, and usually you will see a message in the dashboard reminding you to do so.

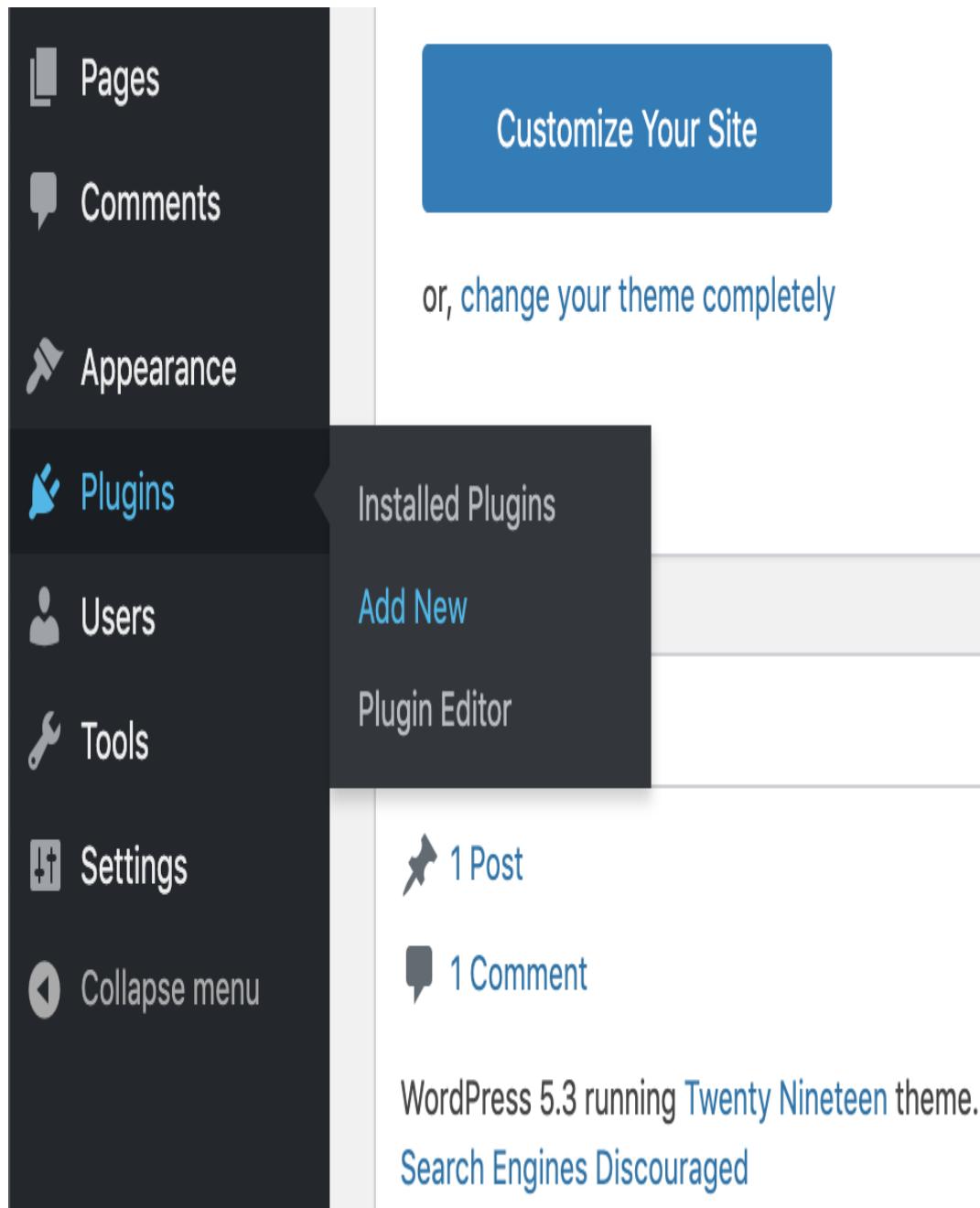


Figure 3-1. Adding a new plugin

If you downloaded a plugin from a source other than the official WordPress plugin repository, you should have a ZIP file of the plugin files. To upload the plugin to your site, in the Plugins section of the dashboard click Upload, and then choose that ZIP file from wherever

you have it saved on your computer. You will be given the choice to activate the plugin if you wish to do so at that time.

## Building Your Own Plugin

The true power of WordPress for app developers is that you can make your own custom plugin to extend WordPress anyway you see fit.

To create a plugin, first create a new folder in *wp-content/plugins* called *my-plugin* and make a PHP file in that folder called *my-plugin.php*. In *my-plugin.php*, write the following code, and feel free to replace any of the values:

```
<?php
/*
 * Plugin Name: My Plugin
 * Plugin URI: https://bwawwp.com/my-plugin/
 * Description: This is my plugin description.
 * Author: messenlehner, strangerstudios
 * Version: 1.0.0
 * Author URI: https://bwawwp.com
 * License: GPL-2.0+
 * License URI: http://www.gnu.org/licenses/gpl-2.0.txt
 */
?>
```

Save your *my-plugin.php* file. Congratulations, you are a WordPress plugin author! Even though your plugin doesn't do anything yet, you should be able to see it in */wp-admin/plugins.php* and activate it. Go ahead and activate it.

Let's make your plugin actually do something. We'll add some text to the footer of your WordPress installation. Copy and save the following code after the plugin information:

```
<?php
function my_plugin_wp_footer() {
    echo 'I read Building Web Apps with WordPress
        and now I am a WordPress Genius!';
}
add_action( 'wp_footer', 'my_plugin_wp_footer' );
?>
```

After you refresh and go to the frontend of your website, you should notice a new message in the footer. Now you are off to the races, and you can customize this basic plugin to do whatever you want. If you are already a PHP developer, start hacking away! If you are new to PHP and WordPress, a good way to kickstart your skills is to download and analyze the code in other plugins to see how they are doing what they do.

This was just a very basic example to get you started. We will explore more code that you can use in any of the plugins that you build throughout the book.

## File Structure for an App Plugin

When you're building a web app with WordPress, we recommend having one main app plugin to store your app's core functionality. On the theme side (covered in [Chapter 4](#)), you will store the majority of your app's frontend code in the active theme.

Some plugins do only one or two things, and one *.php* file is all you need to get things done. Your main app plugin will probably be much more complicated, with asset files (CSS, images, and templates),

included libraries, class files, and potentially thousands of lines of code you will want to organize into more than one file.

The following shows our proposed file structure for an app's main plugin, using the SchoolPress plugin as an example. Not all of these folders and files may be necessary. We add them to a plugin as needed:

- */plugins/schoolpress/adminpages/*
- */plugins/schoolpress/classes/*
- */plugins/schoolpress/css/*
- */plugins/schoolpress/css/admin.css*
- */plugins/schoolpress/css/frontend.css*
- */plugins/schoolpress/js/*
- */plugins/schoolpress/images/*
- */plugins/schoolpress/includes/*
- */plugins/schoolpress/includes/lib/*
- */plugins/schoolpress/includes/functions.php*
- */plugins/schoolpress/pages/*
- */plugins/schoolpress/services/*
- */plugins/schoolpress/scheduled/*
- */plugins/schoolpress/schoolpress.php*

## **/adminpages/**

Place the *.php* files for any dashboard page you add through your plugin in the */adminpages/* directory. For example, here is how you would add a dashboard page and load it out of your */adminpages/* directory:

```
<?php
// add a SchoolPress menu with reports page
function sp_admin_menu() {
    add_menu_page(
        'SchoolPress',
        'SchoolPress',
        'manage_options',
        'sp_reports',
        'sp_reports_page'
    );
}
add_action( 'admin_menu', 'sp_admin_menu' );

// function to load admin page
function sp_reports_page() {
    require_once dirname( __FILE__ ) .
    "/adminpages/reports.php";
}
?>
```

## **/classes/**

Place PHP class definitions in the */classes/* directory. In general, each file in this directory should include just one class definition. The class files should have names like *class.<ClassName>.php*, where *ClassName* is the name given to the class.

## **/css/**

Place CSS files used specifically for your plugin in the */css/* directory. Split your CSS into *admin.css* and *frontend.css* files depending on

whether the CSS affects the WordPress dashboard or something on the frontend. For example, you can also place the CSS libraries needed to support an included JavaScript library in this folder.

Here is some code to enqueue the *admin.css* and *frontend.css* styles from the plugin's CSS folder:

```
<?php

function sp_load_admin_styles() {
    wp_enqueue_style(
        'schoolpress-plugin-admin',
        plugins_url( 'css/admin.css', __FILE__ ),
        array(),
        SCHOOLPRESS_VERSION,
        'screen'
    );
}

add_action( 'admin_enqueue_scripts', 'sp_load_admin_styles'
);

function sp_load_frontend_styles() {
    wp_enqueue_style(
        'schoolpress-plugin-frontend',
        plugins_url( 'css/frontend.css', __FILE__ ),
        array(),
        SCHOOLPRESS_VERSION,
        'screen'
    );
}

add_action( 'wp_enqueue_scripts', 'sp_load_frontend_styles'
);
?>
```

The `admin_enqueue_scripts` hook should be used to enqueue scripts and styles meant for the administrator dashboard. The

`wp_enqueue_scripts` hook should be used to enqueue scripts and styles meant for the frontend.

CSS that affects components of the WordPress dashboard should go into the *admin.css* file. CSS that affects the frontend of the site should go into *frontend.css*, but be careful when adding CSS rules to the *frontend.css* file. When adding frontend styles to your plugin files, ask yourself first whether the CSS rules you are writing should go into the app's theme instead, since the majority of your frontend-style code should be handled by your theme.

The CSS that would go into the plugin's CSS file are generally layout styles that would be appropriate no matter what theme was loaded. Imagine that your site had no theme or CSS loaded at all. What would be the bare minimum CSS needed to have the HTML generated by your plugin make sense? Expect the theme to build on and override that.

For example, your plugin's *frontend.css* should never include styles for coloring. However, a style saying an avatar is 64 pixels wide and floated left could be appropriate.

## /js/

Place any JavaScript files needed by your plugin in this folder. Again, you can split things into an *admin.js* and *frontend.js* file depending on where the JavaScript is needed.

You can also add to this folder third-party JavaScript libraries that you use. Generally, these should be added to a subfolder of the /js/

directory.

Here is some example code to load *admin.js* and *frontend.js* files from your plugin's */js/* directory:

```
<?php

function sp_load_admin_scripts() {
    wp_enqueue_script(
        'schoolpress-plugin-admin',
        plugins_url( 'js/admin.js', __FILE__ ),
        array( 'jquery' ),
        SCHOOLPRESS_VERSION
    );
}

add_action( 'admin_enqueue_scripts', 'sp_load_admin_scripts'
);

function sp_load_frontend_scripts() {
    wp_enqueue_script(
        'schoolpress-plugin-frontend',
        plugins_url( 'js/frontend.js', __FILE__ ),
        array( 'jquery' ),
        SCHOOLPRESS_VERSION
    );
}

add_action( 'wp_enqueue_scripts', 'sp_load_frontend_scripts'
);
?>
```

As with stylesheets, it can be difficult to determine whether some bit of JavaScript should be included in the plugin's JavaScript file or the theme's JavaScript file. In general, JavaScript files that support the theme (e.g., slider effects and menu effects) should go in the theme, and JavaScript files that support the plugin (e.g., Ajax code) goes in the plugin. In practice, however, you will find your plugin using JavaScript defined in your theme, and vice versa.

## NOTE

You could use the same callback function to enqueue both your JavaScript files and CSS stylesheets. You can use both the `wp_enqueue_script()` and `wp_enqueue_style()` functions during the `wp_enqueue_scripts` and `admin_enqueue_scripts` hooks. There is no `wp_enqueue_styles` or `admin_enqueue_styles` hook.

## /images/

Place any images needed by your plugin in the `/images/` directory.

## /includes/

The `/includes/` directory is a kind of catchall for the `.php` files your plugin needs. The only `.php` file in your plugin's root folder should be the main plugin file `schoolpress.php`. All other `.php` files should go in one of the other folders; if none are more appropriate, either make another folder or place it in the `/includes/` folder.

It is standard procedure to add a `functions.php`, `common.php`, or `helpers.php` file to include any helper PHP code used by your plugin. This file should include any small scripts that don't have a central role in the logic or functionality of your plugin but are needed to support it. Examples include functions to trim text, functions to generate random strings, or other framework-like functions that aren't already available through a core WordPress function.

## /includes/lib/

Place the third-party libraries needed for your app into the `/includes/lib/` directory.

## **/pages/**

Place any `.php` code related to frontend pages added by your plugin in the `/pages/` directory. Frontend pages are typically added through shortcodes that you would embed into a standard WordPress page to show the content you want.

The following code snippet illustrates how to create a shortcode that can be placed on a WordPress page to generate a page from your plugin. The preheader here is a chunk of code to run before the `wp_head()` function loads, and thus before any HTML headers or code are sent to the browser. The shortcode function further down outputs HTML to the actual page at the place of the shortcode.

Place this code in `/plugins/<your plugin folder>/pages/stub.php` and then include it (typically using the `require_once()` function) from your main plugin file. Then, add the shortcode `[sp_stub]` to a page of your WordPress site:

```
<?php
// preheader
function sp_stub_preheader() {
    if ( !is_admin() ) {
        global $post, $current_user;
        if ( !empty( $post->post_content ) && strpos
            ( $post->post_content, '[sp_stub]' ) !==
        false ) {
            /*
                Put your preheader code
            here.
        }
    }
}
```

```

        */
    }

}

add_action( 'wp', 'sp_stub_preheader', 1 );

// shortcode [sp_stub]
function sp_stub_shortcode() {
    ob_start();
?>
Place your HTML/etc code here.
<?php
$temp_content = ob_get_contents();
ob_end_clean();
return $temp_content;
}
add_shortcode( 'sp_stub', 'sp_stub_shortcode' );
?>
```

For the preheader code, we first check that the page is being loaded from outside the admin using `!is_admin()`; otherwise, this code might run when we're editing the post in the dashboard. Then we look for the string `[sp_stub]` in the content of the `$post` global. This function is hooked to the `wp` hook, which runs after WordPress sets up the `$post` global for the current page, but before any headers or HTML are output.

The preheader code can be used to check permissions, process form submissions, or prep any code needed for the page. In an MVC model, this would be your model and/or controller code. Because this code is run before any headers are output, you can still safely redirect users to another page. For example, you can `wp_redirect()` them to the login or signup page if they don't have access to view the page.

In the shortcode function, we use `ob_start()`, `ob_get_contents()`, and `ob_end_clean()`, which are PHP functions used to buffer output to a variable. Using this code means that the code between the preceding `?>` and `<?php` tags is placed into the `$temp_content` variable instead of output at the time of processing (which would have it echoed out above the `<html>` tag of your site). This isn't necessary: you could just define a `$temp_content()` function and use PHP to add to that string. Using output buffering allows us to code in a more template-like way, mixing HTML and PHP, which is easier to read.

## **/services/**

Place any `.php` code for Ajax calls in the `/services/` directory.

## **/scheduled/**

Place any `.php` code that is related to cron jobs or code that is meant to be run at scheduled intervals here.

## **/schoolpress.php**

This is the main plugin file. For small plugins, this may be the only file needed. For large plugins, the main plugin file will contain only `include` statements, constant definitions, and some comments about which other files contain the code you might be looking for.

# **Add-Ons to Existing Plugins**

A plugin or piece of code that runs on WordPress and is distributed<sup>1</sup> is supposed to be open source and licensed under the GPL. You could take any plugin in the repository, change the name, and release it as a totally new plugin. Doing this could get you into a bar fight, so we suggest that you don't "fork" plugins like this unless you are also planning to improve on and maintain the new plugin.

What if you found a plugin that does 95% of what you need, but it needs a couple lines of code to get to 100%? Consider making an *add-on* for the plugin.

Most well-developed plugins will have their own hooks and filters, which can allow other developers to create an add-on plugin. Just as you would build a plugin to use hooks and filters in WordPress, you can build a plugin to use hooks and filters in other plugins. In some cases, you may need to hack the original plugin to do what you want, which is totally cool, but maybe you can suggest adding some hooks or filters where you need them to the author of the original plugin instead.

## Use Cases and Examples

So what should we build with the free and premium plugins we just mentioned? Let's add a community around WordPress: *SchoolPress*.

Each teacher will be the administrator of their own group and can easily add students to it. Students can engage in the group activity, or the "Class Wall," as we will call it. With *BuddyPress*, students can

add one another as friends, follow their friends or teachers, and privately message their teachers if they have questions.

With *BadgeOS* and the *BadgeOS* Community add-on, teachers can create fun reward badges for their students to earn as they complete various homework assignments and projects that they can share with friends on their other social networks.

And we can use *Gravity Forms* to create an easy way for students to submit their homework.

## The WordPress Loop

The great and powerful *WordPress loop* is what makes WordPress display its posts. Depending on what theme template file is being called on when navigating your website, WordPress queries the database and retrieves the posts that need to be returned to the end user and then loops through them.

Most correctly built WordPress themes usually have the following files that contain the WordPress loop:

- *index.php*
- *archive.php*
- *category.php*
- *tag.php*
- *single.php*
- *page.php*

If you open any of these files, they'll contain code that may look something like this:

```
<?php
if ( have_posts() ) {
    while ( have_posts() ) {
        the_post();
        the_title( '<h2>', '</h2>' );
        the_content();
    }
} else {
    // show a message like sorry no posts!
}
?>
```

The `have_posts()` function checks to see whether there are posts that need to be looped, and if so, the `while` loop is initiated. The `the_post()` function called first in each iteration of the loop sets up the post with all of its global variables so post-specific data can be displayed to the end user.

## WordPress Global Variables

Global variables are variables you can define and then use anywhere thereafter in the rest of your code. WordPress has a few built-in global variables that can really help you save a lot of time and resources when writing code.

To show a full list of every global variable available to you, run the following code:

```
<?php
echo '<pre>';
print_r( $GLOBALS );
```

```
echo '</pre>';
?>
```

To access a global variable in any custom code you are writing, use code like this:

```
<?php
global $global_variable_name;
?>
```

Some global variables are only made available to you depending on where you are in WordPress. Here is a short list of some of the more popular global variables:

*\$post*

An object that contains all post data from the `wp_posts` table for the current post that you are on within the WordPress loop.

*\$authordata*

An object with all author data of the current post that you are on within the WordPress loop.

## **\$WPDB**

The `$wpdb` class is used to interact with the database directly. Once it's globalized, you can use `$wpdb` in custom functionality to select, update, insert, and delete database records. If you are new to WordPress and aren't familiar with all of the functions to push and pull from the database, `$wpdb` is going to be your best friend.

Queries using `$wpdb` are also useful when you need to manage custom tables required by your app or perform a complicated query

(perhaps joining many tables) faster than the core WordPress functions would run on their own. Please don't assume that the built-in WordPress functions for querying the database are slow. Unless you know exactly what you are doing, you'll want to use the built-in functions for getting posts, users, and metadata. The WordPress core is smart about optimizing queries and caching the results from these calls, which will work well across all of the plugins you are running. However, in certain situations, you can shave a bit of time by rolling your own query. We explore a few examples like this in [Chapter 14](#).

## USING CUSTOM DATABASE TABLES

In SchoolPress, we store the relationship of student submissions to assignments in a custom table. This keeps the core WordPress tables a bit cleaner<sup>2</sup> and allows us to easily query for things like “select all of Isaac’s assignments.”

To add our table to the database, we need to write up the SQL for the `CREATE TABLE` command and query it against the WordPress database. You can use either the `$wpdb->query()` method or the `dbDelta()` function in the WordPress core.

We need to do a few things to keep track of our custom tables. We want to store a `db_version` for our app plugin so we know what version of the database schema we are working with in case it updates between versions. We can also check the version, so we run the setup SQL only once for each version. Another common practice is to store your custom table name as a property of `$wpdb` to make querying it a bit easier later.

Example 3-1 shows a little bit of our database setup function for the SchoolPress app.

*Example 3-1. Database setup for SchoolPress*

---

```
<?php
// set up the database for the SchoolPress app
function sp_setupDB() {
    global $wpdb;

    // shortcuts for SchoolPress DB tables
    $wpdb->schoolpress_assignment_submissions = $wpdb-
>prefix .
        'schoolpress_assignment_submissions';

    $db_version = get_option( 'sp_db_version', 0 );

    // create tables on new installs
    if ( empty( $db_version ) ) {
        global $wpdb;

        $sqlQuery = "
CREATE TABLE '" . $wpdb-
>schoolpress_assignment_submissions . "' (
            `assignment_id` bigint(11) unsigned NOT
NULL,
            `submission_id` bigint(11) unsigned NOT
NULL,
            UNIQUE KEY `assignment_submission`(
                `assignment_id`, `submission_id`),
            UNIQUE KEY `submission_assignment`(
                `submission_id`, `assignment_id`)
        )
        ";

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

        $db_version = '1.0';
        update_option( 'sp_db_version', '1.0' );
    }
}
```

```
add_action( 'init', 'sp_dbSetup', 0 );
?>
```

As the `sp_dbSetup()` function is run early in `init` (priority 0), the table shortcuts are available to any other code you have running. You can't always assume a `wp_` prefix, so the `$wpdb->prefix` property is used to get the database prefix for the WordPress install.

A database version for the SchoolPress app is stored in the WordPress options table. We get the value out of options, and if it is empty, we run code to create our custom tables. The `CREATE TABLE` SQL statement here is pretty standard. To make sure they work, always try to run these commands directly on the MySQL database before pasting them into your plugin code.

We use the `dbDelta()` function to create the database table. This function creates a new table if it doesn't exist. Or if a table with the same name already exists, it figures out the correct `ALTER TABLE` query to make the old table match the new schema.

To use `dbDelta()`, you must be sure to include the `wp-admin/includes/upgrade.php` file since that file is loaded only when needed. Then pass `dbDelta()` the SQL for a `CREATE TABLE` query. Your SQL must be in a specific format that's a little stricter than the general MySQL format (taken from the [WordPress Codex on Creating Tables with Plugins](#)):

1. Put each field on its own line in your SQL statement.
2. Leave two spaces between the words `PRIMARY KEY` and the definition of your primary key.

3. Use the keyword KEY rather than its synonym INDEX, and include at least one KEY.
4. Do not use any apostrophes or backticks around field names.

## RUNNING QUERIES

Using `dbDelta()` is preferred when creating tables because it will automatically update older versions of your tables, but you can also run the `CREATE TABLE` query using `$wpdb->query($sqlQuery);`.

You can run any valid SQL statement using the `$wpdb->query()` method. The `query()` method sets a lot of properties on the `$wpdb` object that are useful for debugging or just keeping track of your queries:

`$wpdb->result`

Contains the raw result from your SQL query.

`$wpdb->num_queries`

Increments each time a query is run.

`$wpdb->last_query`

Contains the last SQL query run.

`$wpdb->last_error`

Contains a string with the last SQL error generated if there was one.

`$wpdb->insert_id`

Contains the ID created from the last successful `INSERT` query.

`$wpdb->rows_affected`

Set to the number of affected rows.

`$wpdb->num_rows`

Set to the number of rows in a result for a SELECT query.

`$wpdb->last_result`

Contains an array of row objects generated through the `mysql_fetch_object()` PHP function.

The return value of the `$wpdb->query()` method is based on the top-of-query run and whether the query was successful:

- *Returns false* if the query failed. You can test for this by using code like `if ($wpdb->query($query) === false) { wp_die("it failed!"); }.`
- The raw MySQL result is returned on CREATE, ALTER, TRUNCATE, and DROP queries.
- The number of rows affected is returned for INSERT, UPDATE, DELETE, and REPLACE queries.
- The number of rows affected is returned for SELECT queries.

## ESCAPING IN DATABASE QUERIES

Noted that values passed into the `query()` method are not escaped automatically. Therefore, you will always need to escape untrusted input when using the `query()` method directly.

There are two main ways of escaping values used in your SQL queries: you can wrap your variables in the `esc_sql()` function

(see [Example 3-2](#)) or you can use the `$wpdb->prepare()` method to build your query.

*Example 3-2. Using the esc\_sql() function*

---

```
global $wpdb;
$user_query = $_REQUEST['uq'];

$sqlQuery = "SELECT user_login FROM $wpdb->users WHERE
user_login LIKE '%' . esc_sql($user_query) . '%' OR
user_email LIKE '%' . esc_sql($user_query) . '%' OR
display_name LIKE '%' . esc_sql($user_query) . '%'
";
$user_logins = $wpdb->get_col($sqlQuery);

if (!empty($user_logins))
{
    echo "<ul>";
    foreach ($user_logins as $user_login)
    {
        echo "<li>$user_login</li>";
    }
    echo "</ul>";
}
```

Alternatively, you could create the query using the `prepare()` method, which functions similarly to the `sprintf()` and `printf()` functions in PHP. Using `prepare()` is preferred whenever possible because it does a better job of escaping data and also helps to avoid issues around misplaced single quotes.

The `prepare()` method of the `$wpdb` class located in `wp-includes/wp-db.php` accepts two or more parameters:

`$query`

A string of your custom SQL statement with placeholders for each dynamic value.

`$args`

One or more additional parameters to be used to replace the placeholders in your SQL statement.

The following directives can be used in the SQL statement string:

- `%d` (integer)
- `%f` (float)
- `%s` (string)
- `%%` (literal percentage sign—no argument needed)

The directives `%d`, `%f`, and `%s` should be left unquoted in the SQL statement, and each placeholder used must have a corresponding argument passed in for it. Literals `(%)` as part of the query must be properly written as `%%`:

```
$sqlQuery = $wpdb->prepare("SELECT user_login FROM $wpdb-
>users WHERE
user_login LIKE %%$% OR
user_email LIKE %%$% OR
display_name LIKE %%$%", $user_query, $user_query,
$user_query);
$user_logins = $wpdb->get_col($sqlQuery);
```

## NOTE

If you use `$wpdb->prepare()` without including the `$args` parameter, you will get a PHP warning message: Missing argument 2 for `wpdb::prepare()`. If your SQL doesn't use any placeholder values, you don't need to use `prepare()`.

Holy percent sign, Batman! The % is used in SQL as a wildcard in SELECT statements when using the LIKE keyword. So if you search for `user_login LIKE %coleman%`, it returned users with user logins like “jcoleman,” “jasoncoleman,” and “coleman1982.” To keep these *literal* % signs in place with the `prepare()` method, we need to double them up to %% , which is translated into just one % in the final query.

The other % in there is used with %s. This is the placeholder where our `$user_query` parameter is going to be swapped in after being escaped.

You may have noticed we used the `$wpdb->get_col()` method in the previous code segment. WordPress offers many useful methods on the `$wpdb` object to SELECTs, INSERTs, and other common queries in MySQL.

## SELECT QUERIES WITH \$WPDB

The WordPress `$wpdb` object has a few useful methods for selecting arrays, objects, rows, columns, or even single values out of the MySQL database using SQL queries.

`$wpdb->get_results($query, $output_type)` will run your query and return the `last_results` array, including all of the rows from your SQL query in the output type specified. By default, the result will be a “numerically indexed array of row objects.” Here’s the full list of output types from the WordPress Codex:

*OBJECT*

Result will be output as a numerically indexed array of row objects.

*OBJECT\_K*

Result will be output as an associative array of row objects, using the first column’s values as keys (duplicates will be discarded).

*ARRAY\_A*

Result will be output as an numerically indexed array of associative arrays, using column names as keys.

*ARRAY\_N*

Result will be output as a numerically indexed array of numerically indexed arrays.

The following code helps show how to use the array returned by `$wpdb->get_results()` when using the *OBJECT* output type:

```
<?php
global $wpdb;
$sqlQuery = "SELECT * FROM $wpdb->posts
    WHERE post_type = 'assignment'
        AND post_status = 'publish' LIMIT 10";
$assignments = $wpdb->get_results( $sqlQuery );

// rows are stored in an array, use foreach to loop through them
```

```

foreach ( $assignments as $assignment ) {
    // each item is an object with property names equal to the
    // SQL column names?
    <h3><?php echo $assignment->post_title;?></h3>
    <?php echo apply_filters( "the_content", $assignment-
        >post_content );?>
    <?php
}
?>

```

`$wpdb->get_col($query, $column_offset = 0)` will return an array of the values in the first column of the MySQL results. The `$column_offset` parameter can be used to grab other columns from the results (0 is the first, 1 is the second, and so on).

This function is most commonly used to grab IDs from a database table to be used in another function call or database query:

```

<?php
global $wpdb;
$sqlQuery = "SELECT ID FROM $wpdb->posts
    WHERE post_type = 'assignment'
    AND post_status = 'publish'
    LIMIT 10";
// getting IDs
$assignment_ids = $wpdb->get_col( $sqlQuery );

// result is an array, loop through them
foreach ( $assignment_ids as $assignment_id ) {
    // we have the id, we can use get_post to get more
    // data
    $assignment = get_post( $assignment_id );
    ?>
    <h3><?php echo $assignment->post_title;?></h3>
    <?php echo apply_filters( "the_content",
        $assignment->post_content );?>
    <?php

```

```
}
```

```
?>
```

Note that we're putting that `global $wpdb;` line in most of our examples here to reinforce the point that you need to make sure that `$wpdb` is in scope before calling one of its methods. In practice, this line is usually at the top of the function or file you are working within.

We use `$wpdb->get_row($query, $output_type, $row_offset)` to get just one row from a result. Instead of getting an array of results, you get just the first object (or array, if the `$output_type` is specified) from the result set.

You can use the `$row_offset` parameter to grab a different row from the results (0 is the first, 1 is the second, and so on).

### *Insert, replace, and update*

You can use `$wpdb->insert($table, $data, $format)` to insert data into the database. Rather than building your own `INSERT` query, you simply pass the table name and an associative array containing the row data and WordPress will build the query and escape it for you. The keys of your `$data` array must map to column names in the table. The values in the array are the values to insert into the table row:

```
<?php
// processing new submissions for assignments
global $wpdb, $current_user;

// create submission
```

```

$assignment_id = intval( $_REQUEST['assignment_id'] );
$submission_id = wp_insert_post(
    array(
        'post_type'      => 'submission',
        'post_author'   => $current_user->ID,
        'post_title'     => sanitize_title(
            $_REQUEST['title'] ),
        'post_content'  => sanitize_text_field(
            $_POST['submission'] )
    )
);

// connect the submission to the assignment
$wpdb->insert(
    $wpdb->schoolpress_assignment_submissions,
    array( "assignment_id"=>$assignment_id,
    "submission_id"=>$submission_id ),
    array( '%d', '%d' )
);

/*
    This insert call will generate a SQL query like:
    INSERT INTO
        'wp_schoolpress_assignment_submissions'
        ('assignment_id', 'submission_id' VALUES (101,10)
*/
?>

```

In the previous code, we use `wp_insert_post()` to create the submission and then use `$wpdb->insert()` to insert a row into our custom table connecting assignments with submissions.

We pass an array of formats to the third parameter to tell the method to format the data as integers when constructing the SQL query. The available formats are `%s` for strings, `%d` for integers, and `%f` for floats. If no format is specified, all data will be formatted as a string.

In most cases, MySQL will properly cast your string into the format needed to store it in the actual table.

To relate two posts like this, we could also simply put the assignment\_id into the post\_parent column of the wp\_posts table. This is adequate to create a parent/child relationship. However, if you want to do a many-to-many relationship (e.g., if you can post the same submission to multiple assignments), you need a separate table or some other way to connect a post to many other posts.

`$wpdb->replace($table, $data, $format)` is similar to the `$wpdb->insert()` method. The `$wpdb->replace()` method will literally generate the same SQL query as `$wpdb->insert()`, but uses the MySQL REPLACE command instead of INSERT, overriding any row with the same keys as the `$data` passed in.

You can use `$wpdb->update($table, $data, $where, $format = null, $where_format = null)` to update rows in a database table. Rather than building your own UPDATE query, simply pass the table and an associative array containing the updated columns and new data along with an associative array `$where` containing the fields to check against in the WHERE clause, and WordPress will build the query and escape the UPDATE query for you.

The \$where and \$where\_format parameters work the same as the \$data and \$format arrays, respectively.

The WHERE clause generated by the update () method will check that the columns are equal to the values passed and those checks are combined together by AND conditions.

The update () method is particularly useful in that you can update any number of fields in a table row using the same function. Here is some code that could be used to update orders in an ecommerce plugin:

```
<?php
global $wpdb;
// just update the status
$wpdb->update(
    'ecommerce_orders', //table name
    array( 'status' => 'paid' ), //data fields
    array( 'id' => $order_id ) //where fields
);

// update more data about the order
$wpdb->update(
    'ecommerce_orders', //table name
    array( 'status' => 'pending', //data fields
        'subtotal' => '100.00',
        'tax' => '6.00',
        'total' => '106.00'
    ),
    array( 'id' => $order_id ) //where fields
);
?>
```

\$wp\_query

An object of the `WP_Query` class that can show you all of the post content returned by WordPress for any given page that you are on. We will talk more about the `WP_Query` class and its methods in the next chapter.

### `$current_user`

An object of all of the data associated with the currently logged-in user. Not only does this object return all of the data for the current user from the `wp_users` table, but it will also give you the roles and capabilities of the current user:

```
<?php
//welcome the logged-in user
global $current_user;
if ( !empty( $current_user->ID ) ) {
    echo 'Howdy, ' . $current_user->display_name;
}
?>
```

When writing your code to run on WordPress, you can define and use your own global variables if it makes sense. These can save you the hassle of rewriting code and recalling functions because once they are defined, you can use them again and again.

## Free Plugins

There are some useful free plugins that can help extend your web application. Plugins exist for almost every purpose, but if you can't find the exact functionality you're looking for, you could modify an existing plugin (open source, right?) or create an entirely new one if you are up for the challenge.

## Admin Columns

[Admin Columns](#) is a very useful plugin and easily one of our favorites for managing what columns are displayed on the WordPress posts, users, comments, media, and any registered custom post type admin listing pages. This plugin also supports adding custom fields, featured images, and custom taxonomies to your admin columns. This plugin's super-useful features also include the ability to add sortable columns, editable fields, and exportable CSV files with the columns you have designated. The pro version of this plugin also integrates with Advanced Custom Fields, WooCommerce, and Yoast SEO, allowing you to add any meta fields from those plugins to your admin columns. One of our favorite use cases for this plugin is being able to bulk-edit SEO values for all of my posts on the listing page instead of on each individual post; talk about a time saver!

## **Advanced Custom Fields**

[Advanced Custom Fields \(ACF\)](#) allows administrators to easily set up available custom fields for any post type. It has a very slick admin UI for establishing the fields and the types of fields that are available for a given post type, and backend users can interact with established fields while adding or updating a post. This plugin is also geared towards developers, with lots of custom hooks and filters built in, as well as custom functions you can easily use to display data on the front end. ACF also has a bunch of pro add-on plugins, including:

### *Repeater field*

Allows you to create a set of subfields which can be repeated over and over again, as a way of adding a list of values.

### *Gallery field*

Provides a nice UI for managing multiple images.

#### *Options page*

Provides functions for adding extra admin pages to update ACF fields.

#### *Flexible content field*

Creates custom layouts with a content-layout UI.

## **BadgeOS**

The BadgeOS plugin can transform any website into a platform for rewarding members, achievements based on their activities. The site administrator can use it to create different achievement types and award the members shareable badges once they complete all the stated requirements for that level. Badges are Mozilla OBI compatible and shareable via *Credly.com*.

Another very cool gamification plugin for WordPress is GamiPress, which is actually a fork of BadgeOS. In our opinion it's better maintained and has more features.

## **Posts 2 Posts**

Posts 2 Posts (P2P) is another powerful plugin for building web apps. You can use it to create many-to-many relationships between posts, pages, and CPTs as well as many-to-many relationships between posts and users.

For an example, you could use P2P to make connections between CPTs for schools, teachers, and subjects. A school could have

multiple teachers, and each teacher could be tied to one or more subjects.

P2P provides intuitive settings, feature-rich widgets, and an easy-to-use meta box attached to any post add/edit page for making new connections.

Most of the time, custom plugin developers should avoid creating additional database tables unless it absolutely makes sense. If we wanted to connect posts to other posts, we could store an array of post IDs in a custom field of another post, but this can become inefficient in a large-scale application. P2P creates its own database tables, `wp_p2p` and `wp_p2pmeta`, for storing the relationships between posts more efficiently; see Tables 3-1 and 3-2, respectively.

*Table 3-1. Database schema for wp\_p2p table*

Column	Type	Collation	Nu ll	Defau lt	Extra
p2p_id	bigint(20)		No	None	AUTO_INCREMENT
p2p_from	bigint(20)		No	None	
p2p_to	bigint(20)		No	None	
p2p_type	varchar(44)	utf8_general_ci	No		

*Table 3-2. Database schema for wp\_p2pmeta table*

Column	Type	Collation	Null	Default	Extra
meta_id	bigint(20)		No	None	AUTO_INCREMENT
p2p_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_val	longtext	utf8_general_ci	Yes	NULL	

For more information on this plugin, make sure to check out the [wiki](#) on GitHub.

## Members

The [Members](#) plugin extends the control that you have over user roles and capabilities in your site. It enables you to edit as well as create and delete user roles and capabilities. This plugin also allows you to set permissions for different user roles to determine which roles are able to add, edit, and/or delete various pieces of content. While it's possible to apply the information in [Chapter 6](#) to use custom code to change roles and permissions, using a plugin like Members can speed this up for some projects and is absolutely necessary if you need nonprogrammers to adjust these rules at a later time.

The [User Role Editor](#) plugin is also popular for managing roles and permissions through a GUI.

## **W3 Total Cache**

Caching your content is a great idea for optimizing the performance of your website. You can save a lot of processing time by displaying cached pages to the end user instead of querying the database every time someone requests data. W3 Total Cache has a lot of built-in features for managing what content gets cached and when the cache should be cleared. More information on using W3 Total Cache can be found in Chapter 14.

## **Yoast SEO**

Yoast SEO is a great plugin to use if you are concerned about search engine optimization (SEO). Once installed, this plugin automatically optimizes your site for search engines. It also adds a preview of the snippet shown in search engine results with fields to override the title and description there. Another useful feature of the Yoast SEO plugin is the “Readability Analysis” report, with suggestions for making your writing easier to understand.

A paid version of Yoast SEO is also available, with advanced features and premium support.

Another popular plugin to address SEO is All in One SEO Pack.

## **Premium Plugins**

Although there are a lot of great free plugins out there, some premium plugins are definitely worth the money. These plugins are usually available for purchase for one-time use, and some also offer

developer licenses that allow you to purchase the plugin for installation on multiple WordPress sites.

## **Gravity Forms**

An absolute must, the [Gravity Forms plugin](#) enables you to easily create custom contact forms for your site. It is an extremely simple process to create a form using the visual form editor, which allows you to drag and drop the fields into the form and reposition them as needed. Standard fields are included as well as the option to create custom fields. The forms are very flexible and can be set up as multiple-page forms with progress bars. Conditional fields allow you to show or hide fields based on selections in previous fields. Another great feature of this plugin is the ability for the forms, once completed, to be forwarded anywhere as chosen by the site admin in the form settings. All in all, this plugin is useful and flexible for anyone needing to create a form on their site, and easy to use for someone without coding knowledge.

## **BackupBuddy**

The [BackupBuddy plugin](#) provides you with the opportunity to back up your entire WordPress install for safekeeping, restoring, or moving your site. Backups can be scheduled on a recurring basis, and the file can then be downloaded to your computer, emailed to you, or sent off to the storage location of your choice, such as Dropbox or an FTP server. This plugin also features a restore option that will easily restore your themes, widgets, and plugins. The plugin also allows you to easily move your site to a new server or domain directly from the WordPress dashboard, which comes in handy if you work on a Dev

server and then move the sites over to a Production environment upon launch.

## WP All Import

The [WP All Import plugin](#) comes in handy if you are looking to import data into WordPress from another source that is in either an XML or CSV file, which are two formats not routinely accepted by WordPress. There is also a pro or premium version of the plugin available for purchase that extends the functionality to allow you to import data into CPTs as well as custom fields. The pro version also enables you to import images from a URL and save them in the media library. Also helpful is the ability to set up recurring imports to periodically check a file for changes or updates and then modify the corresponding post as needed.

## Community Plugins

You can build a full-blown social network with WordPress and a few free plugins. Social networks are great to bring a niche community together. If you have an active social network, you will have lots of organic content being indexed by search engines. If you think you get a lot of comments and interaction on your existing WordPress website, try turning it into a social network to really get the conversions flowing.

## BuddyPress

[BuddyPress](#) is social networking in a box. You can start up a social network with most of the same features as Facebook in a matter of

minutes.

You can download BuddyPress from the plugin repository as you would other plugins, or from the BuddyPress site. This plugin has come a long way since version 1.0 was released in April 2009. First built by Andy Peatling, it originally only worked on WordPress MU (Multi User). Automattic, seeing the potential of a plugin that turns WordPress into a social networking application, started funding the project.

Since version 1.7, BuddyPress has been theme agnostic, meaning that if coded properly, it will work with any theme—well, most themes. Prior to version 1.7 (see [Figure 3-2](#)), to properly use the plugin, you had to use a BuddyPress theme. This worked well for people wanting to build a social network from scratch because they could use the included default theme, purchase a nice premium BuddyPress child theme, or plan to build their own BuddyPress child theme. However, it was limiting for those who already had a WordPress website because they couldn't just turn on BuddyPress and have it work with their existing theme. In most cases, people with existing sites that wanted to turn on BuddyPress needed to do customization, which is fine for those who know CSS, PHP, and how WordPress works. But noncoders would need to hire someone to turn their existing theme (which they may have paid for) into a BuddyPress child or compatible theme. But with newer versions of BuddyPress, it just works!

People with existing websites can now use BuddyPress and its features and have them work in their existing theme. It is also very

easy to override any existing style to tailor the BuddyPress features more to your website. Special thanks to the more recent core contributors, John Jacoby, Boone Gorges, Paul Gibbs, and todo: (real name Ray Ho) for making BuddyPress what it is today, a theme-independent plugin that turns WordPress into a social network.



[What's new?](#) [Changelog](#) [Get involved](#)

## Introducing the BP REST API



BuddyPress 5.0.0 comes with REST API endpoints for members, groups, activities, users, private messages, screen notifications and extended profiles.

BuddyPress endpoints provide machine-readable external access to your WordPress site with a clear, standards-driven interface, paving the way for new and innovative methods of interacting with your community through plugins, themes, apps, and beyond. Ready to get started with development?

Check out the [BP REST API reference](#).

## A new interface for managing group members.

The best way to show the power of the BP REST API is to start using it for one of our Core features.

*Figure 3-2. Welcome to BuddyPress*

## DATABASE TABLES

Unlike a lot of WordPress plugins, BuddyPress creates its own database tables in MySQL. If the original BuddyPress developers were to rewrite the plugin from scratch today, they would probably store activities and notifications as custom posts instead of using custom tables. However, CPTs weren't implemented when the original version of BuddyPress was released, and it would take a lot of effort to change that architecture now. The custom tables that store groups and friend relationships between users are much easier to understand and faster to query against than if these kinds of things were stored as some combination of posts, user meta, and taxonomies.

For smaller distributed plugins, it makes sense to avoid custom tables whenever possible because it means there is less overhead for users of the plugin to worry about. However, for plugins specific to your app or plugins that include as much functionality as BuddyPress, custom tables can help to speed up or better organize your data.

We've included the schema for each BuddyPress table here (Tables 3-3 through 3-18) as an example of how you might go about structuring custom tables for your own apps and also to help you understand how BuddyPress data is stored in case you would like to query for that information directly.

*Table 3-3. Database schema for wp\_bp\_activity table*

Column	Type	Collation	N ul l	Defa ult	Extra
id	bigint(20)		N o	None	AUTO_INCREMENT
user_id	bigint(20)		N o	None	
component	varchar(75)	utf8_general_ci	N o	None	
type	varchar(75)	utf8_general_ci	N o	None	
action	text	utf8_general_ci	N o	None	
content	longtext	utf8_general_ci	N o	None	
primary_link	varchar(255)	utf8_general_ci	N o	None	
item_id	bigint(20)		N o	None	
secondary_item_id	bigint(20)		Ye s	NUL L	
date_recorded	datetime		N o	None	
hide_sitewide	tinyint(1)		Ye s	0	
mptt_left	int(11)		N o	0	

Column	Type	Collation	N ul l	Defa ult	Extra
mptt_right	int(11)		N o	0	
is_spam	tinyint (1)		N o	0	

Table 3-4. Database schema for wp\_bp\_activity\_meta table

Column	Type	Collation	Nu ll	Defa ult	Extra
id	bigint(2 0)		No	None	AUTO_INCREMENT
activity_ _id	bigint(2 0)		No	None	
meta_key	varchar(2 55)	utf8 genera l_ci	Ye s	NUL L	
meta_val ue	longtext	utf8 genera l_ci	Ye s	NUL L	

*Table 3-5. Database schema for wp\_bp\_friends table*

Column	Type	Collation	Null	Default	Extra
id	bigint (20)		No	None	AUTO_INCREMENT
initiator_use_r_id	bigint (20)		No	None	
friend_user_id	bigint (20)		No	None	
is_confirmed	tinyint (1)		Yes	0	
is_limited	tinyint (1)		Yes	0	
date_created	datetime		No	None	

*Table 3-6. Database schema for wp\_bp\_groups table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
creator_id	bigint(20)		No	None	
name	varchar(100)	utf8_general_ci	No	None	
slug	varchar(200)	utf8_general_ci	No	None	
description	longtext	utf8_general_ci	No	None	
status	varchar(100)	utf8_general_ci	No	Public	
enable_forum	tinyint(1)		No	1	
date_created	datetime		No	None	

*Table 3-7. Database schema for wp\_bp\_groups\_groupmeta table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
group_id	bigint(20)		No	None	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_value	longtext	utf8_general_ci	Yes	NULL	

*Table 3-8. Database schema for wp\_bp\_groups\_members table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
group_id	bigint(20)		No	None	
user_id	bigint(20)		No	None	
inviter_id	bigint(20)		No	None	
is_admin	tinyint(1)		No	0	
is_mod	tinyint(1)		No	0	
user_title	varchar(100)	utf8_general_ci	No	None	
date_modified	datetime		No	None	
comments	longtext	utf8_general_ci	No	None	
is_confirmed	tinyint(1)		No	0	
is_banned	tinyint(1)		No	0	
invite_sent	tinyint(1)		No	0	

*Table 3-9. Database schema for wp\_bp\_messages\_messages table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
thread_id	bigint(20)		No	None	
sender_id	bigint(20)		No	None	
subject	varchar(200)	utf8_general_ci	No	None	
message	longtext	utf8_general_ci	No	None	
date_sent	datetime		No	None	

*Table 3-10. Database schema for wp\_bp\_messages\_notices table*

Column	Type	Collation	Nu ll	Defa ult	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
subject	varchar(200)	utf8_general_ci	No	None	
message	longtext	utf8_general_ci	No	None	
date_sent	datetime		No	None	
is_active	tinyint(1)		No	0	

*Table 3-11. Database schema for wp\_bp\_messages\_recipients table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
user_id	bigint(20)		No	None	
thread_id	bigint(20)		No	None	
unread_count	int(10)		No	0	
sender_only	tinyint(1)		No	0	
is_deleted	tinyint(1)		No	0	

*Table 3-12. Database schema for wp\_bp\_notifications table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		N o	None	AUTO_INCREMENT
user_id	bigint(20)		N o	None	
item_id	bigint(20)		N o	None	
secondary_item_id	bigint(20)		Ye s	NUL L	
component_name	varchar(75)	utf8_general_ci	N o	None	
component_action	varchar(75)	utf8_general_ci	N o	None	
date_notified	datetime		N o	None	
is_new	tinyint(1)		N o	0	

*Table 3-13. Database schema for wp\_bp\_user\_blogs table*

Column	Type	Collation	Nul	Defaul	Extra
		n		t	
id	bigint(20)		No	None	AUTO_INCREMENT
user_id	bigint(20)		No	None	
blog_id	vbigint(20)		No	None	

*Table 3-14. Database schema for wp\_bp\_user\_blogs\_blogmeta table*

Column	Type	Collation	Nu	Defa	Extra
			ll	ult	
id	bigint(20)		No	None	AUTO_INCREMENT
blog_id	bigint(20)		No	None	
meta_key	varchar(255)	utf8_general_ci	Ye	NULL	s
meta_val	longtext	utf8_general_ci	Ye	NULL	s

*Table 3-15. Database schema for wp\_bp\_xprofile\_data table*

Column	Type	Collation	Nu ll	Defa ult	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
field_id	bigint(20)		No	None	
user_id	bigint(20)		No	None	
value	longtext	utf8_general_ci	No	None	
last_updated	datetime		No	None	

Table 3-16. Database schema for wp\_bp\_xprofile\_fields table

Column	Type	Collation	N ul l	Defa ult	Extra
id	bigint(20)		N o	None	AUTO_INCREMENT
group_id	bigint(20)		N o	None	
parent_id	bigint(20)		N o	None	
type	varchar(150)	utf8_general_ci	N o	None	
name	varchar(150)	utf8_general_ci	N o	None	
description	longtext	utf8_general_ci	N o	None	
is_required	tinyint(1)		N o	0	
is_default_option	tinyint(1)		N o	0	
field_order	bigint(20)		N o	0	
option_order	bigint(20)		N o	0	
order_by	varchar(15)	utf8_general_ci	N o		
can_delete	tinyint(1)		N o	1	

*Table 3-17. Database schema for wp\_bp\_xprofile\_groups table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
name	varchar(150)	utf8_general_ci	No	None	
description	mediumtext	utf8_general_ci	No	None	
group_order	bigint(20)		No	0	
can_delete	tinyint(1)		No	None	

*Table 3-18. Database schema for wp\_bp\_xprofile\_meta table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
object_id	bigint(20)		No	None	
object_type	varchar(150)	utf8_general_ci	No	None	
meta_key	varchar(255)	utf8_general_ci	Yes	NUL	L
meta_value	longtext	utf8_general_ci	Yes	NUL	L

## COMPONENTS

After activating BuddyPress, head over to the Components panel (see Figure 3-3) in Settings → BuddyPress, or `/wp-admin/options-general.php?page=bp-components`, to set up components you'd like to use.

## BuddyPress Settings

Components

Options

Pages

Credits

All (10) | Active (6) | Inactive (4) | Must-Use (2) | Retired (0)

<input type="checkbox"/>	Component	Description
<input checked="" type="checkbox"/>	Extended Profiles	Customize your community with fully editable profile fields that allow your users
<input checked="" type="checkbox"/>	Account Settings	Allow your users to modify their account and notification settings directly from v
<input type="checkbox"/>	Friend Connections	Let your users make connections so they can track the activity of others and foc
<input type="checkbox"/>	Private Messaging	Allow your users to talk to each other directly and in private. Not just limited to o number of members.
<input checked="" type="checkbox"/>	Activity Streams	Global, personal, and group activity streams with threaded commenting, direct p email notification support.
<input checked="" type="checkbox"/>	Notifications	Notify members of relevant activity with a toolbar bubble and/or via email, and a
<input type="checkbox"/>	User Groups	Groups allow your users to organize themselves into specific public, private or h listings.
<input type="checkbox"/>	Site Tracking	Record activity for new posts and comments from your site.
<input type="checkbox"/>	BuddyPress Core	It's what makes time travel BuddyPress possible!
<input type="checkbox"/>	Community Members	Everything in a BuddyPress community revolves around its members.

### *Figure 3-3. BuddyPress components*

The following components are available:

#### *Extended Profiles*

Like any typical social network, BuddyPress has member profiles. A member has complete control of their profile. Out of the box, all members are listed in a members directory; when you click on a member's name, you're taken to their profile page.

#### *Account Settings*

Members can update their email address, change their password, and even manage the email notifications they receive when other members interact with them.

#### *Friend Connections*

Members can add one another as friends. Members ask other members to be friends by sending and receiving friend requests. Think Facebook friends.

#### *Private Messaging*

Members send private messages to one another and view their messages in one place, like an inbox for your social network. Members can reply, mark as read, delete, and perform other actions with messages as with any large social network.

#### *Activity Streams*

Members can post activity updates to their profiles and groups, leave comments on other members' or groups' activity, and favorite activity posts. Sounds like Facebook, right? BuddyPress has an @mention feature similar to when someone mentions you on Twitter. @mentions automatically link to the mentioned member's profile page, and if that member has their notifications

turned off, they receive an email about it. Activity also comes standard with RSS feeds.

### *User Groups*

A powerful component of BuddyPress, groups can be created organically (or not) by network members. Each group appears on a Groups listing page, and clicking a group's avatar brings you to its page. The group profile is set up like the member profile page, but with group-specific subpages such as group activity, members, admin settings, and invite friends. Groups can be public, private, or hidden, and members can be promoted to group administrators or group moderators.

### *Site Tracking*

New posts and comments on your site create BuddyPress activity posts. If you are running BuddyPress on a WordPress multisite network, posts and comments created on any site in your network will also create BuddyPress activity posts.

All of these core BuddyPress components can be extended with BuddyPress plugins. It can be a little confusing if you are new to all of this, but you can install additional plugins specific to BuddyPress or build your own. There are approximately 485 WordPress plugins that extend or integrate with BuddyPress in one way or another.

## **PAGES**

Once you have decided which core components you want to use, go to the Pages tab at Settings → BuddyPress → Pages, as shown in Figure 3-4. BuddyPress maps the components it is using to new or existing pages. By default, BuddyPress tries to make a new page for each component. If you wanted to call members “Students” instead of “Members,” you could create a regular WordPress page called

“Students” and map the members component to this new page. The same goes for other BuddyPress components. You’ll need to create two pages for member registration: Register and Activate. You will need to map both of these to pages if you wish to have open registration on your social network.

# BuddyPress Settings

Components

Options

Pages

Credits

## Directories

Associate a WordPress Page with each BuddyPress component directory.

Members

Members



[View](#)

Activity Streams

Activity



[View](#)

## Registration

Registration is currently disabled. Before associating a page is allowed, please enable regis

[Save Settings](#)

*Figure 3-4. BuddyPress pages*

## NOTE

To allow open registration, you also must make sure that anyone can register; select the “Anyone can register” checkbox under Settings → General.

## SETTINGS

In the Settings panel (Settings → BuddyPress → Settings), or `/wp-admin/admin.php?page=bp-settings`, you can configure some additional BuddyPress settings:

### *Toolbar*

By default, BuddyPress shows the WordPress admin bar with Login and Register links for users who aren’t logged in. You can also turn off this feature here.

### *Account Deletion*

You can decide whether you will allow registered users to delete their accounts.

### *Avatar Uploads*

You can allow registered members to upload avatars.

### *Profile Syncing*

You can enable BuddyPress to WordPress profile syncing.

### *Group Creation*

You can allow your registered members to create their own groups. Site admins can still create groups if this setting is turned off.

## *Blog & Forum Comments*

You can allow activity stream commenting on blog and forum posts.

## **PROFILE FIELDS**

Located at Users → Profile Fields, this BuddyPress feature allows you to create any number of profile field groups and profile fields for your members. You can collect data such as location, date of birth, likes, dislikes, favorite color, and/or whatever you want. This feature is very flexible in allowing you to organize your profile fields into different profile groups, all of which will be made available on any member's frontend profile page.

When adding any new profile field, you are provided with a slick UI for deciding whether to make your new field required, what type of form element it should be, what the default visibility is, and whether you want your members to be able to decide if they can change the visibility for the field. This form is shown in [Figure 3-5](#).

### **NOTE**

By default, all of the profile fields in the “Base” profile group will show up on the registration page.



*Figure 3-5. BuddyPress profile fields*

## **BUDDYPRESS PLUGINS**

As you can see, BuddyPress is an intuitive and easy-to-use plugin. We talked briefly about installing additional BuddyPress-specific plugins, and following is a quick list of some cool ones to give you ideas on how you can extend BuddyPress:

### *BuddyPress Media*

This plugin allows your members to upload photos, music, and videos to their activity posts. It also allows for your members to organize all of their photos into photo albums on their profile page. There is mobile device support that includes automatic audio and video conversion.

### *BuddyPress Registration Options*

This is a great plugin for stopping spambots from registering on your BuddyPress website. This plugin allows for new member moderation; if moderation is turned on from the admin settings page, any new members will be blocked from interacting with any BuddyPress components (except editing their own profile and uploading their avatar) and will not be listed in any directory until an administrator approves or denies their account. If moderation is turned on, admins can create custom display messages and email alert messages for approved or denied accounts. When an admin approves or denies, custom emails are sent to new members informing them whether they were approved or denied.

### *AppPresser Add-On*

The AppPresser AppCommunity add-on allows AppPresser users to create a social mobile app for their WordPress website similar to Instagram or Facebook using BuddyPress. With

AppCommunity, app users can post activity updates and photos, create friendships, send private messages, join groups, and more.

### BadgeOS Community Add-On

The BadgeOS Community Add-On integrates BadgeOS features into BuddyPress and bbPress. Site members complete achievements and earn badges based on a range of community activities and triggers. This add-on to BadgeOS also includes the ability to display badges and achievements on user profiles and activity feeds.

### bbPress

Got forums? bbPress can fulfill all your forum needs. Unlike BuddyPress, bbPress utilizes custom post types, so it does not create its own tables in the database like it used to in prior versions.

Using bbPress can require a bit of theme work if your theme isn't already styled to support bbPress, but it is by far the easiest way to add forum functionality to a WordPress site.

- 
- 1 In the context of the GPL, distribution means selling your source code or offering it for download on a website like the *WordPress.org* plugin repository. Code that you personally install for someone does not need to inherit the GPL license.
  - 2 A synchronized entry in both the `wp_usermeta` and `wp_postmeta` tables would be needed to provide the same lookup ability of a single `wp_schoolpress_assignment_submissions` table.

# Chapter 4. Themes

---

WordPress themes drive the frontend of your web app. In [Chapter 1](#), we presented the analogy that WordPress themes are like views in a traditional MVC framework. The analogy isn't perfect, but themes and views are similar in that they both control the way your app will look and are where your designers will spend most of their time.

The [Theme Developer Handbook](#) put together by the WordPress community is the definitive source for learning how to build themes for WordPress in a standards-based way. All theme developers should use that resource. This chapter covers areas of theme development especially important to app developers.

## Themes Versus Plugins

At some level, all source files in your themes and plugins are just *.php* files loaded at different times by WordPress. In theory, your entire app code could reside in one theme or one plugin. In practice, you'll want to reserve your theme for code related to the frontend (views) of your website and use plugins for your app's backend (models and controllers).

Where you decide to put some code will depend on whether you are primarily building a full app or an individual plugin or theme.

## Where to Place Code When Developing Apps

If you are building a full web app—basically one WordPress installation—you will have full access to the site and what themes and plugins are installed. Your code could go anywhere. Even so, you should follow *some* thought process when deciding whether a particular feature should be coded as a module of your app's plugin or theme or as a separate plugin. The main benefactor of your good planning at this step will be your developers (maybe just you). Properly organizing your code is going to make it easier for you to maintain your app and develop it further.

When building apps, we try to use the following guidelines:

- Use one main plugin to store the core app code, and one theme to manage the frontend code.
- Any modular functionality that could be useful on other projects or potentially replaced by another plugin should be coded as a separate plugin.
- Never hack the core!<sup>1</sup>

So what is core app code and what is frontend code? Again our pseudo-MVC framework looks like this:

*Plugins = models*

All of your code-defining data structures, business logic, and Ajax services should go into the core plugin. Things like definitions for CPTs and taxonomies, form processing, and class wrappers for the Post and User classes should also go into your core plugin.

*Themes = views*

All of your templating code and frontend logic should go in your theme. The frame of your website, header, footer, menu, and sidebars should be coded in your theme. Simple logic like  
`if(is_user_logged_in()) { //show menu } else { //show login }` should go into your theme.

One thing to consider when deciding where to code features is your development team. If you have a designer and programmer, you should put things the designer will be concerned with in the theme, and things that concern the programmer in the core plugin. Even if you have to finagle a little bit, clearly separating things like that will make it easier for your developers to find what they are looking for.

## **When Developing Plugins**

If you are building a plugin to be used on other websites or modular features that can be used across projects, it makes sense to keep your code within one plugin. In these cases, you can store template files within your plugin to handle the UI components. It is common practice to allow these files to be overwritten by the active WordPress theme, which will be covered later in this chapter.

## **Where to Place Code When Developing Themes**

Similarly, if you are developing a theme that will be distributed, and that relies on CPTs or other customization that would typically be coded in a plugin, it might make sense to instead include that within your theme. If your users must activate a plugin before your theme works at all, you might as well move the plugin code into your theme.

If your theme makes large underlying changes to WordPress, consider putting that plugin-like code into a parent theme and putting your design-related code into a child theme. That way if your users want to change their site's design without losing the other functionality provided by a theme, they can do so more easily.

On the other hand, if the code you are about to add to your theme is not crucial to the theme's workings or there are other plugins that could be used as alternatives for your code, you should move that code into a plugin and distribute your theme as a bundle that includes the themes and recommended plugins. As an example, many premium themes add SEO-related fields to the edit post page to manage page titles, meta descriptions, and meta keywords. This makes sense, since these SEO-related fields represent a kind of view that is seen by Google and other web crawlers. However, there are a few really popular plugins that do this same functionality, and it's hard to argue that your theme wouldn't work without the SEO functionality installed. We would recommend theme developers put their SEO functionality into plugins or otherwise make it easy to disable so other plugins can be used.

In the end, the decision as to where to put what code and how to package things should be based on your users, both end users as well as the developers who will use your themes and plugins. Part of the beauty of WordPress is its flexibility in terms of how you can go about customizing it. There are no strict rules. Consider everything you read about this topic (including from us) as guidelines. If moving code from a plugin file to a theme file makes it easier to work with, do it.

# The Template Hierarchy

When a user visits your site and navigates to a page, WordPress uses a system called the *template hierarchy* to determine which file in the active theme should be used to render the page. For example, if the user browses to a single-post page, WordPress will look for *single-post.php*. If this isn't found, it will look for *single.php*. If that's not found, it will look for *index.php*.

The *index.php* file is the fallback for all page loads, and with *style.css*, is the only required file for your theme. Typically, you'll have a list of files like the following:

- *404.php*
- *author.php*
- *archive.php*
- *attachment.php*
- *category.php*
- *comments.php*
- *date.php*
- *footer.php*
- *front-page.php*
- *functions.php*
- *header.php*
- *home.php*
- *image.php*

- *index.php*
- *page.php*
- *search.php*
- *sidebar.php*
- *single.php*
- *single-(post-type).php*
- *style.css*
- *tag.php*
- *taxonomy.php*

Some files in this list are loaded when you call a specific get function. For example, `get_header()` loads *header.php*, `get_footer()` loads *footer.php*, and `get_sidebar()` loads *sidebar.php*. Passing a name parameter to these functions will add it to the filename loaded; so, for example, `get_header('alternate');` will load *header-alternate.php* from the theme folder.

The function `comments_template()` loads *comments.php* unless you pass a different filename as the first parameter.

The function `get_search_form()` looks for the file *searchform.php* in your theme folder or outputs the default WordPress search form if no file is found.

The WordPress template hierarchy documentation clearly lays out all the various files WordPress will look for in a theme folder when they

are loaded. You can also check out the [Twenty Nineteen Theme](#) or another well-coded theme to see what filenames will be detected by WordPress. Read the comments in those themes to see when each page is loaded.

When developing apps with custom post types, it's common to want to use a different template when viewing your post types on the frontend. You can override the single post and archive view for your post types by adding files with the names *single-<post\_type>.php* and *archive-<post\_type>.php*, where *post\_type* is set to the value used when the post type was registered.

## Page Templates

One of the easiest ways to get arbitrary PHP code running on a WordPress website is to build a page template into your theme and then use that template on one of your pages.

Some common templates found in WordPress themes include contact forms and landing page forms.

### Sample Page Template

[Example 4-1](#) is a pared-down version of a contact form template that you can drop into your theme's folder.

#### *Example 4-1. Sample page template*

---

```
<?php  
/*  
Template Name: Page - Contact Form  
*/
```

```

//get values possibly submitted by form
$email = sanitize_email( $_POST['email'] );
$cname = sanitize_text_field( $_POST['cname'] );
$phone = sanitize_text_field( $_POST['phone'] );
$message = sanitize_text_field( $_POST['message'] );
$sendemail = !empty( $_POST['sendemail'] );

// form submitted?
if ( !empty( $sendemail )
    && !empty( $cname )
    && !empty( $email )
    && empty( $lname ) ) {

    $mailto = get_bloginfo( 'admin_email' );
    $mailsubj = "Contact Form Submission from " .
get_bloginfo( 'name' );
    $mailhead = "From: " . $cname . " <" . $email . ">\n";
    $mailbody = "Name: " . $cname . "\n\n";
    $mailbody .= "Email: $email\n\n";
    $mailbody .= "Phone: $phone\n\n";
    $mailbody .= "Message:\n" . $message;

    // send email to us
    wp_mail( $mailto, $mailsubj, $mailbody, $mailhead );

    // set message for this page and clear vars
    $msg = "Your message has been sent.';

    $email = "";
    $cname = "";
    $phone = "";
    $message = "";

}

elseif ( !empty( $sendemail ) && !is_email( $email ) )
    $msg = "Please enter a valid email address.";
elseif ( !empty( $lname ) )
    $msg = "Are you a spammer?";
elseif ( !empty( $sendemail ) && empty( $cname ) )
    $msg = "Please enter your name.";
elseif ( !empty( $sendemail ) && !empty( $cname ) && empty(

```

```

$email ) )
    $msg = "Please enter your email address.';

// get the header
get_header();
?>
<div id="wrapper">
    <div id="content">
        <?php if ( have_posts() ) : while ( have_posts() ) :
the_post(); ?>
            <h1><?php the_title(); ?></h1>
            <?php if ( !empty( $msg ) ) { ?>
                <div class="message"><?php echo $msg?></div>
            <?php } ?>
            <form class="general" action=<?php the_permalink(); ?>" method="post">
                <div class="form-row">
                    <label for="cname">Name</label>
                    <input type="text" name="cname" value=<?php echo esc_attr($cname);?>/>
                    <small class="red">* Required</small>
                </div>
                <div class="hidden">
                    <label for="lname">Last Name</label>
                    <input type="text" name="lname" value=<?php echo esc_attr($lname);?>/>
                    <small class="red">LEAVE THIS FIELD BLANK</small>
                </div>
                <div class="form-row">
                    <label for="email">Email</label>
                    <input type="text" name="email" value=<?php echo esc_attr($email);?>/>
                    <small class="red">* Required</small>
                </div>
                <div class="form-row">
                    <label for="phone">Phone</label>
                    <input type="text" name="phone" value=<?php echo esc_attr($phone);?>/>
                </div>
                <div class="form-row">
                    <label for="message">Question or Comment</label>
                    <textarea class="textarea" id="message" name="message" rows="4" cols="55">
                        <?php echo esc_textarea( $message ) ?>

```

```
        </textare>
    </div>

    <div class="form-row">
        <label for="sendemail">&ampnbsp</label>
        <input type="submit" id="sendemail" name="sendemail"
value="Submit"/>
    </div>
</form>
<?php endwhile; endif; ?>
</div>
<?php
// get the footer
get_footer();
?>
```

WordPress will scan all of the *.php* files in your active theme's folder and subfolders (and the parent theme's folder and subfolders) for templates. Any files found with a comment that includes the phrase Template Name : in it will then be made available as a template.

The template is loaded after the WordPress `init` and `wp` actions have already fired. The theme header and the `wp_head` action will not load until you call `get_header()` in your template. So you can use the top of your template file to process form input and potentially redirect before any headers are sent to the page.

Your template file will need to include the same HTML markup as your theme's *page.php* or single-post template. In the preceding example, we include a wrapper `<div>` and content `<div>` around the content of the contact form.

The preceding code also uses the `sanitize_text_field()` and `sanitize_email()` functions to clean up values submitted by the

form. Similarly, it uses the `esc_attr()` and `esc_textarea()` functions to prevent cross-site scripting attacks. These functions are covered more in [Chapter 8](#).

The preceding contact form also incorporates a “honey pot.” A field called `lname` would be hidden using CSS, so normal users would not see this field and thus leave it blank when submitting the form. Bots looking to take advantage of your contact form to send you spam will see the `lname` field and will put some value into it. The code that processes the form checks to make sure that the `lname` field is blank before then sending out the email. Like as a honey pot draws in insects, the hidden `lname` field draws spammers into it so you don’t end up sending email on their behalf.

## Using Hooks to Copy Templates

If you’d rather not change multiple template files when you update the ID or class names of your wrapper `<div>`s, you can create a template that uses the `the_content` filter or another action specific to your theme to place content into the main content area of your page. Then you can load another template file, like the core `page.php` template, which will include calls to load your site’s frame and default layout. [Example 4-2](#) shows how to create a page template that loads the `page.php` template and adds further content below it on certain pages.

---

### *Example 4-2. Hooking template*

```
<?php
/*
Template Name: Hooking Template Example
```

```

*/



//use the default page template
require_once(dirname(__FILE__) . "/page.php");

//now add content using a function called during the
//the_content hook
function template_content($content)
{
    //get the current post in this loop
    global $post;

    //get the post object for the current page
    $queried_object = get_queried_object();

    //we don't want to filter posts that aren't the main post
    if(empty($queried_object) || $queried_object->ID != $post->ID)
        return $content;

    //capture output
    ob_start();
    ?>
    <p>This content will show up under the page content.</p>
    <?php
    $temp_content = ob_get_contents();
    ob_end_clean();

    //append and return template content
    return $content . $temp_content;
}

add_action("the_content", "template_content");

```

In this previous example, we do a little trick to check the current \$post against the \$queried\_object. Typically, the global \$post will be the main post of the page you have navigated to. However, other loops on your page will temporarily set the global \$post to whatever post they are dealing with at the time. For example, if your template uses a WordPress menu, that is really a

loop through posts of type menu. Many sidebars and footer sections will loop through other sets of posts.

The `get_queried_object()` function returns the main post object for the current page. The function returns a different but appropriate object if you are on a term or author page. The function returns `null` on archive pages. Because the previous example is a page template that will be loaded only on single page views, calling `get_queried_object()` there will always return a `$post` object for the current page.

You can also insert your own hook into your `page.php` and other core templates to do something similar. Just add something like `do_action('my_template_hook');` at the point in your page template where you'd like to add in extra content.

## When Should You Use a Theme Template?

In Chapter 3, we covered how to use *shortcodes* to create pages for your plugins. The shortcodes are useful because they allow you to add CMS-managed content above and below the shortcode in the post content field and to keep your code organized within your plugin. So, if you are distributing a plugin and need that page template to go along with it, you should use the shortcode method to generate your page.

Similarly, if you are distributing a theme by itself, you'll need to include any templates needed for the theme within the theme folder.

You could include code for shortcode-based templates within your theme, but templates are a more standard way of templating a page.

And finally, if your template needs to alter the HTML of your default page layouts, you will want to use a template file inside of your theme. Example 4-2 piggybacks on the *page.php* template to avoid having to rewrite the wrapping HTML. But if the whole point of the template is to rewrite the wrapping HTML (e.g., with a landing page template where you want to hide the default header, footer, and menu), you definitely need to use a template.

## Theme-Related WordPress Functions

Next, we will discuss `get_template_part( $slug, $name = null )`. Here, the `get_template_part()` function can be used to load other *.php* files (template parts) into a file in your theme.

According to the Codex, `$slug` refers to “the slug name for the generic template,” and `$name` refers to “the name of the specialized template.” In reality, both parameters are simply concatenated with a dash to form the filename looked for: *slug-name.php*.

The Twenty Twelve theme uses `get_template_part()` to load a specific post format “content” part into the WordPress loop:

```
<?php /* Start the Loop */ ?>
<?php while ( have_posts() ) : the_post(); ?>
    <?php get_template_part( 'content',
    get_post_format() ); ?>
<?php endwhile; ?>
```

If your template part is in a subfolder of your theme, add the folder name to the front of the slug:

```
get_template_part('templates/content', 'page');
```

The `get_template_part()` function uses the `locate_template()` function of WordPress to find the template part specified, which then loads the file using the `load_template()` function. `locate_template()` first searches within the child theme. If no matching file is found in the child theme, the parent theme is searched.

Besides searching both the child and parent themes for a file, the other benefit to using `get_template_part()` over a standard PHP `include` or `require` call is that a set of WordPress global variables is set up before the file is included. The following example is the source for the `load_template()` function as of WordPress 4.9.7,<sup>2</sup> showing the global variables that are set. Notice that the `query_vars` array is also extracted into the local scope. The `query_vars $s` is given special attention and escaped here since it's a common vector for XSS attacks, and many themes forget to escape the search query when rendering it into the search field:

```
<?php
function load_template( $_template_file, $require_once =
true ) {
    global $posts, $post, $wp_did_header, $wp_query,
$wp_rewrite;
    global $wpdb, $wp_version, $wp, $id, $comment,
$user_ID;
```

```

    if ( is_array( $wp_query->query_vars ) )
        extract( $wp_query->query_vars, EXTR_SKIP );

    if ( isset( $s ) )
        $s = esc_attr( $s );

    if ( $require_once )
        require_once( $_template_file );
    else
        require( $_template_file );
}

?>

```

## Using `locate_template` in Your Plugins

A common design pattern used in plugins is to include templates in your plugin folder and allow users to override those templates by adding their own versions to the active theme. For example, in SchoolPress, teachers can invite students to their class. The invite form is stored in a template within the plugin:

```

//schoolpress/templates/invite-students.php
?>
<p>Enter</p>
<form action="" method="post">
    <label for="email">Email:</label>
    <input type="text" id="email" name="email" value="" />
    <input type="submit" name="invite" value="Invite
Student" />
</form>

```

SchoolPress is envisioned as an SaaS application, but we also plan to release a plugin version for others to use on their own sites. Users of the plugin may want to override the default template without editing the core plugin, since edits to the core plugin would be overwritten when the plugin was upgraded.

To enable our plugin users to override the invite template, we use code like the following when including the template file:

```
//schoolpress/shortcodes/invite-students.php
function sp_invite_students_shortcode($atts, $content=null,
$code="")
{
    //start output buffering
    ob_start();

    //look for an invite-students template part in the
    active theme
    $template =
locate_template('schoolpress/templates/invite-
students.php');

    //if not found, use the default
    if(empty($template))
        $template = dirname(__FILE__) .
        '/../../templates/invite-students.php';

    //load the template
    load_template($template);

    //get content from buffer and output it
    $temp_content = ob_get_contents();
    ob_end_clean();
    return $temp_content;
}

add_shortcode('invite-students',
'sp_invite_students_shortcode');
```

This code uses our shortcode template from [Chapter 3](#). But instead of embedding the HTML into the shortcode function, we load it from a template file. We first use `locate_template()` to search for the template in the active child and parent themes. Then, if no file is found, we set `$template` to the path of the default template

bundled with the plugin. The template is loaded using `load_template()`.

## Style.css

The `style.css` file of your theme must contain a comment used by WordPress to track the theme's version and other information to show in the WordPress dashboard. Here is the comment from the top of `style.css` in the Twenty Nineteen theme:

```
/*
Theme Name: Twenty Nineteen
Theme URI: https://wordpress.org/themes/twentynineteen/
Author: the WordPress team
Author URI: https://wordpress.org/
Description: Our 2019 default theme is designed to show off
the power of the
block editor. It features custom styles for all the default
blocks, and is
built so that what you see in the editor looks like what
you'll see on your
website. Twenty Nineteen is designed to be adaptable to a
wide range of
websites, whether you're running a photo blog, launching a
new business, or
supporting a non-profit. Featuring ample whitespace and
modern sans-serif
headlines paired with classic serif body text, it's built to
be beautiful on
all screen sizes.
Requires at least: WordPress 4.9.6
Version: 1.4
License: GNU General Public License v2 or later
License URI: LICENSE
Text Domain: twentynineteen
Tags: one-column, flexible-header, accessibility-ready,
custom-colors, custom-menu, custom-logo, editor-style,
featured-images, footer-widgets, rtl-language-support,
sticky-post, threaded-comments, translation-ready
This theme, like WordPress, is licensed under the GPL.
Use it to make something cool, have fun, and share what
```

```
you've learned with others.  
Twenty Nineteen is based on Underscores  
https://underscores.me/,  
(C) 2012-2018 Automattic, Inc.  
Underscores is distributed under the terms of the GNU GPL v2  
or later.  
Normalizing styles have been helped along thanks to the fine  
work of  
Nicolas Gallagher and Jonathan Neal  
https://necolas.github.io/normalize.css/  
*/
```

The `style.css` file of the active theme (and parent theme if applicable) is automatically enqueued by WordPress.

## Versioning Your Theme's CSS Files

It's good practice to set a version for your CSS files when loading them through `wp_enqueue_style()`. This way, if you update your CSS, you can also update the version, and avoid having site users see a seemingly broken site that uses a version of the stylesheet cached by the browser.

When WordPress enqueues your theme's `style.css` file for you, it uses the overall WordPress version when loading the stylesheet. The line output in your site's head tag will look like this:

```
<link rel='stylesheet'  
      id='twentynineteen-style-css'  
      href='.../wp-  
content/themes/twentynineteen/style.css?ver=1.4'  
      type='text/css'  
      media='all' />
```

Updates to the stylesheet, your app's version number, or even the version number set in the `style.css` comment won't update the version

added to the stylesheet when enqueued. It will always match the WordPress version number.

One solution is to remove all CSS from your *style.css* file into other CSS files in your theme and load *those* CSS files through `wp_enqueue_style()` calls in the theme's *functions.php* file. It would look like this for *style.css*:

```
/*
Theme Name: SchoolPress
Version: 1.0

That's it! All CSS can be found in the "css" folder of
the theme.
*/
```

and like this for *functions.php*:

```
<?php
define( 'SCHOOLPRESS_VERSION', '1.0' );
function sp_enqueue_theme_styles() {
    if ( !is_admin() ) {
        wp_enqueue_style( 'schoolpress-theme',
            get_stylesheet_directory_uri() .
        '/css/main.css',
            NULL,
            SCHOOLPRESS_VERSION
        );
    }
}
add_action( 'init', 'sp_enqueue_theme_styles' );
?>
```

A constant like `SCHOOLPRESS_VERSION` would typically be defined in our main plugin file, but it's included here for clarity. The preceding code loads our new */css/main.css* file with the main app

version appended so new versions of the app won't conflict with browser-cached stylesheets.

There is another way to change the version of the main *style.css* file without moving it to another file entirely. We use the `wp_default_styles` filter. This filter passes an object containing the default values used when a stylesheet is enqueued. One of those values is the `default_version`, which can be changed like so:

```
define('SCHOOLPRESS_VERSION', '1.0');
function sp_wp_default_styles($styles)
{
    //use release version for stylesheets
    $styles->default_version = SCHOOLPRESS_VERSION;
}
add_action("wp_default_styles", "sp_wp_default_styles");
```

Now our main stylesheet will be loaded using the SchoolPress app version instead of the main WordPress version. We can keep our CSS in *style.css* if we want to, though it's often a good idea to move at least some parts of the CSS into separate files in a “css” folder of your theme.

## functions.php

The *functions.php* file of your active theme (and parent theme if applicable) is loaded every time WordPress loads. For this reason, the *functions.php* file is a popular place to add little hacks and other random bits of code. On a typical WordPress site, the *functions.php* file can quickly become a mess.

However, we're developing a well-planned WordPress app, and *our function.php* files don't need to be a mess. Just like we break up the core functions of our main app plugin into smaller `includes`, you should do the same with your theme's *functions.php*. You could add files similar to the following to your theme's folder:

*/includes/functions.php*

Where you really place helper functions.

*/includes/settings.php*

For code related to theme settings and options.

*/includes/sidebar.php*

To define sidebars/widget areas.

Additionally, make sure that code you are adding to your theme's *functions.php* is related to the frontend display of your site. Code that applies to the WordPress dashboard, backend processing for your app, or your entire app in general should most likely be added somewhere within the main app plugin.

## Themes and CPTs

As CPTs are just posts, by default, your CPTs will be rendered using the *single.php* template, or *index.php* if no *single.php* template is available. You can also add a file to your theme of the form *single-<post\_type>.php*, where *<post\_type>* is the slug of your CPT. If available, this file is used to render the single-post view of that post type.

Similarly, you can add an *archive-<post\_type>.php* file to render the archive view of your CPT if the `has_archive` flag is set in the CPT definition. We cover CPTs (including specifying templates for them) in more detail in [Chapter 5](#).

## Popular Theme Frameworks

When building apps with WordPress, there are many theme frameworks—both WordPress-specific frameworks and general-purpose HTML/CSS frameworks—you can use. Whether you intend to use the theme framework to build a quick proof of concept or to use it as a core component of your custom-built theme, using a theme framework can save you a lot of time.

We briefly cover some common theme frameworks and dive deeper into how to use two of the most popular ones for WordPress app development.

But first, what does a theme framework provide?

### WordPress Theme Frameworks

WordPress theme frameworks are themes that are meant to be used as parent themes or starter themes to jump-start your frontend development. Theme frameworks typically include basic styles and layouts for blog posts, as well as archives, pages, sidebars, and menus. They are of varying weights, and some include CSS classes, shortcodes, and other handy bits of code to help you create new

layouts and add UI elements to your pages. All frameworks will likely save you a lot of time.

There are two reasons to choose one theme framework over another: you either choose a child theme that visually looks very close to your vision for your app, or you choose a framework coded in a way that feels right when you work with it.

## \_S (UNDERSCORES)

The starter theme `_s` (pronounced “underscores”) is published by Automattic, and has all the common components you need in a WordPress theme. Unlike most other frameworks, `_s` is not meant to be used as a parent theme, but as a starting point for your own parent theme. Most themes developed by Automattic for WordPress.com are based on the `_s` theme.

To use `_s`, download the code and change the directory name and all references to `_s` with the name of your theme. You can find good instructions for doing this in [the project’s \*README\* file](#) or, even better, a tool to automatically do it for you on the [underscores website](#).

The stylesheet in `_s` is very minimal with no real styling, just a bit of code for layout and some common readability and usability settings. `_s` is best for designers who are able to and who want to build their own theme from scratch. It’s basically code you would have to write somehow for your theme yourself. The `_s` code is not abstracted as heavily as some other theme frameworks, and so using the framework

should be easier to pick up for designers more familiar with HTML and CSS than PHP.

## MEMBERLITE

Memberlite is a theme written by Jason Coleman and Kimberly Coleman of Stranger Studios. This theme was built with membership sites in mind, but can be used on a wide variety of sites. The “lite” part of the title describes the theme’s lightweight (technically and aesthetically) design.

Memberlite is designed responsively to fit on screens of varying sizes. It has templates and sections for everything a modern website needs. It has a couple of companion plugins (Memberlite Elements and Memberlite Shortcodes), with tools to add finer control over sidebars, banners, page layouts, and other aspects of your website.

Memberlite is best for designer-developers, and is really our choice for starting themes based on its balance of framework support on the design and coding side of theme development.

## GENESIS

Genesis is a theme framework developed by StudioPress and used in more than 40 child themes published by StudioPress and in many more themes published by third-party designers. The Genesis theme is meant to be used as a parent theme. StudioPress has child themes that are appropriate across a number of business and website types. Or you can create your own child theme that inherits from Genesis.

The Genesis framework abstracts the underlying HTML and CSS more than the other frameworks listed here. We find this makes it a little harder to work with when doing larger customizations. However, Genesis would be a good choice if you find one of their child themes is 80% of the way toward the look you want or if you find their framework easier to work with than other options.

## Non-WordPress Theme Frameworks

In addition to WordPress theme frameworks, there are also application UI frameworks that provide markup, stylesheets, and images for common UI patterns and elements. Some popular UI frameworks include Twitter's [Bootstrap](#) and Zurb's [Foundation](#).

Incorporating a UI framework into your theme can be as easy as copying a few files into the theme folder and enqueueing the stylesheets and JavaScript. This gives you easy access to styled UI elements like buttons, tabs, pagination, breadcrumbs, labels, alerts, and progress bars.

Next, we'll cover how to add Bootstrap assets into a Memberlite child theme, but the same process should work for other combinations of WordPress themes and UI frameworks.

## Creating a Child Theme for Memberlite

To create your theme, you'll need to follow these steps:

1. Create a new folder in your `wp-content/themes` folder. Then, give it the name `memberlite-child`.

2. Create a *style.css* file in the *memberlite-child* folder.

3. Paste the following into your *style.css* file:

```
/*
THEME NAME: Memberlite Child
THEME URI: http://bwawwp.com/wp-
content/themes/memberlite-child/
DESCRIPTION: Memberlite Child Theme
VERSION: 0.1
AUTHOR: Jason Coleman
AUTHOR Uri: http://bwawwp.com
TAGS: memberlite, child, tag
TEMPLATE: memberlite
*/
@import url("../memberlite/style.css");
```

The key field in the comment is the TEMPLATE field, which needs to match the folder of the parent theme, in this case memberlite. The only required file for a child theme is *style.css*. So at this point, you've created a child theme.

You can either copy all the CSS from the parent theme's *style.css* into the child theme's *style.css* and edit what you want to, or use *@import\_url* like we've done here to import the rules from the parent theme's stylesheet and add more rules below to override the parent theme's styles.

To enqueue the bootstrap files, you will also need a *functions.php* file.

4. Create an empty *functions.php* file in the *memberlite-child* folder for now.

## Including Bootstrap in Your App's Theme

In general, importing Bootstrap into the Memberlite theme is kind of silly compared to finding a theme based on Bootstrap or just copying in the CSS rules you need. However, importing frameworks and libraries into your theme is something you might run into. The following will give you an idea of how to go about importing other libraries and frameworks into your theme.

Download the Bootstrap ZIP file into your *memberlite-child* folder. After unzipping it, you will have a *dist* folder containing the CSS and JavaScript files for Bootstrap. You can rename this folder to *bootstrap* and delete the Bootstrap ZIP file. Your child theme folder should now look like this:

- *memberlite-child*
  - *bootstrap*
    - *CSS*
    - *js*
  - *functions.php*
  - *style.css*

Now, we'll enqueue the Bootstrap CSS and JavaScript by adding this code into the *functions.php* file inside your child theme:

```
<?php
function memberlite_child_init() {
    wp_enqueue_style(
        'bootstrap',
        get_stylesheet_directory_uri() .
        '/bootstrap/css/bootstrap.min.css',
        'style',
```

```
        '3.0'
    );
wp_enqueue_script(
    'bootstrap',
    get_stylesheet_directory_uri() .
    '/bootstrap/js/bootstrap.min.js',
    'jquery',
    '3.0'
);
}
add_action( 'init', 'memberlite_child_init' );
?>
```

Note that we set the dependencies for the Bootstrap CSS to `style`, which ensures that the Bootstrap stylesheet loads after the Memberlite stylesheet. We also set the Bootstrap JavaScript to depend on `jquery` and set the version of both files to `3.0` to match the version of Bootstrap used.

At this point you could use any of your favorite Bootstrap styles or JavaScript in your WordPress theme. Many of the Bootstrap styles for columns and layout aren't being used in the Memberlite markup (Memberlite has its own layout system), and so they won't be applicable to your theme. But the styles for form elements and buttons would be useful for app developers.

## Menus

Menus are an important part of most apps, and apps often have special needs for their menus that other websites don't have. Some apps have multiple menus. Many mobile apps have a main navigational menu at the top and a toolbar-like menu along the

bottom. Some apps have dynamic menus. Many apps have different menus or menu items for logged-in users than for logged-out users. Menu items can be based on a user's membership level or admin capabilities.

Before we get into how to build more complicated menus and navigational elements with WordPress, let's cover the standard way to add a menu to your theme.

## Navigation Menus

Since WordPress version 3.0, the standard method for adding navigation menus to themes involved registering the menu in the theme's code, designating where in the theme the menu will appear, and then managing the menu through the WordPress dashboard.

The main benefit to using WordPress's built-in menu functionality is that end users can control the content of their menus using the dashboard GUI. Even if you are a developer with full control over your app, it is still a good idea to use the built-in menus in WordPress since you may have stakeholders who would want to manage menus or you may want to distribute your theme to others in the future. The WordPress navigation menus are also very easy to reposition and can take advantage of other code using menu-related hooks or CSS styles.

To register a new navigational menu, use the

```
register_nav_menu( $location, $description )
```

function. The `$location` parameter is a unique slug used to identify the menu. The `$description` parameter is a longer title

for the menu shown in the drop-down in my menu tool in the dashboard:

```
register_nav_menu('main', 'Main Menu');
```

You can also register many menus at once using the `register_nav_menus()` (with an s) variant. This function accepts an array of locations in which the keys are the `$location` slugs and the values are the `$description` titles:

```
register_nav_menus(array(  
    'main' => 'Main',  
    'logged-in' => 'Logged-In'  
) );
```

To place a navigational menu into your theme, use the `wp_nav_menu()` function:

```
wp_nav_menu( array('theme_location' => 'main') );
```

The `theme_location` parameter should be set to the `$location` set with `register_nav_menu()`. The `wp_nav_menu()` function can take many other parameters to change the behavior and markup of the menu. [This WordPress Codex page on navigation menus](#) is a good resource as the various parameters to the `wp_nav_menu()` function and other ways to customize menus. We cover some of our favorite recipes in the following sections.

## Dynamic Menus

There are two main methods to make your WordPress menus dynamic so that different menu items show up on different pages or under different circumstances. The first is to set up two menus and load a different menu depending on the case. Here is a code example from the Codex showing how to display a different menu to logged-in users and logged-out users:

```
if ( is_user_logged_in() ) {
    wp_nav_menu( array( 'theme_location' => 'logged-in-menu' ) );
} else {
    wp_nav_menu( array( 'theme_location' => 'logged-out-menu' ) );
}
```

The other way to make your menu dynamic is to use the `nav_menu_css_class` filter to add extra CSS classes to specific menu items. Then you can use CSS to hide/show certain menu items based on their CSS class.

Say you want to remove a login link from a menu when you're on the login page.<sup>3</sup> You could use code like this:

```
function remove_login_link($classes, $item)
{
    if(is_page('login') && $item->title == 'Login')
        $classes[] = 'hide';           //hide this item

    return $classes;
}
add_filter('nav_menu_css_class', 'sp_nav_menu_css_class',
10, 2);
```

You can also customize the markup of your menus by using Custom Walker classes (see Chapter 7).

## Responsive Design

We could write another whole book about responsive design. Luckily for us, many people already have, including Clarissa Peterson, who wrote *Learning Responsive Web Design* (O'Reilly). The general concept behind responsive design is to detect properties of the client device and to adjust your app's layout, design, and functionality to work best for that device. Now, let's get into a few different techniques for doing this.

### Device and Display Detection in CSS

*Media queries* are the main method of device detection in CSS. They're used in stylesheets or added as a property of the `<link>` tag used to embed a stylesheet to limit the scope of the stylesheet's CSS rules to a specific media type or case in which a particular media feature is available. Mozilla does a good job explaining media queries, and listing the various properties and operators you can use to construct a media query.

A common use of media queries is to hide certain elements and adjust font and element sizes when someone is printing. You would specify that media query in a `<link>` tag, inside a stylesheet, and through a `wp_enqueue_style` call, as follows:

```
<link rel="stylesheet" media="print" href="example.css" />
```

```

<style>
@media print
{
    .hide-from-print {display: none;}
    .show-when-printing {display: auto;}
}
</style>

<?php
    wp_enqueue_style('example', 'example.css', NULL,
'1.0', 'print');
?>

```

A more typical example in the responsive design world is to check for a min-width and/or max-width in the media query to adjust styles as the screen becomes smaller or larger. The following is an example from the Bootstrap responsive stylesheet that adjusts CSS rules for screens between 768 and 979 pixels (the current width of a typical browser window on a monitor). Sizes greater than 979 pixels could be considered extra wide:

```

@media (min-width: 768px) and (max-width: 979px) {
    .hidden-desktop {
        display: inherit !important;
    }
    .visible-desktop {
        display: none !important ;
    }
    .visible-tablet {
        display: inherit !important;
    }
    .hidden-tablet {
        display: none !important;
    }
}

```

Another common task handled with media queries is to change styles, and specifically swap images, when a browser has a Retina high-resolution screen.<sup>4</sup>

Here is a mix of media queries used in some WordPress dashboard CSS for detecting a high-resolution display. The queries test against pixel ratio and DPI. Values vary among displays, but most standard-definition displays will have a 1:1 pixel ratio and 96 DPI. A Retina display has a pixel ratio of 2:1 and DPI of 196 or higher, but we can test for minimal values somewhere between standard definition and Retina-level definition to catch other high-resolution displays:

```
@media (-o-min-device-pixel-ratio: 5/4), /*  
Opera */  
        (-webkit-min-device-pixel-ratio: 1.25), /*  
Webkit */  
        (min-resolution: 120dpi) { /*  
Others */  
    /* add your high res CSS here */  
}
```

Media queries are powerful, and you can use them to make UIs that are very flexible. Browsers and CSS standards evolve constantly. It's important to stay on top of things so current phones, tablets, and monitors show your app as you intended.

Which properties to look out for and how to adjust your stylesheet to accommodate them is outside the scope of this book, but hopefully you get the idea and understand how to incorporate media queries into your WordPress themes.

## Device and Feature Detection in JavaScript

Your app's JavaScript can also benefit from device and feature detection. jQuery offers methods to detect the window and screen sizes and other information about the browser. Many HTML5 features that may or may not be available in a certain browser can be tested before being put to use.

## **DETECTING THE SCREEN AND WINDOW SIZE WITH JAVASCRIPT AND JQUERY**

JavaScript makes the screen width and height available in the `screen.width` and `screen.height` properties. You can also use `screen.availWidth` and `screen.availHeight` to get the available width and height, which accounts for pixels taken up by toolbars and sidebar panels in the browser window.

If you are already using jQuery, you can use the `width()` method on any element on your page to get its width, but you can also use it on the `$(document)` and `$(window)` objects to get the width of the document and window, respectively. You can also use the `height()` property on the document and window objects and any element on your page.

The values for `$(window).width()` and `$(window).height()` should be the same as `screen.availWidth` and `screen.availHeight`. This is the available size of the browser viewport, minus any toolbars or sidebar panels, or, more accurately, how much room you have for displaying HTML.

The width and height of the `$(document)` will return the total scrollable width and height of your rendered web page. When using the width and height in your JavaScript code, you will often want to update things if the window size changes. This can happen if someone resizes a browser window on their desktop, rotates a phone from portrait to landscape, or does any number of things that could change the width or height of the window. jQuery offers an easy way to detect these changes so you can update your layout accordingly:

```
//bind an event to run when the window is resized
jQuery(window).resize(function() {
    width = jQuery(window).width();
    height = jQuery(window).height();
    //update your layout, etc
});
```

You can bind a resize event to any element, not just the full window. Elements on your page might grow and contract as a user interacts with your page, possibly adding elements through Ajax forms, dragging resizable elements on the screen, or otherwise moving things around.

## FEATURE DETECTION IN JAVASCRIPT

When you're building a modern app UI using HTML5 features, you will sometimes want to detect whether a certain HTML5 feature is unavailable so you can provide an alternative or fallback. Mark Pilgrim's *Dive into HTML5* has a good list of general methods for detecting HTML5 features:

1. Check whether a certain property exists on a global object (such as `window` or `navigator`).

2. Create an element, then check whether a certain property exists on that element.
3. Create an element, check if a certain method exists on that element, and then call the method and check the value it returns.
4. Create an element, set a property to a certain value, and then check whether the property has retained its value.

If you need to do only one such detection, some of the examples on the *Dive into HTML5* site will give you ideas on how to roll your own bit of detection. If you need to do a lot of feature detection, a library like [Modernizr.js](#) will help.

To use Modernizr.js, first grab the version of the script you need from their site: Modernizr offers a tool that will ask you which parts of the script you need, and then generate a minimized .js file containing only those bits. Place this file in your theme or plugin folder and enqueue it:

```
<?php
function sp_wp_footer_modernizr() {
    wp_enqueue_script(
        'modernizr',
        get_stylesheet_directory_uri() .
'/js/modernizr.min.js'
);?>
<script>
    //change search inputs to text if
unsupported
    if(!Modernizr.inputtypes.search)

jQuery('input[type=search]').attr('type', 'text');
</script>
<?php
}
```

```
add_action( 'wp_footer', 'sp_wp_footer_modernizr' );
?>
```

The Modernizr documentation contains a list of features detectable with Modernizr.js.

jQuery also provides a similar set of checks, limited to things that jQuery needs to check itself through the `jQuery.support` object. If a check you are trying to do is done by jQuery already, you can avoid the overhead of Modernizr.js by using the jQuery check. A list of feature flags set by `jQuery.support` can be found on their website:

```
jQuery(document).ready(function() {
    //only load AJAX code if AJAX is available
    if(jQuery.support.ajax)
    {
        //AJAX code goes here
    }
});
```

## Device Detection in PHP

Device detection in PHP is based on the `$_SERVER['HTTP_USER_AGENT']` global created by PHP. This value is set by the browser itself and so is definitely not standardized, often misleading, and potentially spoofed by web crawlers and other bots. It's then best to avoid PHP-based browser detection if you can by making your code as standards based as possible and using the CSS and JavaScript methods described for feature detection.

If you want a general idea of the kind of browser accessing your app, the user agent string is the best we have. Here is a simple test script echoing the user agent string and an example of what one will look like:

```
<?php  
echo $_SERVER['HTTP_USER_AGENT'];  
  
/*  
     Outputs something like:  
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4)  
AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/28.0.1500.95 Safari/537.36  
*/  
?>
```

This user agent string includes useful information, but perhaps too much. There are no fewer than five different browser names in that string. So which browser is it? Mozilla, KHTML, Gecko, Chrome, or Safari? In this case, we were running Chrome on a MacBook Air running OS X.

Did we already mention that there is no standard for the user agent string browsers will send? Historically, browsers include the names of older browsers to basically say, “I can do everything this browser does, too.”

[WebAIM](#) contains a funny summary of the history of various user agent strings, including this bit explaining the pedigree of the Chrome browser:

*And then Google built Chrome, and Chrome used Webkit, and it was like Safari, and wanted pages built for Safari, and so pretended to be Safari. And thus Chrome used WebKit, and pretended to be Safari, and WebKit pretended to be KHTML, and KHTML pretended to be Gecko, and all browsers pretended to be Mozilla, and Chrome called itself Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHTML, like Gecko) Chrome/0.2.149.27 Safari/525.13, and the user agent string was a complete mess, and near useless, and everyone pretended to be everyone else, and confusion abounded.*

—Aaron Anderson

## BROWSER DETECTION IN WORDPRESS CORE

Luckily, WordPress has done a bit of the work behind parsing the user agent string and exposes some global variables and a couple of methods that cover the most common browser detection-related questions. The following globals are set by WordPress in *wp-includes/vars.php*:

- \$is\_lynx
- \$is\_gecko
- \$is\_winIE
- \$is\_macIE
- \$is\_opera
- \$is\_NS4
- \$is\_safari
- \$is\_chrome
- \$is\_iphone

- \$is\_IE

And for detecting certain servers, we have the following:

- \$is\_apache
- \$is\_IIS
- \$is\_iis7

Finally, you can use the `wp_is_mobile()` function, which checks for the word *mobile* in the user agent string as well as a few common mobile browsers.

Here is a quick example showing how you might use these globals to load different scripts and CSS:

```
<?php
function sp_init_browser_hacks() {
    global $is_IE;
    if ( $is_IE ) {
        //check version and load CSS
        $user_agent = strtolower(
$_SERVER['HTTP_USER_AGENT'] );
        if ( strpos( 'msie 6.', $user_agent ) !==
false &&
            strpos( 'opera', $user_agent ) ===
false ) {
            wp_enqueue_style(
                'ie6-hacks',
get_stylesheet_directory_uri() . '/css/ie6.css'
            );
        }
    }
    if ( wp_is_mobile() ) {
        //load our mobile CSS and JS
    }
}
```

```

        wp_enqueue_style(
            'sp-mobile',
            get_stylesheet_directory_uri() .
'css/mobile.css'
        );
        wp_enqueue_script(
            'sp-mobile',
            get_stylesheet_directory_uri() .
'js/mobile.js'
        );
    }
}

add_action( 'init', 'sp_init_browser_hacks' );
?>

```

## BROWSER DETECTION WITH PHP'S GET\_BROWSER()

PHP actually has a great function for browser detection built in:

`get_browser()`. Here is a simple example calling `get_browser()` and displaying some typical results:

```

<?php
$browser = get_browser();
print_r($browser);

/*
    Would produce output like:

        stdClass Object (
[browser_name_regex] => $^mozilla/5\.0 \(.*intel mac os
x.*\)
applewebkit/.* \(KHTML, like
gecko\).*chrome/28\..*safari/.*$$
[browser_name_pattern] => Mozilla/5.0 (*Intel Mac OS X*)
AppleWebKit/* (KHTML, like Gecko)*Chrome/28.*Safari/*
[parent] => Chrome 28.0
[platform] => MacOSX
[win32] =>
[comment] => Chrome 28.0
[browser] => Chrome

```

```
[version] => 28.0
[majorver] => 28
[minorver] => 0
[frames] => 1
[iframes] => 1
[tables] => 1
[cookies] => 1
[javascript] => 1
[javaapplets] => 1
[cssversion] => 3
[platform_version] => unknown
[alpha] =>
[beta] =>
[win16] =>
[win64] =>
[backgroundsounds] =>
[vbscript] =>
[activexcontrols] =>
[ismobiledevice] =>
[issyndicationreader] =>
[crawler] =>
[aolvernversion] => 0
)
*/
```

This is pretty amazing stuff! So why is this function last in the section on detecting a browser with PHP? The answer is that the `get_browser()` function is unavailable or out of date on most servers. For the function to give you useful information, or in most cases work at all, you need to download an up-to-date *browscap.ini* file and configure PHP to find it. If you are distributing your app, you'll want to use a different method to detect browser capabilities. However, if you are running your own app on your own servers, `get_browser()` is fair game.

An up-to-date *browscap.ini* file can be found at the [Browser Capabilities Project website](#). Make sure you get one of the files formatted for PHP. We recommend the *lite\_php\_browscap.ini* file,

which is half the size but contains information on the most popular browsers.

Once you have the *.ini* file on your server, you'll need to update your *php.ini* file to point to it. Your *php.ini* file probably has a line for *browscap* commented out. Uncomment it and make sure it's pointing to the location of the *.ini* file you downloaded. It should look something like this:

```
[browscap]
browscap = /etc/lite_php_browscap.ini
```

Now restart your web server (Apache, Nginx, etc.) and *get\_browser()* should be working.

## Final Note on Browser Detection

We spent a lot of space here on browser detection, but in practice, you should only use it as a last resort. When a certain browser is giving you pain with a piece of design or functionality, it is tempting to try to detect it and code around it. However, if it's possible to find another workaround that gets a similar result without singling out specific browsers, it's usually better to go with that solution.

For one thing, as we've seen here, the user agent string has no standards, and you might need to regularly update your code to parse it to account for new browsers and browser versions.

Second, in some cases, a browser-specific issue is a symptom of a bigger problem in your code. There may be a way to simplify your

design or functionality to work better across multiple browsers, devices, and screen sizes.

The goal with responsive design and programming is to build something that will be flexible enough to account for all of the various browsers and clients accessing your app, whether you know about them or not.

- 
- - <sup>1</sup> If you find that you *must* hack the core files to get something to work, first reconsider whether you really need to do this. If you do need to change a core WordPress file, add hooks instead, and submit those hooks as a patch to the next version of WordPress.
    - <sup>2</sup> We added a line break and removed some curly braces so it would fit better in print.
    - <sup>3</sup> You could check `$_SERVER['PHP_SELF']` to see whether you're on the `wp-login.php` page. In this example, we assume our login is on a WordPress page with the slug `login`.
    - <sup>4</sup> *Retina* is Apple's brand name for their high-resolution displays. However, you may find the term "retina" used in code comments and documentation to refer to any high-resolution display.

# Chapter 5. Custom Post Types, Post Metadata, and Taxonomies

---

CPTs are what really makes WordPress a content management system. With them, you can quickly build out custom functionality and store data in a consistent way.

## Default Post Types and CPTs

With a default installation of WordPress, you have several post types already being used. The post types you may be most familiar with are pages and posts, but there are a few more. These post type values are all stored in the database `wp_posts` table, and they all use the `post_type` field to separate them.

### Page

WordPress pages are what you use for your static content pages like home, about, contact info, bio, or any custom page you want. Pages can be indefinitely nested in any hierarchical structure. They can also be sorted by `menu_order` value.

### Post

Your posts are your blog or news or whatever you want to call your constant barrage of content to be indexed by internet search engines. You can categorize your posts, tag them with keywords, set publish dates, and more. In general, posts are shown in some kind of list view in reverse chronological order on the frontend of your website.

## Attachment

Whenever you upload an image or file to a post, it stores the file not only on the server, but also as a post in the `wp_posts` table with a `post_type` attachment.

## Revisions

WordPress has your back and saves your posts as revisions every time you or anyone edits a post. This feature is on by default, and you can use it to revert your content to what it was if something got messed up along the way.

Sometimes, if your application is set up to make a lot of `post_content` changes, the `wp_posts` table becomes flooded with post revisions. So you may want to limit the amount of revisions stored in the `wp_posts` table. To do this, add the following code to your `wp-config.php` file:

```
define( 'WP_POST_REVISIONS', 5 );
```

Here, 5 is the number of revision posts to store for a given post. A value of 0 will turn off post revisions. A value of `true` or `-1` will

store an infinite number of revisions (it can take a lot of disk space to store infinity something).

## Navigation Menu Item

Every time you build a custom menu using the WordPress core menu builder (`wp-admin → appearance → menus`), you are storing posts with information for your menus.

## Custom CSS

The Customizer has a setting for “Additional CSS.” This CSS is stored in a custom post of the type `custom_css`. The additional CSS is specific to the active theme, and each theme will have a separate `custom_css` post to store the CSS in. It might seem odd to store these additional CSS rules in a custom post, but it was done that way for performance reasons. You could imagine a site with *a lot* of custom CSS. Options are sometimes automatically loaded or cached, and those systems could break or slow down with large amounts of CSS.

## Changesets

Changesets are like post revisions but for your customizer settings instead of post content. When changing settings in the Customizer, those changes are stored in a post of the type `customize_changeset`. If you accidentally close your browser tab and then return to the customizer, WordPress will prompt you to restore those changes from the changeset.

## **oEmbed Cache**

WordPress allows you to embed content from supported oEmbed providers by placing the URL to the content on its own line in the post editor. If you place the URL to a YouTube video on its own line, WordPress will detect that the URL is for YouTube.com, then make a call to the corresponding oEmbed URL to insert the YouTube player into the post on the frontend of your site. The embed code from that request is cached into a custom post of type `oembed_cache` and set as a child of the post you were editing.

## **User Requests**

Since WordPress 4.9, administrators have been able to manage personal data exports or personal data deletions on behalf of users.<sup>1</sup> You can find the screen to manage personal data exports and deletions on the Tools menu in the dashboard. These requests are stored in a custom post of type `user_request`. See more about how these requests are handled in the [WordPress reference for the `wp\_create\_user\_request\(\)` function](#).

## **Reusable Blocks**

Part of the new block-based editor introduced in WordPress 5.0 is the ability to save blocks you configure within a post as a “reusable block.” These reusable blocks are stored in a custom post of the type `wp_block`. Reusable blocks are covered in [Chapter 11](#).

## **Defining and Registering CPTs**

Just like the default WordPress post types, you can create your own CPTs to manage any data you need, depending on what you are building. Every CPT is really just a post used differently. You could register a CPT for a dinner menu at a restaurant, for cars for an auto dealer, for people to track patient information and documents at a doctor's office, or for pretty much anything you can think of. No, really, any type of content you can think of can be stored as a post with attached files, custom metadata, and custom taxonomies.

In our SchoolPress example, we are going to be building a CPT for managing homework assignments on a teacher's website. Our teacher wants to make a post of some kind where they can add assignments and their students can find them on the class website. The teacher would also like to be able to upload supporting documents and have commenting available in case any students have questions. A CPT sounds in order, doesn't it?

We can store this information the same way posts are dealt with and display them to the end user in the theme using the same `wp_query` loop we would with posts.

## **`register_post_type( $post_type, $args );`**

You can register a CPT by using the function `register_post_type()`. And in most cases, you'll register your CPT in your theme's *functions.php* file or in a custom plugin file. This function expects two parameters—the name of the post type you are creating and an array of arguments:

### `$post_type`

The name of your CPT; in our example, our CPT name is “homework.” This string must be no longer than 20 characters and can’t have capital letters, spaces, or any special characters except a hyphen or an underscore. If you are making a plugin for public consumption, you will want to use a prefix on your CPT’s name to avoid conflicts if another plugin uses the same name.

### `$args`

This is an array of many different arguments that will dictate how your CPT will be set up.

Following is a list of all available arguments and what they are used for:

#### `label`

The display name of your post type. In our example, we use “Homework.”

#### `labels`

An optional array of labels to use for describing your post type throughout the user interface:

#### `name`

The plural display name of your post type. This will overrides the `label` argument.

#### `singular_name`

The singular name for any particular post. This defaults to the `name` if not specified.

#### `add_new`

Defaults to the string “Add New.”

*add\_new\_item*

Defaults to “Add New Post.”

*edit\_item*

Defaults to “Edit Post.”

*new\_item*

Defaults to “New Post.”

*view\_item*

Defaults to “View Post.”

*search\_items*

Defaults to “Search Posts.”

*not\_found*

Defaults to “No Posts Found.”

*not\_found\_in\_trash*

Defaults to “No posts found in Trash.”

*parent\_item\_colon*

Defaults to “Parent Page:” and is only used on hierarchical post types.

*all\_items*

Defaults to “All Posts.”

*menu\_name*

The menu name for the post type, usually the same as `label` or `labels->name`.

*description*

An optional string that describes your post type.

*publicly\_queryable*

An optional Boolean that specifies whether queries on your post type can be run on the frontend or theme of your application. By default, `publicly_queryable` is turned on.

*exclude\_from\_search*

An optional Boolean that specifies whether your post type posts can be queried and displayed in the default WordPress search results. This is off by default so that your posts will be searchable.

*capability\_type*

An optional string or array. If not specifically defined, `capability_type` will default to `post`. You can pass in a string of an existing post type, and the new post type you are registering will inherit that post type's capabilities. You can also define your own capability type, which will set default capabilities for your CPT for reading, publishing, editing, and deleting. You can also pass in an array if you want to use different singular and plural words for your capabilities. For example, you can just pass in the string "homework" since the singular and plural forms for "homework" are the same, but you would pass in an array like `array( 'submission', 'submissions' )` when the forms are different.

*capabilities*

An optional array of the capabilities of the post type you are registering. You can use this instead of `capability_type` if you want more granular control over the capabilities you are assigning to your new CPT.

There are two types of capabilities: meta and primitive. Meta capabilities are tied to specific posts, whereas primitive capabilities are more general purpose. In practice, this means that when checking if a user has a meta capability, you must pass in a `$post_id` parameter:

```
//meta capabilities are related to specific posts
if(current_user_can("edit_post", $post_id))
{
    //the current user can edit the post with ID =
$post_id
}
```

Unlike meta capabilities, primitive capabilities aren't checked against a specific post:

```
//primitive capabilities aren't related to specific
posts
if(current_user_can("edit_posts"))
{
    //the current user can edit posts in general
}
```

The capabilities that can be assigned to your custom post type are as follows:

`edit_post`

A meta capability for a user to edit a particular post.

`read_post`

A meta capability for a user to read a particular post.

`delete_post`

A meta capability for a user to delete a particular post.

`edit_posts`

A primitive capability for a user to be able to create and edit posts.

*edit\_others\_posts*

A primitive capability for a user to be able to edit others' posts.

*publish\_posts*

A primitive capability for a user to be able to publish posts.

*read\_private\_posts*

A primitive capability for a user to be able to read private posts.

*read*

A primitive capability for a user to be able to read posts.

*delete\_posts*

A primitive capability for a user to be able to delete posts.

*delete\_private\_posts*

A primitive capability for a user to be able to delete private posts.

*delete\_published\_posts*

A primitive capability for a user to be able to delete posts.

*delete\_others\_posts*

A primitive capability for a user to be able to delete other people's posts.

*edit\_private\_posts*

A primitive capability for a user to be able to edit private posts.

### *edit\_published\_posts*

A primitive capability for a user to be able to publish posts.

### *map\_meta\_cap*

Whether to use the internal default meta capability handling (capabilities and roles are covered in [Chapter 6](#)). Defaults to `false`. You can always define your own capabilities using `capabilities`; but if you don't, setting `map_meta_cap` to `true` will make the following primitive capabilities be used by default or in addition to using `capability_type`: `read`, `delete_posts`, `delete_private_posts`, `delete_published_posts`, `delete_others_posts`, `edit_private_posts`, and `edit_published_posts`.

### *hierarchical*

An optional Boolean that specifies whether a post can be hierarchical and have a parent post. WordPress pages are set up like this so you can nest pages under other pages. The `hierarchical` argument is turned off by default.

### *public*

An optional Boolean that specifies if a post type is supposed to be used publicly or not in the backend or frontend of WordPress. By default, this argument is `false`; so without including this argument and setting it to `true`, you couldn't use this `post_type` in your theme. If you set `public` to `true`, it automatically sets `exclude_from_search`, `publicly_queryable` and `show_ui_nav_menus` to `true` unless otherwise specified.

Most CPTs will be public so they are shown on the frontend or available to manage through the WordPress dashboard. Other CPTs (like the default Revisions CPT) are updated behind the

scenes based on other interactions with your app and would have `public` set to `false`.

#### *rewrite*

An optional Boolean or array used to create a custom permalink structure for a post type. By default, this is set to `true`, and the permalink structure for a custom post is

`/post_type/post_title/`. If set to `false`, no custom permalink would be created. You can completely customize the permalink structure of a post by passing in an array with the following arguments:

#### *slug*

Defaults to the `post_type` but can be any string you want. Remember not to use the same slug in more than one post type because they must be unique.

#### *with\_front*

Whether or not to prepend the “front base” to the front of the CPT permalink. If set to `true`, the slug of the “front page” set on the Settings → Reading page of the dashboard will be added to the permalink for posts of this post type.

#### *feeds*

Boolean that specifies whether a post type can have an RSS feed. The default value of this argument is set to the value of the `has_archive` argument. If `feeds` is set to `false`, no feeds will be available.

#### *pages*

Boolean that turns on pagination for a post type. If `true`, archive pages for this post type will support pagination.

#### *ep\_mask*

EP, or endpoints, can be very useful. With this argument you assign an endpoint mask for a post type. For instance, we could set up an endpoint for a post type of homework called “pop-quiz.” The permalink would look like

/homework/post-title/pop-quiz/. In MVC terminology, a CPT is similar to a module, and endpoints can be thought of as different views for that module. (We cover endpoints and other rewrite functions in [Chapter 7](#).)

#### `has_archive`

An optional Boolean or string that specifies whether a post type can have an archive page. By default this argument is set to `false`, so you will want to set it to `true` if you would like to use it in your theme. The `archive-<post_type>.php` file in your theme will be used to render the archive page. If that file is not available, the `archive.php` or `index.php` file will be used instead.

#### `query_var`

An optional Boolean or string that sets the `query_var` key for the post type. This is the name of your post type in the database and used when writing queries to work with this post type. The default value for this argument is set to the value of `post_type` argument. In most cases you wouldn’t need your `query_var` and your `post_type` to be different, but you can imagine a long post type name like `directory_entry` for which you would want to use a shorter slug like “dir.”

#### `supports`

An optional Boolean or array that specifies what meta box features will be made available on the new post or edit post page. By default, an array with the arguments of `title` and `editor` is passed in. Following is a list of all available arguments (to use one of these features with your CPT, be sure it’s included in the `supports` array):

- title
- editor
- comments
- revisions
- trackbacks
- author
- excerpt
- page-attributes
- thumbnail
- custom-fields
- post-formats

*register\_meta\_box\_cb*

An optional string that allows you to provide a custom callback function for integrating your own custom meta boxes.

*permalink\_epmask*

An optional string for specifying which endpoint types you would like to associate with a custom post type. The default rewrite endpoint bitmask is EP\_PERMALINK. For more information on endpoints, see [Chapter 7](#).

*taxonomies*

An optional array that specifies any built-in (categories and tags) or custom registered taxonomies you would like to associate with a post type. By default, no taxonomies are referenced. For more on taxonomies, see [“Creating Custom Taxonomies”](#).

### *show\_ui*

An optional Boolean that specifies whether the basic post UI will be made available for a post type in the backend. The default value is set to the value of the `public` argument. If `show_ui` is `false`, you will have no way of populating your posts from the backend administration area.

#### **NOTE**

It's a good idea to set `show_ui` to `true`, even for CPTs that won't generally be added or edited through the administrator dashboard. For example, the bbPress plugin adds Topics and Replies as CPTs that are added and edited through the forum UI on the frontend. However, `show_ui` is set to `true`, providing another interface from which administrators can search, view, and manage topics and replies.

### *menu\_position*

An optional integer used to set the menu order of a post type menu item in the backend, left-side navigation.

The WordPress Codex provides a [nice list of common menu position values](#) to help you figure out where to place the menu item for your CPT:

- 5: below Posts
- 10: below Media
- 15: below Links
- 20: below Pages
- 25: below comments

- 60: below first separator
- 65: below Plugins
- 70: below Users
- 75: below Tools
- 80: below Settings
- 100: below second separator

#### *menu\_icon*

An optional string of a URL to a custom icon that can be used to represent a post type.

#### *can\_export*

An optional Boolean that specifies whether a post type can be exported via the WordPress exporter in Tools → Export. This argument is set to `true` by default, allowing the administrator to export.

#### *show\_in\_nav\_menus*

An optional Boolean that specifies whether posts from a post type can be added to a custom navigation menu in Appearance → Menus. The default value of this argument is set to the value of the `public` argument.

#### *show\_in\_menu*

An optional Boolean or string that specifies whether to show the post type in the backend admin menu and possibly where to show it. If set to `true`, the post type is displayed as its own item on the menu. If set to `false`, no menu item for the post type is shown. You can also pass in a string of the name of any other menu item. Doing this places the post type in the submenu of the passed-in

menu item. The default value of this argument is set to the value of the `show_ui` argument.

#### *show\_in\_admin\_bar*

An optional Boolean that specifies whether a post type is available in the WordPress administrator bar. The default value of this argument is set to the value of the `show_in_menu` argument.

#### *delete\_with\_user*

An optional Boolean that specifies whether to delete all the posts for a post type created by a given user. If set to `true`, posts the user created will be moved to the trash when the user is deleted. If set to `false`, posts will not be moved to the trash when the user is deleted. By default, posts are moved to the trash if the argument `post_type_supports` has `author` within it. If not, posts are not moved to the trash.

#### *show\_in\_rest*

An optional Boolean that specifies whether this CPT is exposed through the REST API. The default value of this argument is `false`. The REST API is covered in [Chapter 10](#).

#### *rest\_base*

An optional string to change the base slug that is used when accessing CPTs of this type through the REST API. The default value for this argument is the post type name set as the first parameter to the `register_post_type()` function.

#### *rest\_controller\_class*

An optional string to change the controller used when accessing this post type through the REST API. This argument should be set to `WP_REST_Posts_Controller` or the name of a PHP class that inherits the `WP_REST_Posts_Controller` class. The

default value for this argument is  
`WP_REST_Posts_Controller`.

#### \_builtin

You shouldn't ever need to use this argument. Default WordPress post types use this to differentiate themselves from CPTs.

#### \_edit\_link

The URL of the edit link on the post. This is also for internal use, and you shouldn't need to use it. If you'd like to change the page linked to when clicking to edit a post, use the `get_edit_post_link` filter, which passes the default edit link along with the ID of the post.

Example 5-1 illustrates registering new “Homework” and “Submissions” CPTs using `register_post_type()`. You can find the code for the `register_post_type()` function in `wp-includes/post.php`. Notice that in our example we are using only a few of the many available arguments.

#### Example 5-1. Registering a CPT

```
<?php
// custom function to register a "homework" post type
function schoolpress_register_post_type_homework() {
    register_post_type( 'homework',
        array(
            'labels' => array(
                'name' => __( 'Homework' ),
                'singular_name' => __(
                    'Homework'
                ),
            ),
            'public' => true,
            'has_archive' => true,
        )
    );
}
```

```

}

// call our custom function with the init hook
add_action( 'init', 'schoolpress_register_post_type_homework'
);

// custom function to register a "submissions" post type
function schoolpress_register_post_type_submission() {
    register_post_type( 'submissions',
        array(
            'labels' => array(
                'name' => __( 'Submissions' ),
                'singular_name' => __(
                    'Submission'
                ),
                'public' => true,
                'has_archive' => true,
            )
        );
}

// call our custom function with the init hook
add_action( 'init',
'schoolpress_register_post_type_submission' );
?>

```

If you dropped the preceding code in your active theme's *functions.php* file or an active plugin, you should notice two new menu items on the WordPress admin called Homework and Submissions under the Comments menu item.

### NOTE

If you get tired of writing functions to register the various custom post types that you want to use, you can use this cool plugin called [Custom Post Type UI](#).

# What Is a Taxonomy and How Should I Use It?

We briefly touched on taxonomies in Chapter 2, but what exactly is a taxonomy? Taxonomies group posts by terms. Think post categories and post tags; these are just built-in taxonomies attached to the default “post” post type. You can define as many custom taxonomies or categories as you want and span them across multiple post types. For example, we can create a custom taxonomy “Subject” that has all school-related subjects as its terms and is tied to our “Homework” CPT.

## Taxonomies Versus Post Meta

One question you will often encounter when you want to attach bits of data to posts is whether to use a taxonomy or a post meta field (or both). Generally, terms that group different posts together should be coded as taxonomies, while data specific to each individual post should be coded as post meta fields.

### *Post meta fields*

These work well for data specific to individual posts and not used to group posts together. In SchoolPress, it makes sense to code things like “required assignment length” (e.g., 500 words) as a meta field. In practice, only a few different lengths will ever be used, but we won’t ever need to “get all assignments that require 500 words.” So a post meta field is adequate for this information.

### *Taxonomies*

These are good for data that is used to group posts together. In SchoolPress, it makes sense to code things such as an

assignment's subject (e.g., math or English) as a taxonomy. Unlike assignment length, we will want to run queries like "get all math assignments." This is easily done through a taxonomy query. More important, queries like this run faster on taxonomy data than on meta fields.

Why are taxonomy queries generally faster? Meta fields are stored in the `wp_postmeta`. If we were storing an assignment's due date as a post meta field, it would look like this:

meta_id	post_id	meta_key	meta_value
1	1	due_date	2018-09-07
2	2	due_date	2018-09-14

The `meta_id`, `post_id`, and `meta_key` columns are indexed, but the `meta_value` column is not. This means that queries like "get the due date for this assignment" will run quickly, but queries like "get all assignments due on 2018-09-07" will run slower, especially if you have a large site with lots of data piled into the `wp_postmeta` table. The reason the `meta_value` key is the lone column in `wp_postmeta` without an index is that adding an index here would greatly increase both the storage required for this table and also the insert times. In practice, a site will have many different meta values, whereas there will be a smaller set of post IDs and meta keys to build indexes for.

If you stored assignment due dates in a custom taxonomy, the "get all assignments due on this date" query will run much faster. Each

specific due date would be a term in the `wp_terms` table with a corresponding entry in the `wp_terms_taxonomy` table. The `wp_terms_relationships` table that attaches terms to posts has both the `object_id` (posts are objects here) and `term_taxonomy_id` fields indexed. So “get all posts with this `term_taxonomy_id`” is a speedy query.

If you just want to show the due date on the assignment page, you should store it in the post meta fields. If you want to offer a report of all assignments due on a certain date, you should consider adding a taxonomy to track due dates.

On the other hand, due to the nature of due dates (you potentially have 365 new terms each year), using a taxonomy for them might be overkill. You would end up with a lot of useless terms in your database keeping track of which assignments were due two years ago.

Also, in this specific case, the speed increases might be negligible because the due date report is for a subset of assignments within a specific class group. In practice, we won’t be querying for assignments by due date across the entire `wp_postmeta` table. We’ll filter the query to run only on assignment posts for a specific class. While there may be millions of rows in the `wp_postmeta` table for a SchoolPress site at scale (hundreds of schools, thousands of teachers and classes), there will only be a few assignments for a specific class or group of classes one student is in.

Another consideration when choosing between meta fields and taxonomies is how that data will be managed by users. If a field is only going to be used in the backend code, and you don't have query speed issues, storing it in post meta is as simple as one call to `update_post_meta()`. If you'd like administrators to be able to create new terms, write descriptions for them, build hierarchies, and use drop-downs or checkboxes to assign them to posts, well then we've just described exactly what you get for free when you register a taxonomy. When using post meta fields, you need to build your own UI into a meta box.

Finally, I did mention earlier that there are times when you want to use both a meta field *and* a taxonomy to track one piece of data. An example of this in the context of the SchoolPress app could be tracking a textbook and chapter for an assignment. Imagine you want a report for a student with all of their assignments organized by textbook and ordered by chapter within those books. Because you want to allow teachers to manage textbooks as terms in the administrator menu, and you'll want to do queries like "get all assignments for this textbook," it makes sense to store textbooks in a custom taxonomy.

On the other hand, chapters can be stored in post meta fields. Chapters are common across books and assignments, but it doesn't make sense to query for "all chapter 1 assignments" across many different textbooks. Since we'll be able to prefilter to get all assignments by textbook or by student, we can use a chapter meta field, or possibly a `textbook_chapter` meta field with data like "PrinciplesOfMath.Ch1" to order the assignments for the report.

Phew...now that we've figured out when we'll want to use taxonomies, let's find out how to create them.

## Creating Custom Taxonomies

You can register your own taxonomies by using the function `register_taxonomy()`, found in *wp-includes/taxonomy.php*.

**`register_taxonomy( $taxonomy, $object_type, $args )`**

The `register_taxonomy()` function accepts the following three parameters:

`$taxonomy`

A required string of the name of your taxonomy. In our example, our taxonomy name is “subject.”

`$object_type`

A required array or string of the CPT(s) you are attaching this taxonomy to. In our example, we are using a string and attaching the subject taxonomy to the homework post type. We could set it to more than one post type by passing in an array of post type names.

`$args`

This is optional array of many arguments dictates how your custom taxonomy is set up. Notice that in our example we use only a few of the many available arguments that could be passed into the `register_taxonomy()` function.

Following is a list of all available arguments:

*label*

Optional string of the display name of your taxonomy.

*labels*

Optional array of labels to use for describing your taxonomy throughout the user interface:

*name*

The plural display name of your taxonomy. This will override the label argument.

*singular\_name*

The name for one object of this taxonomy. Defaults to “Category.”

*search\_items*

Defaults to “Search Categories.”

*popular\_items*

This string isn’t used on hierarchical taxonomies. Defaults to “Popular Tags.”

*all\_items*

Defaults to “All Categories.”

*parent\_item*

This string is only used on hierarchical taxonomies. Defaults to “Parent Category.”

*parent\_item\_colon*

The same as the *parent\_item* argument but with a colon at the end.

*edit\_item*

Defaults to “Edit Category.”

*view\_item*

Defaults to “View Category.”

*update\_item*

Defaults to “Update Category.”

*add\_new\_item*

Defaults to “Add New Category.”

*new\_item\_name*

Defaults to “New Category Name.”

*separate\_items\_with\_commas*

This string is used on nonhierarchical taxonomies. Defaults to “Separate tags with commas.”

*add\_or\_remove\_items*

This string is used on nonhierarchical taxonomies. Defaults to “Add or remove tags.”

*choose\_from\_most\_used*

This string is used on nonhierarchical taxonomies. Defaults to “Choose from the most used tags.”

*hierarchical*

Optional Boolean that specifies whether a taxonomy is hierarchical or if a taxonomy term may have parent terms or subterms. This is like the default categories taxonomy, and nonhierarchical taxonomies are like the default tags taxonomy. The default value for this argument is set to `false`.

*update\_count\_callback*

Optional string that works like a hook. It's called when the count of the associated post type is updated.

*rewrite*

Optional Boolean or array used to customize the permalink structure of a taxonomy. The default rewrite value is set to the taxonomy slug.

*query\_var*

Optional Boolean or string that can be used to customize the `query_var`, `?$query_var=$term`. By default, the taxonomy name is used as the `query_var`.

*public*

Optional Boolean that specifies whether a taxonomy should be used publicly in the backend or frontend of WordPress. By default, this argument is `false`; so if you didn't include this argument and set it to `true`, you couldn't use this taxonomy in your theme. If you set `public` to `true`, it automatically sets `show_ui`, `publicly_queryable`, and `show_in_nav_menus` to `true` unless otherwise noted.

*publicly\_queryable*

Optional Boolean that specifies whether the taxonomy should be publicly queryable on the frontend. The default is set to `true`.

*show\_ui*

Optional Boolean that specifies whether the taxonomy will have a backend admin UI, similar to the categories or tags interface. The default value of this argument is set to the value of the `public` argument.

*show\_in\_nav\_menus*

Optional Boolean specifying if a taxonomy will be available in navigation menus. This argument's default value is set to the value of the `public` argument.

*show\_in\_rest*

Optional Boolean that specifies whether this taxonomy is exposed through the REST API. The default value of this argument is `false`. The REST API is covered in [Chapter 10](#).

*rest\_base*

Optional string to change the base slug that is used when accessing CPTs with this taxonomy through the REST API. The default value for this argument is the taxonomy name set as the first parameter to the `register_taxonomy()` function.

*rest\_controller\_class*

Optional string to change the controller used when accessing posts with this taxonomy through the REST API. This argument should be set to `WP_REST_Terms_Controller` or the name of a PHP class that inherits that class. The default value for this argument is `WP_REST_Terms_Controller`.

*show\_tagcloud*

Optional Boolean that specifies whether the taxonomy can be included in the Tag Cloud Widget. The default value of this argument is set to the value of the `show_ui` argument.

*show\_admin\_column*

Optional Boolean that specifies whether a new column will be created for your taxonomy on the post type it is attached to on the post type's edit/list page in the backend. This is `false` by default.

## *capabilities*

Optional array of capabilities for this taxonomy with a default of none. You can pass in the following arguments and/or any custom-created capabilities:

- manage\_terms
- edit\_terms
- delete\_terms
- assign\_terms

In our homework post type example, we are going to make a taxonomy called “Subject” so we can create a term for each subject like math, science, language arts, and social studies:

```
<?php
// custom function to register the "subject" taxonomy
function schoolpress_register_taxonomy_subject() {
register_taxonomy(
    'subject',
    'homework',
    array(
        'label' => __( 'Subjects' ),
        'rewrite' => array( 'slug' => 'subject' ),
        'hierarchical' => true
    )
);
// call our custom function with the init hook
add_action( 'init', 'schoolpress_register_taxonomy_subject'
);
?>
```

Notice in the preceding code the subject taxonomy is set up like categories on a post because it's hierarchical argument is set to true.

You can create as many subjects as you would like and nest them.

Under Homework → Subjects in the backend, you can add your terms the same way you would add new categories to a post.

## **register\_taxonomy\_for\_object\_type( \$taxonomy, \$object\_type )**

What if you want to use a default taxonomy on a CPT? If you want to use the same tags taxonomy attached to the posts post type on our homework post type, you can use the

`register_taxonomy_for_object_type()` function to attach any taxonomies to any post types. The `register_taxonomy_for_object_type()` function is also located in *wp-includes/taxonomy.php*.

The `register_taxonomy_for_object_type()` function accepts two parameters:

`$taxonomy`

Required string of the name of the taxonomy.

`$object_type`

Required string of the name of the post type to which you want to attach your taxonomy.

In this example, we are attaching the default tags taxonomy to our custom homework post type:

```
<?php  
function
```

```
schoolpress_register_taxonomy_for_object_type_homework() {  
    register_taxonomy_for_object_type( 'post_tag',  
    'homework' );  
}  
add_action( 'init',  
'schoolpress_register_taxonomy_for_object_type_homework' );  
?>
```

If you run the example, notice that the “tags” taxonomy is now available under the Homework menu item. The [Custom Post Types UI plugin](#) also has a UI for creating and managing custom taxonomies.

## Using CPTs and Taxonomies in Your Themes and Plugins

In the following sections, we cover some things to keep in mind when using CPTs in your themes or plugins.

### The Theme Archive and Single Template Files

Most WordPress themes will have an *archive.php* file that renders your posts on a archive/listing page, and a *single.php* file that is responsible for rendering information about a single post. You can easily create dedicated archive and single files for your registered CPTs.

Make a copy of *archive.php* and name it *archive-homework.php*. You should now automatically have a listing archive page of all your homework assignment posts in the same format of your regular posts archive page (at *domain.com/homework/*).

You can apply the same method to the *single.php* file. Copy it and call it *single-homework.php*. You should now have a single page for each of your homework assignments (at [domain.com/homework/science-worksheet/](http://domain.com/homework/science-worksheet/)). Now you can change the markup of the CPT archive or single file to display your data differently from how your blog posts are displayed.

### NOTE

To use a custom archive file, you must set the `has_archive` argument when registering your custom post type to `true`. The `has_archive` argument is part of the `register_post_type()` function.

## Good Old `WP_Query` and `get_posts()`

In some instances, creating an archive and single *.php* file for your custom post type may not be enough for the custom functionality you require. What if you want to loop through all the posts for a specific post type in a sidebar widget or in a shortcode on a page? With `WP_Query` or `get_posts()`, you can set the `post_type` parameter to query and loop through your CPT posts the same way you would with regular posts.

In Example 5-2, we'll build a homework submission form below any content provided for the single post of the `homework` post type.

### *Example 5-2. Homework submission form*

---

```
<?php  
function schoolpress_the_content_homework_submission($content)  
{
```

```

global $post, $current_user;

// Don't do this for any other post type than homework.
if ( ! is_single() || $post->post_type != 'homework' )
    return $content;

// Don't do this if the user is not logged in.
if ( ! is_user_logged_in() )
    return $content;

// Check if the current user has already made a
submission
// to this homework assignment.
$submissions = get_posts( array(
    'post_author'      => $current_user->ID,
    'posts_per_page'   => '1',
    'post_type'        => 'submissions',
    'meta_key'         => '_submission_homework_id',
    'meta_value'       => $post->ID
) );
foreach ( $submissions as $submission ) {
    $submission_id = $submission->ID;
}

// Process the form submission if the user hasn't
already.
if ( !$submission_id &&
        isset( $_POST['submit-homework-
submission'] ) &&
        isset( $_POST['homework-submission'] )
) {

    $submission = $_POST['homework-submission'];
    $post_title = $post->post_title;
    $post_title .= ' - Submission by ' .
$current_user->display_name;
    // Insert the current users submission as a
post into our
// submissions CPT.
$args = array(
    'post_title'      => $post_title,

```

```

        'post_content' => $submission,
        'post_type'     => 'submissions',
        'post_status'   => 'publish',
        'post_author'   => $current_user->ID
    );
    $submission_id = wp_insert_post( $args );
    // Add post meta to tie this submission post
    // to the homework post.
    add_post_meta( $submission_id,
        '_submission_homework_id',
        $post->ID );
    // Create a custom message.
    $message = __(
        'Your homework has been submitted and
is
        awaiting review.',
        'schoolpress'
    );
    $message = '<div class="homework-submission-
message">' . $message .
    '</div>';
    // Drop message before the filtered $content
variable.
    $content = $message . $content;
}

// Add a link to the user's submission if a submission
was already made.
if( $submission_id ) {

    $message = sprintf( __(
        'Click %s here %s to view your
        submission to this homework
        assignment.',
        'schoolpress' ),
        '<a href="' . get_permalink(
$submission_id ) . '">',
        '</a>' );
    $message = '<div class="homework-submission-
link">' . $message .
    '</div>';
    $content .= $message;
}

```

```

// Add a basic submission form after the $content
variable being filtered.
} else {

    ob_start();
?>
<h3><?php _e( 'Submit your Homework below!', 'schoolpress' );?></h3>
<form method="post">
<?php
wp_editor( '', 'homework-submission', array(
'media_buttons' => false ) );
?>
<input type="submit" name="submit-homework-
submission" value="Submit" />
</form>
<?php
$form = ob_get_contents();
ob_end_clean();
$content .= $form;
}

return $content;
}
// Add a filter on 'the_content' so we can run our custom code
// to deal with homework submissions
add_filter( 'the_content',
'schoolpress_the_content_homework_submission', 999 );
?>
```

You have probably noticed that we haven't yet discussed the following functions:

### `ob_start()`

This PHP function is used to turn output buffering on. While output buffering is active, no output is sent to the browser; instead, the output is stored in an internal buffer.

### `wp_editor()`

This WordPress function outputs the same WYSIWYG editor that you get while adding or editing a post. You can call this function anywhere you would like to stick an editor. We thought the homework submission form would be a perfect place. We cover all the parameters of this function in [Chapter 7](#).

### `ob_get_contents()`

We set a variable called `$form` to this PHP function. This makes all content between calling the `ob_start()` function and this function into a variable called `$form`.

### `ob_end_clean()`

This PHP function clears the output buffer and turns off output buffering.

We used these functions in the previous sequence because the `wp_editor()` function does not currently have an argument to return the editor as a variable and outputs it to the browser when it's called. If we didn't use these functions, we wouldn't be able to put our editor after the `$content` variable passed into the `the_content` filter.

In the following code, we'll make sure that only administrators have access to all homework submissions and that other users can access only the homework submissions they created:

```
<?php
function schoolpress_submissions_template_redirect() {
    global $post, $user_ID;

    // only run this function for the submissions post type
    if ( $post->post_type != 'submissions' )
        return;
```

```

// check if post_author is the current user_ID
if ( $post->post_author == $user_ID )
    $no_redirect = true;

// check if current user is an administrator
if ( current_user_can( 'manage_options' ) )
    $no_redirect = true;

// if $no_redirect is false redirect to the home page
if ( ! $no_redirect ) {
    wp_redirect( home_url() );
    exit();
}

// Use the template_redirect hook to call a function that
decides if the
// current user can access the current homework submission.
add_action( 'template_redirect',
'schoolpress_submissions_template_redirect' );
?>

```

## Metadata with CPTs

You can utilize the same post meta functions we went over in detail in Chapter 2 with any CPT you create. Getting, adding, updating, and deleting post metadata is consistent across all posts types.

If you registered a CPT and added custom-fields in the `supports` argument, by default, when adding a new post or editing a post of that post type, you will see a meta box called “Custom Fields.” You may already be familiar with this meta box; it’s a very basic form used to maintain metadata attached to a post.

## NOTE

If you don't see the Custom Fields section on the edit post page, you may need to enable it. If you are using the Block Editor, click the More Tools and Options button (the three dots in the upper right). At the bottom of the page, click Options and then, in the Advanced Panels section, find the checkbox for Custom Fields. If you're using the Classic Editor, at upper right, click the Screen Options tab, then find the Custom Fields checkbox.

What if you require a slicker UI for adding metadata on the backend? Well, building a custom meta box would be the solution for you.

**`add_meta_box( $id, $title, $callback, $screen, $context, $priority, $callback_args )`**

The `add_meta_box()` function will add a meta box to one or more screens:

*\$id*

A required string of a unique identifier for the meta box you are creating.

*\$title*

A required string of the title or visible name of the meta box you are creating.

*\$callback*

A required string of a function name that's called to output the HTML inside the meta box you are creating.

*\$screen*

The optional screen or screens where your meta box will appear. Accepts a screen ID, `WP_Screen` object, or array of screen IDs. The default is null.

#### `$context`

An optional string of the context within the page where your meta box should show (normal, advanced, side). The default is advanced.

#### `$priority`

An optional string of the priority within the context where the boxes should show (high, low).

#### `$callback_args`

An optional array of arguments that will be passed in the callback function you referenced with the `$callback` parameter. Your callback function will automatically receive the `$post` object and any other arguments you set here.

### NOTE

While we focus here on how to add meta boxes to the edit post screen, meta boxes can be used on any admin screen. To use meta boxes on your own page, use a unique value for the `screen` parameter and then call `do_meta_boxes()` for that screen in your page's callback function. This is good for reporting pages or other pages where you'd like users to be able to hide boxes, rearrange boxes, or add new boxes through custom hooks.

In [Example 5-3](#), we are going to build a custom meta box for all posts of our homework post type. This meta box will contain a checkbox indicating whether a homework submission is required as well as a date selector for the homework assignment's due date.

### *Example 5-3. Custom meta box*

---

```
<?php
// Callback for adding a custom meta box.
function schoolpress_homework_add_meta_boxes() {

    add_meta_box(
        'homework_meta',
        'Additonal Homework Info',
        'schoolpress_homework_meta_box',
        'homework',
        'side'
    );

}

// Use the add_meta_boxes hook to call a custom function to
// add a new meta box.
add_action( 'add_meta_boxes',
'schoolpress_homework_add_meta_boxes' );

// This is the callback function called from add_meta_box.
function schoolpress_homework_meta_box( $post ){
    // Using 2 liens here so the url will fit in the book ;
    $smoothness_url = 'http://ajax.googleapis.com/ajax/libs/';
    $smoothness_url.=
'jqueryui/1.12.1/themes/smoothness/jquery-ui.css';

    // Enqueue jquery date picker.
    wp_enqueue_script( 'jquery-ui-datepicker' );
    wp_enqueue_style( 'jquery-style', $smoothness_url );

    // Set metadata if already exists.
    $is_required = get_post_meta( $post->ID,
        '_schoolpress_homework_is_required', 1 );

    $due_date = get_post_meta( $post->ID,
        '_schoolpress_homework_due_date', 1 );
    // Output metadata fields.
    ?>
<p>
<input type="checkbox"
    name="is_required" value="1" <?php checked( $is_required,
    '1' ); ?>>
```

```
This assignment is required.  
</p>  
<p>  
Due Date:  
<input type="text"  
name="due_date" id="due_date" value="<?php echo  
$due_date;?>">  
</p>  
<script>  
// Attach jQuery date picker to our due_date field.  
jQuery(document).ready(function() {  
    jQuery('#due_date').datepicker({  
        dateFormat : 'mm/dd/yy'  
    });  
});  
</script>  
<?php  
}  
  
// Callback for saving custom metadata to the database.  
function schoolpress_homework_save_post( $post_id ) {  
  
    // Don't save anything if WP is auto saving.  
    if ( defined( 'DOING_AUTOSAVE' ) && DOING_AUTOSAVE )  
        return $post_id;  
  
    // Check if correct post type and that the user has correct  
    permissions.  
    if ( 'homework' == $_POST['post_type'] ) {  
  
        if ( ! current_user_can( 'edit_page', $post_id ) )  
            return $post_id;  
  
    } else {  
  
        if ( ! current_user_can( 'edit_post', $post_id ) )  
            return $post_id;  
    }  
  
    // Update homework metadata.  
    update_post_meta( $post_id,  
        '_schoolpress_homework_is_required',  
        $_POST['is_required']  
    );
```

```
update_post_meta( $post_id,
    '_schoolpress_homework_due_date',
    $_POST['due_date']
);

}

// Call a custom function to handle saving our metadata.
add_action( 'save_post', 'schoolpress_homework_save_post' );
?>
```

If you are a good developer, you’re probably thinking to yourself: where are the nonces? How come these `$_POST` values aren’t sanitized? If you aren’t thinking this, you should be, because security is very important! If you don’t know what we are talking about, see [Chapter 8](#), where will cover these best practices (and nonces) in more detail. We deliberately left out this additional code to keep the example short and sweet, but know that when you are writing custom code, you should always use nonces and sanitize your data.

### NOTE

When creating meta boxes and custom meta fields, we recommend utilizing the [CMB2 plugin](#). You can easily include CMB2 in your theme or any custom plugin to give you a fast and easy way to create custom meta boxes and the meta fields inside them.

## Using Meta Boxes with the Block Editor

The `add_meta_box()` function works with both the Block Editor and the Classic Editor. In both cases, meta boxes added with the “side” context will show up in the right sidebar of the edit post screen. With the Classic Editor, all sidebar meta boxes are always

visible.<sup>2</sup> With the Block Editor, sidebar meta boxes appear under the Document tab in the right sidebar.

With both versions of the editor, meta boxes added with the normal or advanced contexts will show up below the main editor area.

Because you can also use blocks to update post meta, some meta boxes may work better as a block instead. We cover blocks more fully in [Chapter 11](#), but following are some questions to ask yourself when deciding whether a certain field or function should be developed as a Custom Block Type or a meta box. The answers here are not hard-and-fast rules, but should be read as a suggestion to lean toward one implementation over another.

## **DOES THIS METADATA NEED TO BE SET FOR EVERY POST OF THIS TYPE?**

Use a meta box with the “normal” context to show up below the post editor. Alternatively, you can use a block along with a block template so every new post contains the block by default. More on block templates in [Chapter 11](#).

## **WILL THE CONTROLS FOR THIS METADATA FIT IN THE SIDEBAR?**

If not, use a meta box with the “normal” context to show up below the post editor. There will be a larger area for your forms and fields to fit into. Paid Memberships Pro’s “Require Membership” meta box, with its single set of checkboxes, fits nicely into the sidebar.

WooCommerce’s pricing fields, with tabs of their own, fit better below the post body.

## **DOES THIS METADATA NEED TO BE PLACED WITHIN THE POST CONTENT?**

Use a Custom Block Type so you can position the block within the post body.

## **CAN USERS ADD MULTIPLE COPIES OF THIS METADATA TO THE POST?**

Use a Custom Block Type so you can add multiple copies to the same post. Alternatively, you can use a meta box below the post along with PHP and JavaScript code to add extra fields on demand. A version of this is shown in Chapter 9.

## **Custom Wrapper Classes for CPTs**

CPTs are just posts. So you can use a call like `get_post( $post_id )` to get an object of the `WP_Post` class to work with.

For complex CPTs, it helps to create a wrapper class so you can interact with your CPT in a more object-oriented way.

The basic idea is to create a custom-defined PHP class that includes as a property a post object generated from the ID of the CPT post. In addition to storing that post object, the wrapper class also houses methods for all of the functionality related to that CPT.

Example 5-4 shows the outline of a wrapper class for our homework CPT.

---

*Example 5-4. Homework CPT wrapper class*

```

<?php
/*
    Class wrapper for homework CPT
    /wp-content/plugins/schoolpress/classes/class.homework.php
*/
class Homework {
    // Constructor can take a $post_id.
    function __construct( $post_id = NULL ) {
        if ( !empty( $post_id ) )
            $this->getPost( $post_id );
    }

    // Get the associated post and prepopulate some
    // properties.
    function getPost( $post_id ) {
        //get post
        $this->post = get_post( $post_id );

        //set some properties for easy access
        if ( !empty( $this->post ) ) {
            $this->id = $this->post->ID;
            $this->post_id = $this->post->ID;
            $this->title = $this->post->post_title;
            $this->teacher_id = $this->post->post_author;
            $this->content = $this->post->post_content;
            $this->required = $this->post-
>_schoolpress_homework_is_required;
            $this->due_date = $this->post->due_date;
        }

        // Return post id if found or false if not.
        if ( !empty( $this->id ) )
            return $this->id;
        else
            return false;
    }
}
?>

```

The constructor of this class can take a \$post\_id as a parameter and will pass that to the getPost() method, which attaches a

\$post object to the class instance and also prepopulates a few properties for easy access. Example 5-5 shows how to instantiate an object for a specific homework assignment and print out the contents.

*Example 5-5. Get and print a specific homework assignment*

---

```
$assignment_id = 1;
$assignment = new Homework($assignment_id);
echo '<pre>';
print_r($assignment);
echo '</pre>';
//Outputs:
/*
Homework Object
(
    [post] => WP_Post Object
        (
            [ID] => 1
            [post_author] => 1
            [post_date] => 2013-03-28 14:53:56
            [post_date_gmt] => 2013-03-28 14:53:56
            [post_content] => This is the assignment...
            [post_title] => Assignment #1
            [post_excerpt] =>
            [post_status] => publish
            [comment_status] => open
            [ping_status] => open
            [post_password] =>
            [post_name] => assignment-1
            [to_ping] =>
            [pinged] =>
            [post_modified] => 2013-03-28 14:53:56
            [post_modified_gmt] => 2013-03-28 14:53:56
            [post_content_filtered] =>
            [post_parent] => 0
            [guid] => http://schoolpress.me/?p=1
            [menu_order] => 0
            [post_type] => homework
            [post_mime_type] =>
            [comment_count] => 3
            [filter] => raw
            [format_content] =>
        )
)
```

```
[id] => 1
[post_id] => 1
[title] => Assignment 1
[teacher_id] => 1
[content] => This is the assignment...
[required] => 1
[due_date] => 2013-11-05
)
*/
```

## Extending WP\_Post Versus Wrapping It

Another option here would be to extend the `WP_Post` class, but currently this is not possible, as the `WP_Post` class is defined as final, meaning it is a class that can't be extended. The WordPress core team has said they're doing this to keep people from building plugins that rely on extending the `WP_Post` object since `WP_Post` is due for overhaul in future versions of WordPress. We think they're being big fuddy duddies.<sup>3</sup>

In Chapter 6, we extend the `WP_User` class (which isn't defined as final). But the best we can do with `WP_Post` is to create a wrapper class for it.

## Why Use Wrapper Classes?

Building a wrapper class for your CPT is a good idea for a few reasons:

- You can put all your code to register the CPT in one place.
- You can put all your code to register related taxonomies in one place.

- You can build all your CPT-related functionality as methods on the wrapper class.
- Your code will read better.

## Keep Your CPTs and Taxonomies Together

Put all of your code to register the CPT and taxonomies in one place. Instead of having one block of code to register a CPT and define the taxonomies and a separate class wrapper to handle working with the CPT, you can place your CPT and taxonomy definitions into the class wrapper itself:

```
/*
    Class wrapper for homework CPT with init function
    /wp-
content/plugins/schoolpress/classes/class.homework.php
*/
class Homework
{
    // Constructor can take a $post_id.
    function __construct($post_id = NULL)
    {
        if (!empty($post_id))
            $this->getPost($post_id);
    }

    // Get the associated post and prepopulate some
properties.
    function getPost($post_id)
    {
        /* snipped */
    }

    // Register CPT and taxonomies on init.
    function init()
    {
        // Register the homework CPT.
        register_post_type(
```

```

        'homework',
        array(
            'labels' => array(
                'name' => __(
                    'Homework' ),
                    'singular_name' =>
                    __( 'Homework' )
                ) ,
                'public' => true,
                'has_archive' => true,
            )
        );
    }

    // Register the subject taxonomy.
    register_taxonomy(
        'subject',
        'homework',
        arrayarraytruearray

```

This code has been snipped (the full version can be found on [this book's GitHub site](#)), but shows how you would add an `init()` method to your class that is hooked into the `init` action. The `init()` method then runs all the code required to define the CPT. You could also define other hooks and filters here, with the callbacks linked to other methods in the `Homework` class.

You can organize things in other ways, but we find that having all of your CPT-related code in one place helps a lot.

## Keep It in the Wrapper Class

Build all of your CPT-related functionality as methods on the wrapper class. When we registered our homework CPT, a page was added to the dashboard allowing us to “Edit Homework.” Teachers can create homework like any other post, with a title and body content. Teachers can publish the homework when it’s ready to be pushed out to students. All of this post-related functionality is available for free when you create a CPT.

On the other hand, much of the functionality around many CPTs, including our homework CPT, needs to be coded up. With a wrapper class in place, this functionality can be added as methods of our Homework class.

For example, one thing we want to do with our homework posts is to gather all the submissions for a particular assignment. Once we do this, we can render them in a list or process them in some way.

Example 5-6 shows a couple of methods we can add to our Homework class to gather related submissions and to calculate a flat-scale grading curve.

### *Example 5-6. Adding methods to the Homework class*

---

```
<?php  
class Homework  
{  
    /* Snipped constructor and other methods from earlier  
examples */
```

```

/*
     Get related submissions.
     Set $force to true to force the method to get
children again.
*/
function getSubmissions($force = false)
{
    // Need a post ID to do this.
    if(empty($this->id))
        return array();

    // Did we get them already?
    if(!empty($this->submissions) && !$force)
        return $this->submissions;

    // Okay get submissions.
    $this->submissions = get_children(array(
        'post_parent' => $this->id,
        'post_type' => 'submissions',
        'post_status' => 'published'
    ));

    // Make sure submissions is an array at least.
    if(empty($this->submissions))
        $this->submissions = array();

    return $this->submissions;
}

/*
     Calculate a grade curve
*/
function doFlatCurve($maxscore = 100)
{
    $this->getSubmissions();

    // Figure out the highest score.
    $highscore = 0;
    foreach($this->submissions as $submission)
    {
        $highscore = max($submission->score,
$highscore);
}

```

```

        }

        // Figure out the curve.
        $curve = $maxscore - $highscore;

        // Fix lower scores.
        foreach($this->submissions as $submission)
        {
            update_post_meta(
                $submission->ID,
                "score",
                min( $maxscore, $submission-
>score + $curve )
            );
        }
    }
?>

```

## Wrapper Classes Read Better

In addition to organizing your code to make things easier to find, working with wrapper classes also makes your code easier to read and understand. With fully wrapped Homework and Submission CPTs and special user classes (covered in [Chapter 6](#)), code like the following is possible:

```

<?php
// Use a method of the Student class to check if the current
user is a student
if ( Student::is_student() ) {
    // Student defaults to current user.
    $student = new Student();

    // Let's figure out when their next assignment is due.
    $assignment = $student->getNextAssignment();

    // Display info and links.
    if ( !empty( $assignment ) ) {

```

```

?>
<p>Your next assignment
<a href="<?php echo get_permalink( $assignment->id );?>">
<?php echo $assignment->title;?></a>
for the
<a href="<?php echo get_permalink( $assignment->class_id
);?>">
<?php echo $assignment->class->title;?></a>
class is due on <?php echo $assignment->getDueDate();?>.
</p>
<?php
}
?
}
?>

```

The code would be much more complicated if all the `get_post()` calls and loops through arrays of child posts were out in the open. Using an object-oriented approach makes your code more accessible to other developers working it.

- <sup>1</sup> The new privacy controls were motivated by the EU GDPR, passed in 2018. The regulation applies not just to all EU residents, but also all sites with EU visitors. So pretty much all sites. The idea of allowing users to export or delete noncritical personal data is a good one no matter where you live.
- <sup>2</sup> Although, clicking the heading of a meta box, no matter where it is displayed, hides or shows the full meta box content.
- <sup>3</sup> But seriously, the core team is really smart and makes a good point. If someone extended the `WP_Post` class and created new properties and methods for their new class, it would cause trouble if WordPress core later chose to use properties and methods with the same names in the base `WP_Post` class.

# Chapter 6. Users, Roles, and Capabilities

---

In [Chapter 1](#), we established logins as a crucial component of any web app. One of the great things about using WordPress for your apps is that you get fully featured user management out of the box. The core WordPress app includes:

- Secure logins with passwords that are salted and hashed
- User records with an email address, username, display name, avatar, and bio
- Administrator views to browse, search, add, edit, and delete users
- User roles to separate administrators from editors, authors, contributors, and subscribers
- Pages for users to log in, register, and reset passwords

By using various WordPress functions and APIs, we can do the following:

- Add and manage user meta or profile fields for each user.
- Define custom roles and capabilities for finer control over which users can access which areas.

Managing users in WordPress is a fairly straightforward affair. The User tab in the dashboard makes it easy to browse, search, add, edit,

and delete users. It's easy to manage users via code as well.

This chapter will cover these areas:

- How to access user data in your code
- How to add custom fields to users
- How to customize the user profiles and reports in the dashboard
- How to add, update, and delete users
- How to define custom roles and capabilities
- How to extend the WordPress `User` class to create your own user-focused classes

## Getting User Data

In this section, we'll explore how to instantiate a WordPress user object in code and how to get basic user information, like login and email address, and user metadata out of that object.

The `WP_User` class is the workhorse for managing WordPress users in code. Just like anything else in WordPress and PHP, there are a few different ways to get a `WP_User` object to work with. Here are some of the most popular methods:

```
// get the WP_User object WordPress creates for the
// currently logged-in user
global $current_user;

// get the currently logged-in user with the
wp_get_current_user() function
$user = wp_get_current_user();
```

```
// set some variables
$user_id = 1;
$username = 'jason';
$email = 'jason@strangerstudios.com';

// get a user by ID
$user = wp_getuserdata( $user_id );

// get a user by another field
$user1 = wp_get_user_by( 'login', $username );
$user2 = wp_get_user_by( 'email', $email );

// use the WP_User constructor directly
$user = new WP_User( $user_id );

//use the WP_User constructor with a username
$user = new WP_User( $username );
```

Once you have a `WP_User` object, you can get any piece of user data you want:

```
// get the currently logged-in user
$user = wp_get_current_user();

// echo the user's display_name
echo $user->display_name;

// use user's email address to send an email
wp_mail( $user->user_email, 'Email Subject', 'Email Body' );

// get any user meta value
echo 'Department: ' . $user->department;
```

You can access data stored in the `wp_users` table (`user_login`, `user_nicename`, `user_email`, `user_url`, `user_registered`, `user_status`, and `display_name`) by using the arrow operator—for example, `$user->display_name`.

You can also access any value in the `wp_usermeta` table by using the arrow operator—for example, `$user->meta_key`—or by using the `get_user_meta()` function. These two lines of code produce the same result:

```
<?php
$full_name = trim( $user->first_name . ' ' . $user-
>last_name );
$full_name = trim( get_user_meta( $user->ID, 'first_name' )
.
' ' . get_user_meta( $user->ID, 'last_name' ) );
?>
```

It's useful to understand the trick WordPress is using to allow you to access user meta on demand as if each meta field were a property of the `WP_User` class. The `WP_User` class is using overloaded properties or the `__get()` “magic method.”<sup>1</sup>

With magic methods, any property of the `WP_User` object that you try to get that isn't an actual property of the object will be passed to the `__get()` method of the class. Here is a simplified<sup>2</sup> version of the `__get()` method used in the `WP_User` class:

```
function __get( $key ) {
    if ( isset( $this->data->$key ) ) {
        $value = $this->data->$key;
    } else {
        $value = get_user_meta( $this->ID, $key,
    true );
    }

    return $value;
}
```

Let's analyze this. The method first checks if a value exists in the `$data` property of the `WP_User` object. If so, that value is used. If not, the method uses the `get_user_meta()` function to see if any meta value exists using the key passed in.

Because we're loading meta values on demand in this way, there is less memory overhead when instantiating a new `WP_User` object. On the other hand, because meta values aren't available until you specifically ask for them, you can't dump all metadata on a user using code like `print_r( $user )` or `print_r( $user->data )`.

To loop through all the metadata for a user, use the `get_user_meta()` function with no `$key` parameter passed in:

```
// dump all metadata for a user
$user_meta = get_user_meta( $user_id );
foreach( $user_meta as $key => $value )
    echo $key . ':' . $value . '<br />';
```

Knowing how WordPress uses the `__get()` function is interesting, but it's also important so you can avoid a couple of limitations of this magic method.

The `__get()` and `__set()` methods are not called when assignments are chained together. For example, the code `$year = $user->graduation_year = '2012'` would produce inconsistent results.

Similarly, `__get()` is not called when coded within an `empty()` or `isset()` function call. So `if(empty($user->graduation_year))` will also be `false`, even if there exists some user meta with the key `graduation_year`.

The solution to these two issues is to become a little more verbose with your code:

```
// Split assignments into multiple lines when using magic
methods.
$user->graduation_year = '2012';
$year = '2012';

//To test if a meta value is empty, set a local variable
first.
$year = $user->graduation_year;
if ( empty( $year ) )
    $year = '2012';
```

## Add, Update, and Delete Users

We touched on some basic functions for adding, updating, and deleting users in [Chapter 2](#), but as working with user data is such an important part of any web application, we present a brief overview here with some additional examples and different use case scenarios.

Occasionally, you will need to add users through code instead of using the WordPress dashboard. In our SchoolPress app, we might want to allow teachers to enter a list of email addresses and generate a user for each email entered.

Or maybe you want to customize the registration process. The built-in WordPress registration form is difficult to customize. It's often easier to build your own form and use WordPress functions to add the user yourself on the backend.<sup>3</sup>

As you should already know, the function for adding a user to WordPress is `wp_insert_user()`, which takes an array of user data and inserts it into the `wp_users` and `wp_usermeta` tables:

```
// insert user from values we've gathered
$user_id = wp_insert_user( array(
    'user_login' => $username,
    'user_pass' => $password,
    'user_email' => $email,
    'first_name' => $firstname,
    'last_name' => $lastname
)
);

// check if username or email has already been used
if ( is_wp_error( $user_id ) ){
    echo $return->get_error_message();
} else {
    // continue on with whatever you want to do with the new
$user_id
}
```

The following code automatically logs someone in after adding that person's user. The `wp_signon()` function authenticates the user and sets up the secure cookies to log in the user:

```
// okay, log them in to WP
$creds = array();
$creds['user_login'] = $username;
$creds['user_password'] = $password;
```

```
$creds['remember'] = true;  
$user = wp_signon( $creds, false );
```

Updating users is as easy as adding them with the `wp_update_user()` function. You pass in an array of user data and metadata. As long as there is an ID key in the array with a valid user ID as the value, WordPress will set any specified user values:

```
// this will update a user's email and leave other values  
// alone  
$userdata = array( 'ID' => 1, 'user_email' =>  
    'jason@strangerstudios.com' );  
wp_update_user( $userdata );  
  
// this function is also perfect for updating multiple user  
// meta fields at once  
wp_update_user( array(  
    'ID' => 1,  
    'company' => 'Stranger Studios',  
    'title' => 'CEO',  
    'personality' => 'crescent fresh'  
) );
```

### NOTE

You can't update a user's `user_login` through `wp_update_user`. Also, if a user's `user_pass` is updated, the user will be logged out. You can use the preceding autologin code to log the user back in using the new password.

You can also update one user meta value at a time using the following function:

```
update_user_meta( $user_id, $meta_key, $meta_value,  
    $prev_value )
```

The following code segments illustrate some more features:

```
// arrays will get serialized
$children = array( 'Isaac', 'Marin' );
update_user_meta( $user_id, 'children', $children );

// you can also store an array by storing multiple values
// with the same key
add_user_meta( $user_id, 'children', 'Isaac' );
add_user_meta( $user_id, 'children', 'Marin' );

// when storing multiple values, specify the $prev_value
// parameter
// to select which one changes
update_user_meta( $user_id, 'children', 'Isaac Ford',
'Isaac' );
update_user_meta( $user_id, 'children', 'Marin Josephine',
'Marin' );

//delete all user meta by key
delete_user_meta( $user_id, 'children' );

//delete just one row when there are multiple values for one
key
delete_user_meta( $user_id, 'children', 'Isaac Ford' );
```

Note that in the code, we show two different ways to store arrays in user meta. This is similar to storing options via `update_option()` or posting meta via `update_post_meta()`. The first method (one serialized value per key) keeps row count down on the `wp_usermeta` table, which can make queries by `meta_key` faster. The second method (multiple values per key) allows you to query by `meta_value`. For example, storing child names as separate user meta entries lets you do queries like this:

```
<?php
// get the IDs of all users with children named Isaac
$parents_of_isaac = $wpdb->get_col( "SELECT user_id
    FROM $wpdb->usermeta
    WHERE meta_key = 'children'
    AND meta_value = 'Isaac'" );
?>
```

Although it's possible to query the `wp_usermeta` and `wp_postmeta` tables by `meta_value`, be careful about query times. The `meta_value` column is not indexed, so queries against large datasets may be slow. Many-to-one relationships like this can also be stored in custom taxonomies, which can show better performance.

Deleting a user in code, though dangerous, is incredibly easy to do:

```
//this file contains wp_delete_user and is not always
loaded, so let's make sure
require_once(ABSPATH . '/wp-admin/includes/user.php');

//delete the user
wp_delete_user( $user_id );

//or delete a user and reassign their posts to user with ID
#1
wp_delete_user( $user_id, 1 );
```

For network site setups, you will need to use the `wpmu_delete_user()` function to delete the user from the entire network. Otherwise `wp_delete_user()` just deletes the user from the current blog. You can use the `is_multisite()` function to detect which function should be used:

```
// I want to make sure we really delete the user.
if ( is_multisite() )
```

```
wp_delete_user( $user_id );
else
    wpmu_delete_user( $user_id );
```

## Hooks and Filters

Perhaps more common than adding and updating user data yourself are scenarios in which you want to do some other bit of code when new users are added or deleted. For example, you may want to create and link a new CPT post to a user when they register. Or maybe you want to clean up connections and data stored in custom tables when a user is deleted. You can do this through some user-related hooks and filters.

The hook to run code after a user is registered is `user_register`. The hook passes in the user ID of the newly created user:

```
//create a new "course" CPT when a teacher registers
function sp_user_register( $user_id ){
    // check if the new user is a teacher (see chapter 15
    for details)
    if ( pmpro_hasMembershipLevel( 'teacher', $user_id ) ) {
        // add a new "course" CPT with this user as author
        wp_insert_post( array(
            'post_title' => 'My First Course',
            'post_content' => 'This is a sample course...',
            'post_author' => $user_id,
            'post_status' => 'draft',
            'post_type' => 'course'
        ) );
    }
}
add_action( 'user_register', 'sp_user_register' );
```

The hook to run code just *before* deleting a user is `delete_user`. A similar hook `deleted_user` (note the past tense) runs just *after* a user has been deleted.

These hooks are mostly interchangeable, but there are a couple of things to note:

- If you hook on `delete_user` early enough, you might be able to abort the user delete.
- If you hook on `deleted_user`, some user data and connections may already be gone and unavailable:

```
<?php
// send an email when a user is being deleted
function sp_delete_user( $user_id ){
    $user = get_userdata( $user_id );
    wp_mail( $user->user_email,
        "You've been deleted.",
        'Your account at SchoolPress has been deleted.'
    );
}
// want to be able to get user_email so hook in early
add_action( 'delete_user', 'sp_delete_user' );
?>
```

## What Are Roles and Capabilities?

Roles and capabilities are how WordPress controls what users have access to view and do on your site. Each user can have one role, and each role can have one or many capabilities. Each capability determines if a user can or can't view a certain type of content or perform a certain action.

There are five default roles in every WordPress installation: Administrator, Editor, Author, Contributor, and Subscriber. If you are running a network site, you'll have a sixth role, Super Administrator, which has admin access to all sites on the network. A full list of capabilities and how they map to the default WordPress roles can be found on the [WordPress Support Roles and Capabilities page](#).

In a little bit, we go over how to create new roles outside the WordPress defaults. However, for most apps it makes sense to stick to the default roles: have your app administrators use the Administrator role and have all of your users/customers use the Subscriber role.

If your app users will be creating content, consider making them Authors (can create and publish posts) or Contributors (can create but not publish posts). If your app has moderators, consider making them Editors.

Using the default roles is a good idea because certain plugins will expect your users to have one of these roles. If your admins are really users in an office manager role, you may need to get a third-party plugin to work with those users. The opposite is sometimes true as well. You might need to hide functionality made available to your users based on the roles they have, especially if you are using roles outside of Administrators (can access everything) and Subscribers (can just view stuff).

## Checking a User's Role and Capabilities

At times you'll need to check whether a user can do something before you let them do it. This can be done with the `current_user_can()` function. This function takes one parameter, which is a string value for the \$capability to check. The following code illustrates the usage of this function:

```
if ( current_user_can( 'manage_options' ) ) {
    // has the manage options capability, typically an admin
}

if ( current_user_can( 'edit_user', $user_id ) ) {
    // can edit the user with ID = $user_id.
    // typically either the user himself or an admin
}

if ( current_user_can( 'edit_post', $post_id ) ) {
    // can edit the post with ID = $post_id.
    // typically the author of the post or an admin or
editor
}

if ( current_user_can( 'subscriber' ) ) {
    // one way to check if the current user is a subscriber
}
```

You can also use the function `user_can()` to check whether someone other than the current user has a capability. Pass in the `$user_id` of the user you want to check, the capability, and any other arguments needed:

```
<?php
/*
    Output comments for the current post,
    highlighting anyone who has capabilities to edit it.
*/
global $post;    // current post we are looking at

$comments = get_comments( 'post_id=' . $post->ID );
```

```

foreach( $comments as $comment ) {
    // default CSS classes for all comments
    $classes = 'comment';

    // add can-edit CSS class to authors
    if ( user_can( $comment->user_id, 'edit_post', $post->ID
) ) {
        $classes .= ' can-edit';
    }
?>
<div id="comment-<?php echo $comment->comment_ID;?>">
    class="<?php echo $classes;?>">
        Comment by <?php echo $comment->comment_author; ?>:
        <?php echo wpautop( $comment->comment_content ); ?>
    </div>
<?php
}

```

Though it is possible to check for a user's role using `current_user_can()`, it's better practice to test a user's capabilities instead of the user's role. This allows your code to continue to work even if users are given different roles or roles are assigned different capabilities. For example, checking for `manage_options` will work as you intend whether the user is an Administrator or a custom role with the `manage_options` capability added.

Testing a user's role should be limited to cases in which you really need to know the user's role instead of a capability. If you find yourself checking for someone's role before performing certain actions, you should take it as a hint that you need to add a new capability.

The following is a function to upgrade any Subscriber whose ID is passed in to the Author role. To be extra sure, we check the `roles` array of the user object instead of using the `user_can()` function. We use the `set_role()` method of the user class to set the new role:

```
function upgradeSubscriberToAuthor( $user_id ) {  
    $user = new WP_User( $user_id );  
    if ( in_array( 'subscriber', $user->roles ) ) {  
        $user->set_role( 'author' );  
    }  
}
```

## Creating Custom Roles and Capabilities

As we said earlier, it's a good idea to stick with the default WordPress roles if possible. However, if you have different classes of users and need to restrict what they are doing in new ways, adding custom roles and capabilities is the way to do it.

In our SchoolPress app, teachers are Authors and students are Subscribers. However, we do need a custom role for office managers who can manage users but cannot edit content, themes, options, plugins, or the general WordPress settings. We can set up the Office Manager role like so:

```
function sp_roles_and_caps() {  
    // Office Manager Role  
    remove_role('office');           // in case we updated the  
    caps below  
    add_role( 'office', 'Office Manager', array(  
        'read' => true,  
        'create_users' => true,
```

```
        'delete_users' => true,
        'edit_users' => true,
        'list_users' => true,
        'promote_users' => true,
        'remove_users' => true,
        'office_report' => true // new cap for our custom
report
    );
}
// run this function on plugin activation
register_activation_hook( __FILE__, 'sp_roles_and_caps' );
```

When the `add_role()` function is run, it updates the `wp_user_roles` option in the `wp_options` table, where WordPress looks to get information on roles and capabilities. So you want to run this function only once upon activation instead of every time at runtime. That's why we register this function using `register_activation_hook()`.

We also run `remove_role('office')` at the start there, which is useful if you want to delete a role completely, but is also useful to clear out the “office” role before adding it again in case you edited the capabilities or other settings for the role. Without the `remove_role()` line, the `add_role()` line will not run since the role already exists.

The `add_role()` function takes three parameters: a role name, a display name, and an array of capabilities. Use the [WordPress Support site](#) to find the names of the default capabilities or look them up in the `/wp-admin/includes/schema.php` file of your WordPress installation.

## NOTE

WordPress only runs an activation hook when a plugin is activated. It won't run when a plugin is updated or when you just change one of the PHP files in the plugin. To get new user roles and capabilities to stick, you may need to deactivate and reactivate a plugin. If you are creating a publicly available plugin, you can check the role and capability settings on the `admin_init` hook and reset the roles and capabilities if needed. Paid Memberships Pro has posted to GitHub [the file](#) showing how this is done.

Adding new capabilities is as simple as including a new capability name in the `add_role()` call or using the `add_cap()` method on an existing role. Here is an example showing how to get an instance of the role class and add a capability to it:

```
// give admins our office_report cap to let them view that report
$role = get_role( 'administrator' );
$role->add_cap( 'office_report' );
```

Again, this code needs to run only once, which will save it in the database. Put code like this inside a function registered via `register_activation_hook()`, as in the last example.

You can also use the `remove_cap()` method of the role class, which is useful if you want to remove some functionality from the default roles. For example, the following code will remove the `edit_pages` capabilities from Editors so they can edit any blog post, but no pages (post of type “page”):

```
// don't let editors edit pages
$role = get_role( 'editor' );
```

```
$role->remove_cap( 'edit_pages' );
```

You can do some powerful things by adding and editing roles and capabilities. Defining what users have access to view and do is an important part of building an app. Different roles can be built for different membership levels or upgrades associated with your product. [Chapter 15](#) introduces the Paid Memberships Pro plugin, which adds “membership levels” as a separate classification for your users. This can be used in place of custom roles, but more often is used in conjunction with them. For details on how membership levels and roles can work together, see [Chapter 15](#).

## Extending the WP\_User Class

Similar to how we wrapped the `WP_Post` class in [Chapter 5](#) to create a more specific class for our custom post types, we can extend the `WP_User` class to create useful classes that will help us organize our code related to different types of users.

For example, in our SchoolPress app, we have two main user types: Teachers and Students.<sup>4</sup> Both types are at their core just WordPress users, but each type of user will also have functionality unique to them. We can encapsulate that unique functionality by writing Teacher and Student classes that extend the `WP_User` class.

Wouldn’t it be great if we could write code like this?

```
<?php  
// Student is a class that extends WP_User  
$student = new Student();
```

```
foreach( $student->getAssignments() as $assignment ) {  
    // assignment here is an instance of a class that  
    extends WP_Post  
    $assignment->print();  
}  
?>
```

And here is how that code would look in a less object-oriented way:

```
$student = wp_get_current_user(); // return WP_User object  
for current user  
foreach( sp_getAssignmentsByUser( $student->ID ) as  
$assignment ) {  
    sp_printAssignment( $assignment->ID );  
}
```

Both blocks of code are functionally equivalent, but the first example is easier to read and work with. Perhaps more important, having all of your student-related functions coded as methods on the Student class will help keep things organized.

Here are the initialization and getAssignments() method for the Student class:

```
<?php  
class Student extends WP_User {  
    // no constructor so WP_User's constructor is used  
  
    // method to get assignments for this Student  
    function getAssignments() {  
        // get assignments via get_posts if we haven't yet  
        if ( ! isset( $this->data->assignments ) )  
            $this->data->assignments = get_posts( array(  
                'post_type' => 'assignment', // assignments  
                'numberposts' => -1, // all posts  
                'author' => $this->ID // user ID for  
                this Student
```

```

        ));

        return $this->data->assignments;
    }

    // magic method to detect $student->assignments
    function __get( $key ) {
        if ( $key == 'assignments' ) {
            return $this->getAssignments();
        } else {
            // fallback to default WP_User magic method
            return parent::__get( $key );
        }
    }
?>

```

Here, we define the `Student` class to extend the `WP_User` class by just adding `extends WP_User` to the class definition.

We don't write our own constructor function, because we want to use the same one as the `WP_User` class. Namely, we want to get a `Student/User` by ID so that we're able to write `$student = new Student($user_ID);`.

The `getAssignments()` method uses the `get_posts()` function to get all posts of type "assignment" that are authored by the user associated with this `Student`. We store the array of assignment posts in the `$data` property, which is defined in the `WP_User` class and stores all of the base user data and metadata. This allows us to use code like `$student->assignments` to get the assignments later.

Normally if `$student->assignments` is a defined property of `$student`, the value of that property will be returned. But if there is no “assignments” property, PHP will send “assignments” as the `$key` parameter to your `__get` method. Here, we check that `$key == "assignments"` and then return the value of the `getAssignments()` method defined later. If `$key` is something other than “assignments” we pass it to the `__get()` method of the parent `WP_User` class, which checks for the value in the `$data` property of the class instance or, failing that, sends the key to the `get_user_meta()` function.

At first blush, all this does is allow you to type `$student->assignments` instead of `$student->getAssignments()`, which technically is true. However, coding things this way allows us to cache the assignments in the `$data` property of the object so we don’t need to query for it again if it’s accessed more than once. It will also make your code more consistent with other WordPress code: if you want the student’s email, it’s `$student->user_email`; if you want student’s first name, it’s `$student->first_name`; if you want the student’s assignments, it’s `$student->assignments`. The person using the code doesn’t have to know that one of them is stored in the `wp_users` table, one is stored in `wp_usermeta`, and one is the result of a post query.

## Adding Registration and Profile Fields

It's very common to need to add further profile fields for users in your app. In the previous section, we discussed how to use the `wp_update_user()` and `update_user_meta()` functions to manage those values. In this section, we go over how to add editable fields for our user meta to the registration and profile pages.

In our SchoolPress app, we need to capture some data about our users. For students, we want to capture their graduation year, major, minor, and advisor's name. For teachers, we want to capture their department and office location. For both types of users, we want to capture their gender, age, and phone number.

There are a few different plugins out there that will help you do this more quickly. For example, if you install the Paid Memberships Pro Register Helper plugin,<sup>5</sup> you can use the code in Example 6-1 to add these fields to the registration and profile pages.

---

*Example 6-1. Registering additional fields for users*

---

```
<?php
function ps_registration_fields() {
    // store fields in an array
    $fields = array();

    // fields for all users
    $fields[] = new PMProRH_Field(
        'gender',
        'select',
        array(
            'options' => array(
                '' => 'Choose One',
                'male' => 'Male',
                'female' => 'Female',
                'other' => 'Other',
                'undisclosed' => 'Prefer not to say'
            )
        )
    );
}
```

```

        ) ,
        'profile' => true,
        'required' => true
    )
);

$fields[] = new PMProRH_Field(
    'age',
    'text',
    array(
        'size' => 10,
        'profile' => true,
        'required' => true
    )
);
$fields[] = new PMProRH_Field(
    'phone',
    'text',
    array(
        'size' => 20,
        'label' => 'Phone Number',
        'profile' => true,
        'required' => true
    )
);
// fields for teachers
$fields[] = new PMProRH_Field(
    'department',
    'text',
    array(
        'size' => 40,
        'profile' => true,
        'required' => true
    )
);
$fields[] = new PMProRH_Field(
    'office',
    'text',
    array(
        'size' => 40,

```

```

        'profile' => true,
        'required' => true
    )
);

// fields for students
$fields[] = new PMPProRH_Field(
    'graduation_year',
    'text',
    array(
        'label' => 'Expected Graduation year',
        'size' => 10,
        'profile' => true,
        'required' => true
    )
);
$fields[] = new PMPProRH_Field(
    'major',
    'text',
    array( 'size' => 40, 'profile' => true, 'required' =>
true )
);
$fields[] = new PMPProRH_Field(
    'minor',
    'text',
    array( 'size' => 40, 'profile' => true )
);

// add fields to the registration page
foreach( $fields as $field ) {
    pmprorh_add_registration_field( 'after_password',
$field );
}
}

add_action( 'init', 'ps_registration_fields' );
?>
```

You can find full instructions on how to use PMPro Register Helper and the syntax for defining fields in the plugin's *README* file.

Rather than cover that here, let's go through adding one field to the register and profile pages by hand using the same hooks and filters PMPro Register Helper does:

1. Add our field to the registration page:

```
<?php

function sp_register_form() {
    // get the age value passed into
    // the form
    if ( ! empty( $_REQUEST['age'] ) )
        $age = intval( $_REQUEST['age'] )
    ;
    else
        $age = '';

    // show input
?>
<p>
    <label for="age">Age<br />
    <input type="text" name="age"
id="age" class="input"
        value="<?php echo esc_attr(
$age ); ?>" />
    </label>
    </p>
<?php
}

add_action( 'register_form',
'sp_register_form' );
?>
```

## NOTE

We check if ( ! empty( \$\_REQUEST['age'] ) ) to avoid a PHP warning when users first visit the registration page and there isn't any form data in \$REQUEST yet. We also use the esc\_attr() function when outputting the age into the value attribute of the input field. Escape functions are covered in detail in [Chapter 8](#).

2. Update our user's age after registering:

```
function sp_register_user( $user_id ) {
    // get the age value passed into
    // the form
    $age = intval( $_REQUEST['age'] );

    // update user meta
    update_user_meta( $user_id, 'age',
    $age );
}

add_action( 'register_user',
'sp_register_user' );
```

3. Add the age field to the user profile page. We need to hook into both show\_user\_profile and edit\_user\_profile to show our custom field both when users are viewing their own profile and when administrators are editing other users' profiles:

```
<?php
function sp_user_profile( $user ) {
    // show input
```

```

$age = $user->age;
?>
<table class="form-table">
<tbody>
<tr>
    <th><label
for="age">Age</label></th>
    <td>
        <input type="text"
name="age" id="age" class="input"
        value=<?php echo esc_attr(
$age ) ; ?>" />
    </td>
</tr>
</tbody>
</table>
<?php
}
//user's own profile
add_action( 'show_user_profile',
'sp_user_profile' );
//admins editing user profiles
add_action( 'edit_user_profile',
'sp_user_profile' );
?>

```

Note how the default WordPress registration page HTML uses `<p>` tags to separate fields, whereas the default profile HTML in the dashboard uses table rows.

4. Update our field when updating a profile:

```

<?php

function sp_profile_update( $user_id ) {
    //make sure the current user can
    edit this user
    if ( ! current_user_can(
        'edit_user', $user_id ) ) {
        return false;
    }

    // check if value has been posted
    if ( isset( $_POST['age'] ) ) {
        // update user meta
        update_user_meta( $user_id,
            'age', intval( $_POST['age'] ) );
    }
}

// user's own profile
add_action( 'personal_options_update',
    'sp_profile_update' );
// admins editing
add_action( 'edit_user_profile_update',
    'sp_profile_update' );
?>

```

Again, we're hooking into two separate hooks: one for when users are viewing their own profile, and another for when administrators are editing other users' profiles.

So that's how you add a field to the registration and profile pages. Just iterate through that for each field you want to add (or piggyback

on a plugin like PMPro Register Helper so it will do this for you), and you're good to go.

## Customizing the Users Table in the Dashboard

With all of this extra metadata for our users, it is sometimes necessary to extend the basic users list table in the WordPress dashboard.

You can create your own admin page, with custom queries, and a report that mimics the style of the dashboard list tables (as we did for the “Members List” in Paid Memberships Pro). Or, you can use hooks and filters provided by WordPress to add columns and filters to the standard user list, which is what we will cover now.

To do this, we use the `manage_users_columns` and `manage_users_custom_column` filters. Let's add our age field to the user's list:

```
// add our column to the table
function sp_manage_users_columns( $columns ) {
    $columns['age'] = 'Age';
    return $columns;
}

add_filter( 'manage_users_columns',
    'sp_manage_users_columns' );

// tell WordPress how to populate the column
function sp_manage_users_custom_column( $value,
$column_name, $user_id ){
    $user = get_userdata( $user_id );
    if ( $column_name == 'age' )
        $value = $user->age;
```

```
    return $value;
}
add_filter( 'manage_users_custom_column',
    'sp_manage_users_custom_column', 10, 3);
```

The `manage_users_columns` filter passes in an array containing all of the default WordPress columns (and any added by other plugins). You can add columns, remove them (using `unset( $columns['column_name'] )`), and reorder them. The keys in the `$columns` array are unique strings to identify them. The values in the `$columns` array are the headings for each column.

The `manage_users_custom_column` filter is applied to each column in the `manage_users_columns` array that isn't a default WordPress column (i.e., any custom column you add). In the `sp_manage_users_custom_column()` callback, you can do any calculations needed to get the values for each custom column. Typically, the function contains a large `if/then/else` block or a `switch` statement checking the value of `$column_name` and returning the correct value for each column.

If you use the preceding code, you will get an `Age` column added to your users page, but by default you won't be able to click it to sort the users list by age as you can with some of the default users list columns. Here's the code for that:

```
<?php
// make the column sortable
function sp_manage_users_sortable_columns( $columns ) {
    $columns['age'] = 'Age';
```

```

        return $columns;
    }

    add_filter( 'manage_users_sortable_columns',
        'sp_manage_users_sortable_columns' );

    // update user_query if sorting by Age
    function sp_pre_user_query( $user_query ) {
        global $wpdb, $current_screen;

        // make sure we are viewing the users list in the
        dashboard
        if ( $current_screen->id != 'users' ) {
            return;
        }

        // order by age
        if ( $user_query->query_vars['orderby'] == 'Age' ) {
            $user_query->query_from .= " INNER JOIN $wpdb-
>usermeta m1
                ON $wpdb->users u1
                AND (u1.ID = m1.user_id)
                AND (m1.meta_key = 'age');
            $user_query->query orderby = " ORDER BY
m1.meta_value
                " . esc_sql( $user_query->query_vars['order'] );
        }
    }

    add_action( 'pre_user_query', 'sp_pre_user_query' );
?>
```

Here we have defined Age as a sortable column with the manage\_users\_sortable\_columns filter. We use the pre\_user\_query filter to detect the &sortby=Age parameter on the users list page and update the \$user\_query object to join on the wp\_usermeta table and order by age. Notice how we use the \$current\_screen global, which is set on the admin page, to make sure we are on the users list page before editing the query.

The `$user_query->query_vars['order']` value is going to be ASC or DESC, but we wrap the variable in the `esc_sql()` function to protect our query against SQL injections. Even better would be to check that the value is exactly ASC or DESC and throw an error for any other value.

## Plugins

Now that you've seen how to customize various aspects of the WordPress user management system, let's go over a few user-related plugins that will make your life building web apps a little easier.

### Theme My Login

Your members don't need to know that your site is built on WordPress; you can use a login form that's seamlessly integrated with your site design rather than the default WordPress login. The [Theme My Login](#) plugin does this perfectly. Traffic to `wp-login.php` is redirected to a login page that looks like the rest of your site instead of the WordPress backend.

Theme My Login also has useful paid modules for theming user profiles, hiding the dashboard from nonadministrators, and controlling where users are redirected on login and logout.

### Hide the Admin Bar from Nonadministrators

The [Hide the Admin Bar](#) plugin does exactly what the name states. Only administrators will see the WordPress admin bar when browsing your site's frontend. The plugin is just a few lines of code and can be

edited to your needs; for example, to let editors and authors view the admin bar.

## Paid Memberships Pro

The Paid Memberships Pro plugin is brought to you by Stranger Studios and allows you to monetize the content on your site by creating a membership community. This is ideal for any business or blogger looking to lock down some or all of the content or collect fees for services provided. This plugin easily integrates with payment gateways such as Stripe, Paypal, and Authorize.net. Settings for recurring or one-time payments are included. Paid Memberships Pro allows for the creation of different membership levels within your site.

## PMPro Register Helper

The PMPro Register Helper plugin was initially programmed to work with Paid Memberships Pro, but can be used for other projects. This plugin simplifies the process of adding extra fields to the registration and profile fields. Instead of a set of three hooks and functions for each field, fields can be added in a couple lines of code, like this:

```
<?php
$text = new PMProRH_Field(
    'company',
    'text',
    array(
        'size' => 40,
        'class' => 'company',
        'profile' => true,
        'required' => true
    )
)
```

```
) ;  
pmprorh_add_registration_field( 'after_billing_fields',  
$text );  
?>
```

The Register Helper plugin also has shortcodes to insert signup forms into your pages and sidebars and modules to act as starting points for your own registration, profile, and members directory pages.

## Members

The [Members plugin](#) extends the control that you have over user roles and capabilities in your site. It enables you to edit as well as create and delete user roles and capabilities. This plugin also allows you to set permissions for different user roles to determine which roles have the ability to add, edit, and/or delete various pieces of content.

You could always write your own code to add roles and capabilities, but Members adds a nice GUI on top of that functionality that is often useful.

## WP User Fields

Just kidding. There is no plugin called “WP User Fields” yet. It’s 2019 and there still isn’t a good, user-friendly plugin for adding user fields to the default WordPress registration, profile, and user lists. The best solutions rely on another membership or form plugin.

Are you the one to build the user fields plugin WordPress needs? Finish reading this book, and you’ll have the tools to do it. Start with

the next chapter on working with WordPress APIs, Objects, and Helper Functions.

---

- 1 Any class method starting with two underscores is considered a magic method in PHP because it is magically kicked off during certain events.
- 2 For clarity, we took out parts of the method that were for reverse compatibility and filtering in certain circumstances. The preceding code contains the spirit of the method.
- 3 This is how the Paid Memberships Pro plugin registers users from the checkout page.
- 4 When we talk about teachers and students as people, we will leave them lowercase. When talking about our Teacher and Student user types and objects, we capitalize these words.
- 5 Paid Memberships Pro Register Helper was built to work with Paid Memberships Pro, but will work without it as well.

# Chapter 7. Working with WordPress APIs, Objects, and Helper Functions

---

In this chapter, we cover several WordPress APIs, objects, and helper functions that aren't otherwise covered in the rest of the book but are still important pieces of a WordPress developer's arsenal.

## Shortcode API

Shortcodes are specially formatted pieces of text that can be used to insert dynamic output into your posts, pages, widgets, and other static content areas.

Shortcodes come in three main flavors.:

- A single shortcode like [myshortcode]
- Shortcodes with attributes like [myshortcode id="1" type="text"]
- Enclosing shortcodes like [myshortcode id="1"]  
    ... some content here ... [/myshortcode]

In [Chapter 3](#), we shared an example of how to use shortcodes to add arbitrary content into a WordPress post or page. In that example, like flavor number one, we simply swapped out the shortcode for our

content. You can also add attributes to the shortcode to affect the callback function processing it or wrap some content in an opening and closing shortcode pair to filter some particular content.

The primary step of creating shortcodes is to define the callback function for your shortcode using the `add_shortcode()` function. Any attributes are added to an array that is passed to the callback as the first `$atts` parameter. Any enclosed content is passed to the callback as the second `$content` parameter.

The following code creates a shortcode called `msg` and makes use of attributes and enclosed content:

```
<?php
/*
shortcode callback for [msg] shortcode
Example: [msg type="error"]This is an error message.[/msg]
Output:
<div class="message message-error">
    <p>This is an error message.</p>
</div>
*/
function sp_msg_shortcode($atts, $content)
{
    //default attributes
    extract( shortcode_atts( array(
        'type' => 'information',
        ), $atts ) );
    $content = do_shortcode($content);      //allow nested
shortcodes
    $r = '<div class="message message-' .
        $type . '"><p>' . $content . '</p></div>';
    return $r;
}
add_shortcode('msg', 'sp_msg_shortcode');
?>
```

Notice that the content you want displayed is returned from the callback function rather than echoed to the output buffer. This is because the shortcode filter is typically run before any content has been pushed to the screen. If there were any `echo` calls inside this function, the output would show up at the top of the page instead of inline where you want it.

## Shortcode Attributes

The other important piece demonstrated in the preceding code is how the default attributes are set. The `shortcode_atts()` function takes three parameters: `$pairs`, `$atts`, and `$shortcode`.

`$pairs` is an array of default attributes, where each key is the attribute name and each value is the attribute value.

`$atts` is a similar array of attributes, usually passed in straight from the `$atts` parameter passed to the shortcode callback function. The `shortcode_atts()` function merges the default and passed attributes into one array.

The `$shortcode` parameter is optional. If set to match the shortcode name, it will trigger a filter `shortcode_atts_{shortcode}` that can be used by other plugins and so on to override the default attributes.

The results of `shortcode_atts()` are then passed to the PHP function `extract()`, which creates a variable in the local scope for every key in the attributes array.

In this way, the variable `$type` in our example is available to the rest of the function and contains either the default value of `message` or whatever value was set in the shortcode itself.

## Nested Shortcodes

Finally, we pass the inner `$content` through the `do_shortcode()` function to enable nested shortcodes. If you had a `[help_link]` shortcode that generated a link to your documentation depending on what section of a site you were on or the type of user logged in, you might want to use that shortcode within the `[msg]` shortcode:

```
[msg type="error"]
    An error has occurred. Use the following link for
    help: [help_link].
[/msg]
```

As long as the callback function for the `[msg]` shortcode passes its results through `do_shortcode()`, the inner `[help_link]` shortcode will be filtered as intended.

## CAUTION

Though nested shortcodes of different types will work, nesting the same shortcode within itself will break. The regular expression (regex) parser that pulls the shortcodes out of content is engineered for speed. The parser needs to scan through the content only once. Handling nested shortcodes of the same type would require multiple passes through the content, which would slow the algorithm down. The solution to this is to either 1) avoid nesting the same shortcode within itself, 2) use differently named shortcodes that link to the same callback function, or 3) write a custom regex parser for your shortcode and parse the shortcodes out yourself.

You can also use the `do_shortcode()` function to apply shortcodes to custom fields, content pulled from custom tables, or other content not already being run through the `the_content` filter. In most cases outside of shortcode callback functions themselves, it will be more appropriate to use `apply_filters("the_content", $content)`, which will apply all filters on the `the_content` hook including the shortcode filter:

```
<?php
global $post;
$sidebar_content = $post->sidebar_content;
?>
<div class="post">
    <?php the_content(); ?>
</div>
<div class="sidebar">
    <?php
        //echo do_shortcode($sidebar_content);
        echo apply_filters('the_content', $sidebar_content);
    ?>
</div>
```

## Removing Shortcodes

As with actions and filters, you can remove registered shortcodes to keep them from being applied to a certain post or on content you are passing directly to `do_shortcode()` or through the `the_content` filter. The `remove_shortcode()` function takes the shortcode name as its only parameter and will unregister the specified shortcode. `remove_all_shortcodes()` will unregister all shortcodes.

### NOTE

When calling `remove_shortcode()`, make sure that the call comes late enough in the execution of WordPress for the shortcode you want removed to have already been added. For example, if a plugin adds the shortcode during the `init` action on priority 10, you will want to put your call to `remove_shortcode()` during the `init` action on priority 11 or higher or through another hook that fires after `init`.

The array of registered shortcodes is stored in a global variable `$shortcode_tags`. It can be useful to make copies of this variable or edit it directly. For example, if you want to exclude certain shortcodes from a specific piece of content, you can make a backup copy of all shortcodes, remove the offending shortcodes, apply shortcodes, and then restore the original list of shortcodes:

```
//make a copy of the original shortcodes
global $shortcode_tags;
$original_shortcode_tags = $shortcode_tags;

//remove the [msg] shortcode
```

```
unset($shortcode_tags['msg']);

//do shortcodes and echo
$content = do_shortcode($content);
echo $content;

//restore the original shortcodes
$shortcode_tags = $original_shortcode_tags;
```

## Other Useful Shortcode-Related Functions

The following functions are useful when working with shortcodes:

*shortcode\_exists( \$tag )*

Checks whether the shortcode \$tag has been registered.

*has\_shortcode( \$content, \$tag )*

Checks whether the shortcode \$tag appears within the \$content variable.

*shortcode\_parse\_atts( \$text )*

Pulls attributes out of a shortcode. This is done for you when parsing a shortcode, but can be called directly if you want to pull attributes out of other text like HTML tags or other templates.

*strip\_shortcodes( \$text )*

Strips all shortcodes out of the \$text variable and replaces them with empty text instead of calling the callback function.

More details about the Shortcode API can be found in the [WordPress Codex](#).

## Widgets API

Widgets allow you to place contained pieces of code and content in various widget areas throughout your WordPress site. The most typical use cases are to add widgets to a sidebar or footer area. You could always hardcode these sections on a website, but using widgets allows your nondevelopers to drag and drop widgets from one area to another or to tweak their settings through the Widgets page in the admin dashboard. WordPress comes with many built-in widgets, including the basic text widget shown in [Figure 7-1](#).

Plenty of plugins also include widgets for showing various content. We don't get into the use and styling of widgets here, since this is covered well in the [WordPress Codex page on widgets](#), but we will explore how to add widgets and widget areas to your plugins and themes.

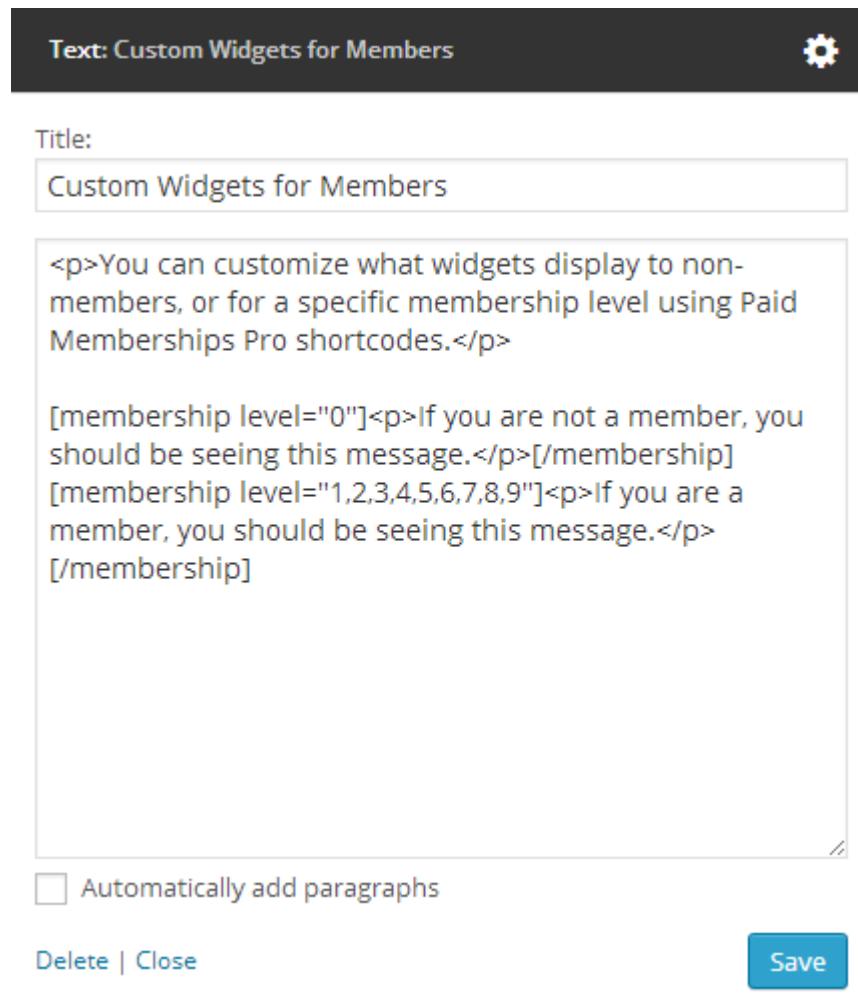


Figure 7-1. Text widget settings

### NOTE

The UI of the Widgets page in the admin dashboard is going through an overhaul for WordPress version 3.8; however, the functions and API calls to add new widgets through code should not be affected much, if at all.

## Before You Add Your Own Widget

Before you go about developing a new widget, it's worth spending some time to see whether an existing widget will work for you. If you

get creative, you can sometimes avoid building a new widget.

Search the repository for plugins that may already have the widget you need. If so, double-check the code there and see whether it will work.

Text widgets can be used to add arbitrary text into a widget space. You can also embed JavaScript code this way or add a shortcode to the text area and use a shortcode to output the functionality you want (you may have created the shortcode already for other use) instead of creating a new widget.

If your widget is displaying a list of links, it might make sense to build a menu of those links and use the Custom Menu widget that is built in to WordPress. Other widgets that display recent posts from a category often will work with CPTs and custom taxonomies either out of the box or with a little bit of effort.

If you do need to add a brand-new widget, the following section covers the steps required.

## Adding Widgets

To add a new widget to WordPress, you must create a new PHP class for the widget that extends the `WP_Widget` class of WordPress. The `WP_Widget` class, found in `wp-includes/widgets.php`, is a good read. The comments in the code explain how the class works and which methods you must override to build your own widget class. There are four main methods that you must override, shown clearly in the

following code by the sample widget class from the [WordPress Codex page for the Widgets API](#):

```
/*
 Taken from the Widgets API Codex Page at:
 http://codex.wordpress.org/Widgets_API
*/
class My_Widget extends WP_Widget {

    public function __construct() {
        // widget actual processes
    }

    public function widget( $args, $instance ) {
        // outputs the content of the widget
    }

    public function form( $instance ) {
        // outputs the options form on admin
    }

    public function update( $new_instance, $old_instance
) {
        // processes widget options to be saved
    }
}
add_action( 'widgets_init', function(){
    register_widget( 'My_Widget' );
}) ;
```

The `add_action()` call passes an anonymous function as the second parameter, which is supported only in PHP versions 5.3 and higher. Technically, WordPress requires only PHP version 5.2.4 and higher. The alternative is to use the `create_function()` function of PHP, which is slower and potentially less secure than using an anonymous function. However, if you plan to release your

code to a wide audience, you might want to use the alternative method shown in the following code:

```
/*
 Taken from the Widgets API Codex Page at:
 http://codex.wordpress.org/Widgets_API
*/
add_action('widgets_init',
    create_function('', 'return
register_widget("My_Widget");')
);
```

Pulling this all together, [Example 7-1](#) presents a new widget for the SchoolPress site. This widget will show either a globally defined note set in the widget settings or a note specific to the current BuddyPress group set by the group administrators.

### Example 7-1. SchoolPress note widget

---

```
<?php
/*
Widget to show the current class note.
Teachers (Group Admins) can change note for each group.
Shows the global note set in the widget settings if non-
empty.
*/
class SchoolPress_Note_Widget extends WP_Widget
{
    public function __construct() {
        parent::__construct(
            'schoolpress_note',
            'SchoolPress Note',
            array( 'description' => 'Note to Show on Group Pages'
        );
    }

    public function widget( $args, $instance ) {
        global $current_user;

        //saving a note edit?
```

```

    if ( !empty( $_POST['schoolpress_note_text'] ) 
        && !empty( $_POST['class_id'] ) ) {
        //make sure this is an admin
        if(groups_is_user_admin($current_user->ID,intval($_POST['class_id']))) {
            //should escape the text and possibly use a nonce
            update_option(
                'schoolpress_note_' . intval(
                    $_POST['class_id']),
                $_POST['schoolpress_note_text']
            );
        }
    }

    //look for a global note
    $note = $instance['note'];

    //get class id for this group
    $class_id = bp_get_current_group_id();

    //look for a class note
    if ( empty( $note ) && !empty( $class_id ) ) {
        $note = get_option( "schoolpress_note_" .
            $class_id );
    }

    //display note
    if ( !empty( $note ) ) {
        ?>
        <div id="schoolpress_note">
            <?php echo wpaution( $note );?>
        </div>
        <?php

        //show edit for group admins
        if ( groups_is_user_admin( $current_user->ID,
            $class_id ) ) {
            ?>
            <a id="schoolpress_note_edit_trigger">Edit</a>
            <div id="schoolpress_note_edit"
                style="display: none;">
                <form action="" method="post">

```

```

<input type="hidden"
       name="class_id"
       value="php echo intval($class_id); ?&gt;
"&gt; /&gt;
        &lt;textarea name="schoolpress_note_text"
cols="30" rows="5"&gt;
        &lt;?php echo
esc_textarea(get_option('schoolpress_note_'.$class_id))
;?&gt;
        &lt;/textarea&gt;
        &lt;input type="submit" value="Save" /&gt;
        &lt;a id="schoolpress_note_edit_cancel"
href="javascript:void(0);"&gt;
            Cancel
        &lt;/a&gt;
        &lt;/form&gt;
        &lt;/div&gt;
        &lt;script&gt;
jQuery(document).ready(function() {

jQuery('#schoolpress_note_edit_trigger').click(function() {
    jQuery('#schoolpress_note').hide();
    jQuery('#schoolpress_note_edit').show();
});

jQuery('#schoolpress_note_edit_cancel').click(function() {
    jQuery('#schoolpress_note').show();
    jQuery('#schoolpress_note_edit').hide();
});
    });
    &lt;/script&gt;
    &lt;?php
}
}
}

public function form( $instance ) {
    if ( isset( $instance['note'] ) )
        $note = $instance['note'];
    else
        $note = "";
</pre

```

```

?>
<p>
    <label for="<?php echo $this->get_field_id(
'note' ); ?>">
        <?php _e( 'Note:' ); ?>
    </label>
    <textarea id="<?php echo $this->get_field_id(
'note' ); ?>">
        name="<?php echo $this-
>get_field_name( 'note' ); ?>">
        <?php echo esc_textarea( $note );?>
    </textarea>
</p>
<?php
}

public function update( $new_instance, $old_instance ) {
    $instance = array();
    $instance['note'] = $new_instance['note'];

    return $instance;
}
}

add_action( 'widgets_init', function() {
    register_widget( 'SchoolPress_Note_Widget' );
} );
?>

```

## Defining a Widget Area

To add widget areas or sidebars to your theme, you need to do two things. First, register the widget area with WordPress. Then, add code to your theme at the point where you want your widget area to appear.

Registering a widget area is fairly straightforward using the `register_sidebar()` function, which takes an array of arguments as its only parameter. The available arguments are as

follows, taken from the [WordPress Codex page](#) on the `register_sidebar()` function:

*name*

Sidebar name (defaults to `Sidebar#`, where # is the ID of the sidebar)

*id*

Sidebar ID—must be all in lowercase, with no spaces (default is a numeric auto-incremented ID)

*description*

Text description of what/where the sidebar is; shown on widget management screen since 2.9 (default: empty)

*class*

CSS class name to assign to the widget HTML (default: empty)

*before\_widget*

HTML to place before every widget (default: `<li id="%1$s" class="widget %2$s">`); uses `sprintf` for variable substitution

*after\_widget*

HTML to place after every widget (default: `</li>\n`)

*before\_title*

HTML to place before every title (default: `<h2 class="widgettitle">`)

*after\_title*

HTML to place after every title (default: `</h2>\n`)

To register a bare-bones sidebar for the assignment pages of our SchoolPress theme, we would add the following to our theme's *functions.php* or *includes/sidebar.php* file:

```
register_sidebar(array(  
    'name' => 'Assignment Pages Sidebar',  
    'id' => 'schoolpress_assignment_pages',  
    'description' => 'Sidebar used on assignment pages.',  
    'before_widget' => '',  
    'after_widget' => '',  
    'before_title' => '',  
    'after_title' => ''  
) );
```

The values for `before/after_widget` and `before/after_title` would be set based on how our theme styles widgets and titles. Some expect `<li>` elements; others use `<div>` elements. But if all of the styling is handled by our widget's code, we can just set everything to empty strings. Next, we need to actually embed the widget area into our theme. We do this using the `dynamic_sidebar()` function, which takes the ID of a registered sidebar as its only parameter:

```
if (!dynamic_sidebar('schoolpress_student_status'))  
{  
    //fallback code in case my_widget_area sidebar was not  
    found  
}
```

The code will load the `schoolpress_student_status` sidebar if found. If it is not found, `dynamic_sidebar()` will return `false` and the code inside the curly braces there will be executed instead. This can be used to show default content in a sidebar area if

the sidebar area doesn't have any widgets within it or doesn't exist at all.

Historically, WordPress themes were developed with a sidebar area, and themes would hardcode certain features into them. Widgets were first introduced primarily to replace these static sidebars with dynamic sidebars that could be controlled through the Widgets page of the dashboard. This is why the term *sidebar* is used to define widget areas, even though widgets are used in places other than just sidebars.

If you need to know whether a sidebar is registered and in use (has widgets) without actually embedding the widgets, you can use the `is_active_sidebar()` function. Just pass in the ID of the sidebar, and the function will return `true` if the sidebar is registered or `false` if it is not. The Twenty Thirteen theme uses this function to check that a sidebar has widgets before rendering the wrapping HTML for the sidebar:

```
<?php
//from twenty-thirteen/sidebar.php
if ( is_active_sidebar( 'sidebar-2' ) ) : ?>
    <div id="tertiary" class="sidebar-container"
role="complementary">
        <div class="sidebar-inner">
            <div class="widget-area">
                <?php dynamic_sidebar(
'sidebar-2' ); ?>
                    </div><!-- .widget-area -->
                </div><!-- .sidebar-inner -->
            </div><!-- #tertiary -->
<?php endif; ?>
```

## Embedding a Widget Outside of a Dynamic Sidebar

The normal process to add widgets to your pages is described in the previous section, where you define a dynamic sidebar and then add your widget to the sidebar through the Widgets page in the admin dashboard.

Alternatively, if you know exactly which widget you want to include somewhere and don't want the placement of the widget left up to the administrators controlling the Widgets settings in the dashboard, you can embed a widget using the `the_widget( $widget, $instance, $args )` function:

`$widget`

The PHP class name for your widget

`$instance`

An array containing the settings for your widget

`$args`

An array containing the arguments normally passed to `register_sidebar()`

Besides hardcoding the placement of the widget, using the `the_widget()` function also allows you to set the settings of the widget programmatically. In the following code, we embed the StudentPress Note widget directly into a theme page. We set the instance array to include an empty string for the `$note` value, ensuring that the group note is shown if available:

```
//show note widget, overriding global note
the_widget('SchoolPress_Note_Widget', //classname
           array('note'=>''), //instance
vars
           array( //widget
vars
           'before_widget' => '',
           'after_widget' => '',
           'before_title' => '',
           'after_title' => ''
)
);
```

## Dashboard Widgets API

Dashboard widgets are the boxes that appear on the home page of your WordPress admin dashboard (see [Figure 7-2](#)).

By default, WordPress includes a few different dashboard widgets. By adding and removing widgets from the dashboard using the Dashboard Widgets API, you can make your WordPress app more useful by placing the information and tools most required by your app directly on the dashboard home page. It's a nice touch that should be done for all WordPress apps with users who will be accessing the WordPress administrator.

# Dashboard

## At a Glance

 1 Post

 3 Pages

 1 Comment

WordPress 5.3 running [Twenty Nineteen](#) theme.

[Search Engines Discouraged](#)

## Activity

### Recently Published

Oct 11th, 1:43 pm [Hello world!](#)

### Recent Comments



From [A WordPress Commenter](#) on [Hello world!](#)

Hi, this is a comment. To get started with moderating, editing, and deleting comments, please visit the Comments screen in...

[All](#) (1) | [Mine](#) (0) | [Pending](#) (0) | [Approved](#) (1) | [Spam](#) (0) | [Trash](#) (0)

## Quick Draft

Title

Content

What's on your mind?

[Save Draft](#)

## WordPress Events and News

Enter your closest city to find nea

City: [Cincinnati](#)

[The Month in WordPress: November 2018](#)

[WPTavern: Gutenberg: One Year Later](#)

[WPTavern: Initial Documentation](#)

[WPTavern: Black Friday Banner Giveaway](#)

[Meetups](#)  | [WordCamps](#)  | 

*Figure 7-2. Dashboard widgets*

## Removing Dashboard Widgets

The dashboard widgets are really just meta boxes assigned to the dashboard page of the administrator. The [WordPress Codex Dashboard Widgets API](#) page lists the default widgets shown on the WordPress dashboard:

```
// From the Dashboard Widgets API Codex Page
// Main column:
$wp_meta_boxes['dashboard']['normal']['high']
['dashboard_browser_nag']
$wp_meta_boxes['dashboard']['normal']['core']
['dashboard_right_now']
$wp_meta_boxes['dashboard']['normal']['core']
['dashboard_recent_comments']
$wp_meta_boxes['dashboard']['normal']['core']
['dashboard_incoming_links']
$wp_meta_boxes['dashboard']['normal']['core']
['dashboard_plugins']

// Side Column:
$wp_meta_boxes['dashboard']['side']['core']
['dashboard_quick_press']
$wp_meta_boxes['dashboard']['side']['core']
['dashboard_recent_drafts']
$wp_meta_boxes['dashboard']['side']['core']
['dashboard_primary']
$wp_meta_boxes['dashboard']['side']['core']
['dashboard_secondary']
```

To remove widgets from the dashboard, you can use the `remove_meta_box( $id, $page, $context )` function:

`$id`

The ID defined when the meta box was added. This is set as the `id` attribute of the `<div>` element created for the meta box.

`$page`

The name of the administrator page the meta box was added to.  
Use `dashboard` to remove dashboard meta boxes.

`$context`

Either `normal`, `advanced`, or `side`, depending on where the meta box was added.

To remove all of the default widgets, you can hook into `wp_dashboard_setup` and make a call to `remove_meta_box()` for each widget you'd like to remove:

```
// Remove all default WordPress dashboard widgets
function sp_remove_dashboard_widgets()
{
    remove_meta_box('dashboard_browser_nag',
'dashboard', 'normal');
    remove_meta_box('dashboard_right_now', 'dashboard',
'normal');
    remove_meta_box('dashboard_recent_comments',
'dashboard', 'normal');
    remove_meta_box('dashboard_incoming_links',
'dashboard', 'normal');
    remove_meta_box('dashboard_plugins', 'dashboard',
'normal');

    remove_meta_box('dashboard_quick_press',
'dashboard', 'side');
    remove_meta_box('dashboard_recent_drafts',
'dashboard', 'side');
    remove_meta_box('dashboard_primary', 'dashboard',
'side');
    remove_meta_box('dashboard_secondary', 'dashboard',
'side');
```

```
'side');
}
add_action('wp_dashboard_setup',
'sp_remove_dashboard_widgets');
```

A different set of widgets is added to the multisite network dashboard, and you must use a different hook to remove the network dashboard widgets. The following code hooks on wp\_network\_dashboard\_setup and removes the meta boxes added to the dashboard-network \$page:

```
//Remove network dashboard widgets
function sp_remove_network_dashboard_widgets()
{
    remove_meta_box('network_dashboard_right_now', 'dashboard-
network', 'normal');
    remove_meta_box('dashboard_plugins', 'dashboard-network',
'normal');
    remove_meta_box('dashboard_primary', 'dashboard-network',
'side');
    remove_meta_box('dashboard_secondary', 'dashboard-
network', 'side');
}
add_action('wp_network_dashboard_setup',
'sp_remove_network_dashboard_widgets');
```

You could use similar code to remove default meta boxes from other dashboard pages, like the edit page and edit post pages. The \$page value to use when removing meta boxes there are page and post, respectively.

## Adding Your Own Dashboard Widget

The `wp_add_dashboard_widget()` function is a wrapper to `add_meta_box()` that adds a widget to your admin dashboard page. The `wp_add_dashboard_widget()` function takes four parameters:

`$widget_id`

An ID for your widgets that is added as a CSS class name to the wrapper for the widget and also used as the array key for the dashboard widgets array.

`$widget_name`

Name of the widget displayed in the widget heading.

`$callback`

Callback function that renders the widget.

`$control_callback`

Optional. Defaults to `null`. Callback function to handle the display and processing of a configuration page for the widget.

Example 7-2 adds a dashboard widget to show the status of current assignments (Figure 7-3). The code includes the call to `wp_add_dashboard_widget()` to register the dashboard widget, as well as the callback function to display that actual widget and another callback function to handle the widget's configuration view (Figure 7-4).

---

*Example 7-2. Assignments dashboard widget*

```
<?php  
/*  
     Add dashboard widgets  
*/
```

```

function sp_add_dashboard_widgets() {
    wp_add_dashboard_widget(
        'schoolpress_assignments',
        'Assignments',
        'sp_assignments_dashboard_widget',
        'sp_assignments_dashboard_widget_configuration'
    );
}

add_action( 'wp_dashboard_setup', 'sp_add_dashboard_widgets'
);

/*
    Assignments dashboard widget
*/
//widget
function sp_assignments_dashboard_widget() {
    $options = get_option(
"assignments_dashboard_widget_options", array() );

    if ( !empty( $options['course_id'] ) ) {
        $group = groups_get_group( array(
            'group_id'=>$options['course_id']
        ) );
    }

    if ( !empty( $group ) ) {
        echo "Showing assignments for class " .
            $group->name . ".<br />...";
        /*
            get assignments for this group and
list their status
        */
    }
    else {
        echo "Showing all assignments.<br />...";
        /*
            get all assignments and list their
status
        */
    }
}

```

```

//configuration
function sp_assignments_dashboard_widget_configuration() {
    //get old settings or default to empty array
    $options = get_option(
        "assignments_dashboard_widget_options", array() );
}

//saving options?
if ( isset(
$_POST['assignments_dashboard_options_save'] ) ) {
    //get course_id
    $options['course_id'] = intval(
        $_POST['assignments_dashboard_course_id']
    );

    //save it
    update_option(
        "assignments_dashboard_widget_options", $options );
}

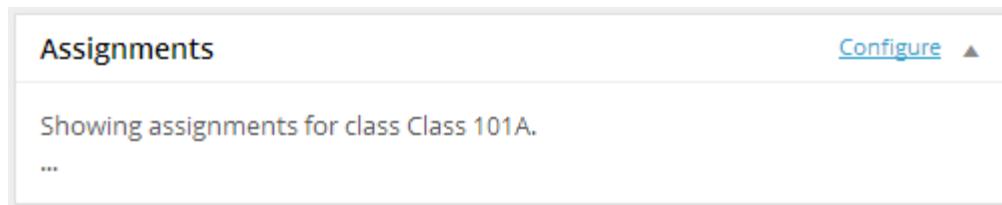
//show options form
$groups = groups_get_groups( array(
    'orderby'=>'name',
    'order'=>'ASC' ) );
?>
<p>Choose a class/group to show assignments from.</p>
<div class="feature_post_class_wrap">
    <label>Class</label>
    <select
        name="assignments_dashboard_course_id">
        <option value="" >?php selected(
            $options['course_id'], "" );?>
            All Classes
        </option>
        <?php
            $groups = groups_get_groups( array(
                'orderby'=>'name',
                'order'=>'ASC'
            ) );
            if ( !empty( $groups ) && !empty(
                $groups['groups'] ) ) {
                foreach ( $groups['groups'] as $group
            ) {

```

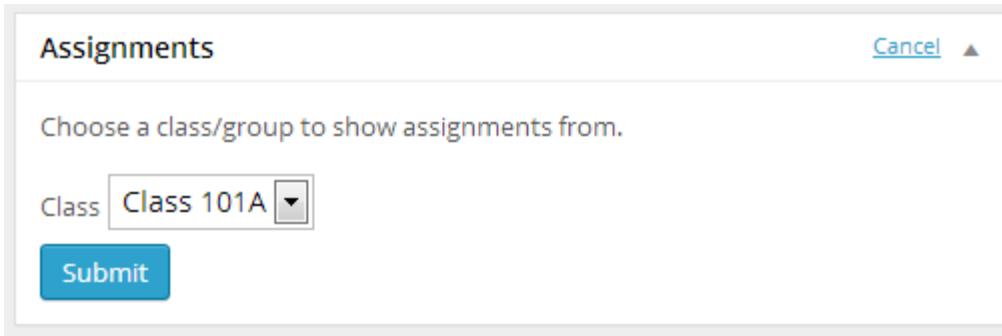
```

        ?>
        <option value="<?php echo intval(
$group->id ) ;?>"
            <?php selected( $options['course_id'],
$group->id ) ;?>>
                <?php echo $group->name; ?>
                </option>
                <?php
            }
        }
        ?>
        </select>
    </div>
    <input type="hidden"
name="assignments_dashboard_options_save" value="1" />
    <?php
}
?>

```



*Figure 7-3. Our assignments widget*



*Figure 7-4. The configuration view of our assignments widget*

Note that we hook into `wp_dashboard_setup` for the function that adds our widget. If we wanted our widget to show up on the network dashboard, we would need to use the `wp_network_dashboard_setup` hook.

The `sp_assignments_dashboard_widget()` function draws the actual widget shown on the dashboard page. This is where we would add our code to loop through assignments and show stats on what percentage of assignments have been turned in.

The

`sp_assignments_dashboard_widget_configuration()` function draws the configuration form and also includes code to process the form submission and update the option we use to store the configuration.

## Settings API

WordPress offers an API you can use to generate options and settings forms for your plugins in the administrator dashboard. This Settings API is very thoroughly documented in the [WordPress Codex](#). Tom McFarlin also has a great [WordPress Settings API tutorial at Envato Tutsplus](#). These resources cover the details of adding menu pages and settings within them for use in your plugins and themes. Following are some tips specific to app developers.

## Do You Really Need a Settings Page?

Before spending the time to create a settings page and adding to the technical debt of your app, consider using a `global` variable to store an array of the options used by your plugin or app:

```
global $schoolpress_settings;  
$schoolpress_settings = array(  
    'info_email' => 'info@schoolpress.me',
```

```
'info_email_name' => 'SchoolPress'  
);
```

For apps that won't be managed by nondevelopers and/or won't be distributed, using a global variable to store your settings might be enough. Just store a global variable like the one in the preceding code at the top of your plugin file or within a *includes/settings.php* file.

Why build the UI if you aren't going to use it?

Even if your plugin or theme will eventually be distributed, we like to start with a global variable like this anyway. The settings that you think you need in the beginning may not be the ones you need at the end of your project. Settings may be added or removed throughout development. Settings you think need a drop-down list may need a free text field instead. The Settings API makes it easy to add settings and update them later, but it is still much easier to change one element in a global array than it is to add or modify a handful of function calls and definitions.

If most of the following statements apply to you, consider using a global variable for your settings instead of building a settings UI:

- This plugin will not be distributed outside my team.
- The only people changing these settings are developers.
- These settings will not need to be different across our different environments.
- These settings are likely to change before release.

## Could You Use a Hook or Filter Instead?

Another alternative to adding a setting to your plugin through the Settings API is to use a hook or filter instead. If a setting you are imagining would be used only by a minority of your users, consider adding a hook or filter to facilitate it.

For example, someone using our WP-Doc plugin may request the ability to restrict *.doc* generation to administrators only or a specific subset of WordPress roles. We could add a settings page with a list of roles with checkboxes to enable or disable *.doc* downloads for that role. Maybe it should just be one checkbox to enable downloads for all roles or just administrators. Maybe it should be a free text field to enter a capability name to check for before allowing the download.

A filter might be a better way to do this. We can add a capability check before the *.doc* is served and use a filter to let developers override the default array of capabilities checked. This code should be added to the `wpdoc_template_redirect()` function of the WP-Doc plugin, before the *.doc* page is rendered:

```
//don't require any caps by default, but allow developers to
//add checks
$caps = apply_filters('wpdoc_caps', array());

if (!empty($caps))
{
    //guilty until proven innocent
    $hascap = false;

    //must be logged in to have any caps at all
    if(is_user_logged_in())
    {
        //make sure the current user has one of the caps
        foreach($caps as $cap)
        {
```

```

    if(current_user_can($cap))
    {
        $hascap = true;
        break; //stop checking
    }
}

if(!$hascap)
{
    //don't show them the file
    header('HTTP/1.1 503 Service Unavailable', true, 503);
    echo "HTTP/1.1 503 Service Unavailable";
    exit;
}

```

You could then override the `wpdoc_caps` array by adding actions like these:

```

//require any user account
add_filter('wpdoc_caps', function($caps) { return
array('read'); });

//require admin account
add_filter('wpdoc_caps', function($caps) { return
array('manage_options'); });

//authors only or users with a custom capability (doc)
add_filter('wpdoc_caps', function($caps) { return
array('edit_post', 'doc'); });

```

## NOTE

The preceding example uses anonymous functions, also known as *closures*, so the `add_filter()` call can be written on one line without using a separate callback function. This syntax requires PHP version 5.3 or higher.

To recap, the greater the number of the following statements that are true, the more it makes sense to use a hook or filter instead of a settings UI:

- Only a small number of people will want to change this setting.
- The people changing this setting are likely to be developers.
- The people changing this setting are likely to have custom needs.
- This setting would require a large number of individual settings or more complicated UI.

## **Use Standards When Adding Settings**

If and when you do need to add settings to your plugin or theme, refer to the tutorials listed earlier in this chapter to make sure you are using the Settings API correctly to add your settings.

Using the Settings API takes a bit of upfront work, but does let you add and edit settings more easily later on. And as you're doing things the WordPress way, other developers will understand how your code works and will be able to hook into it. If a developer wants to make an add-on for your plugin, they will be able to hook into your existing menus and settings sections to add new settings for their plugins.

Using the Settings API will also ensure that your settings look similar to the other settings through a user's WordPress dashboard. You don't want developers to have to learn a new UI just to use your plugin.

## Ignore Standards When Adding Settings

Though you typically want to use the Settings API and the WordPress standards when adding settings for your plugin, sometimes it makes sense to ignore those standards.

The main case here is whether you have a large number of settings that deserve a very custom UI. If you have only one or two settings, users won't be spending a lot of time inside the settings screens. They will just want to change those two settings as fast as possible.

However, if your plugin requires dozens of settings, possibly across multiple tabs or screens, possibly related to one another, it makes sense to treat the settings for your app as an app itself. You should devote some attention to make sure that the UI and UX for your settings screen is as optimized as possible.

The WordPress Settings API is pretty flexible in terms of how things are displayed. You can control how each section is rendered and how each individual setting field is rendered. But in the end, it really is focused on one or more tabs with sections with fields on them. For applications with a large number of settings that interact with one another, you might want to use a different organization for your settings.

Don't be scared to ignore the standards here. Add a menu to the dashboard, have the callback function for it include a set of organized *.php* files to generate the settings form and process it, and follow these tips if possible:

- Add your menu sections and items per the standards, even if your settings pages themselves use a custom layout.
- Remember to sanitize your inputs and use nonces when appropriate.
- Use hooks and filters to whenever possible, if you'd like to allow others to extend your settings.
- Use the same HTML elements and CSS classes whenever possible so the general style remains consistent with the rest of WordPress now and through future updates.

Due to the complexity of ecommerce software, it makes sense that ecommerce plugins often have complicated settings screens. Here are two examples of plugins that do custom settings pages well:

- [Paid Memberships Pro](#) (whose code is posted on GitHub)
- [WooCommerce](#) (whose code is posted on GitHub)

## Rewrite API

Apache comes with a handy module called `mod_rewrite` that allows you to route incoming URLs to different URLs or file locations using rules that are typically added to an `.htaccess` file in your site root folder. Other web servers have similar URL rewriting systems; here are the standard rules for WordPress:

```
# BEGIN WordPress
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
```

```
RewriteRule . /index.php [L]
</IfModule>
# END WordPress
```

If you'd like to understand more about Apache's `mod_rewrite` module and how the WordPress rules work, David Walsh does an excellent [line-by-line explanation](#) of the WordPress `.htaccess` file on his blog. Generally, these rules reroute all incoming traffic to *any* nondirectory or nonfile URL to the `index.php` file of your WordPress installation.

WordPress then parses the actual URL to determine which post, page, or other content to show. For example, under most permalink settings, the URL `/about/` will route to the page or post with the slug "about."

For the most part, you can let WordPress do its thing and handle permalink redirects on its own. However, if you need to add your own rules to handle certain URLs in particular ways, you can do this through the Rewrite API.

## Adding Rewrite Rules

The basic function to add a rewrite rule is `add_rewrite_rule( $rule, $rewrite, $position )`:

`$rule`

A regular expression to match against the URL, just like you would use in an Apache rewrite rule.

`$rewrite`

The URL to rewrite to if the rule is matched. Matched groups from the rule regular expressions are contained in an array called \$matches.

*\$position*

Specifies whether to place the rules above the default WordPress rules (top) or below them (bottom).

If you want to pass a subject line to your contact form through the URL, you could use a URL like */contact/special-offer/*, which would load the contact page and prepopulate the subject to `special-offer`. You could add a rewrite rule like this:

```
add_rewrite_rule(
    '/contact/([^/]+)/?',
    'index.php?name=contact&subject=' . $matches[1],
    'top'
);
add_rewrite_rule(
    flush_rewrite_rules();
```

With this added to the rewrite rules, a visit to */contact/special-offer/* would redirect to the */contact/* page and populate the global `$wp_query->query_vars['subject']` with the value `special-offer`, or whatever text was added after */contact/*. Your contact form could use this value to prepopulate the subject value of the email sent.

## Flushing Rewrite Rules

WordPress caches the rewrite rules. So when you add a rule like this, you need to flush the rewrite rules so they take effect. Flushing the rewrite rules can take some time, so it's important that you don't do it

on every page load. To keep the rewrite rules in order, every plugin that affects them should do these three things:

1. Add the rule during plugin activation and immediately flush the rewrite rules using the `flush_rewrite_rules()` function.
2. Add the rule during the `init` hook in case the rules are flushed manually through the Permalinks Settings page of the dashboard or by another plugin.
3. Add a call to `flush_rewrite_rules()` during deactivation so the rule is removed on deactivation.

The following code shows how our contact subject rule should be added according to the three previous steps:

```
//Add rule and flush on activation.
function sp_activation()
{
    add_rewrite_rule(
        '/contact/([^/]+)/?',
        'index.php?name=contact&subject=' . $matches[1],
        'top'
    );
    flush_rewrite_rules();
}

register_activation_hook(__FILE__, 'sp_activation');

/*
    Add rule on init in case another plugin flushes,
    but don't flush cause it's expensive
*/
function sp_init()
{
    add_rewrite_rule(
        '/contact/([^/]+)/?',
        'index.php?name=contact&subject=' . $matches[1],
```

```
        'top'
    );
}

add_action('init', 'sp_init');

//Flush rewrite rules on deactivation to remove our rule.
function sp_deactivation()
{
    flush_rewrite_rules();
}
register_deactivation_hook(__FILE__, 'sp_deactivation');
```

## Other Rewrite Functions

WordPress offers some other functions to insert special kinds of rewrite rules:

[add\\_rewrite\\_tag\(\)](#)

Another way to add custom querystring variables.

[add\\_feed\(\)](#)

Add a new kind of feed to function like the RSS and ATOM feeds.

[add\\_rewrite\\_endpoint](#)

Add querystring variables to the end of a URL.

The Codex pages for each function explains things well. Some functions will make more sense for certain uses versus others.

Example 7-3 shows how to use the `add_rewrite_endpoint()` function to detect when `/doc/` is added to the end of a URL and to force the download of a `.doc` file. This code makes use of the fact that any HTML document with a `.doc` extension will be read by Microsoft Word as a `.doc` file.

The `add_rewrite_endpoint()` function takes two parameters:

`$name*`

Name of the endpoint—for example, '`doc`'.

`$places*`

Specifies which pages to add the endpoint rule to. Uses the `EP_*` constants defined in *wp-includes/rewrite.php*.

### *Example 7-3. The WP DOC plugin*

---

```
<?php
/*
Plugin Name: WP DOC
Plugin URI: http://bwawwp.com/wp-docx/
Description: Add /doc/ to the end of a page or post to
download a .docx version.
Version: .1
Author: Stranger Studios
*/

/*
    Register Rewrite Endpoint
*/
//Add /doc/ endpoint on activation.
function wpdoc_activation()
{
    add_rewrite_endpoint('doc', EP_PERMALINK | EP_PAGES);
    flush_rewrite_rules();
}
register_activation_hook(__FILE__, 'wpdoc_activation');

//and init in case another plugin flushes, but don't flush
cause it's expensive
function wpdoc_init()
{
    add_rewrite_endpoint('doc', EP_PERMALINK | EP_PAGES);
}
add_action('init', 'wpdoc_init');

//flush rewrite rules on deactivation to remove our endpoint
```

```

function wpdoc_deactivation()
{
    flush_rewrite_rules();
}

register_deactivation_hook(__FILE__, 'wpdoc_deactivation');

/*
    Detect /doc/ use and return a .doc file.
*/
function wpdoc_template_redirect()
{
    global $wp_query;
    if(isset($wp_query->query_vars['doc']))
    {
        global $post;

        //double check this is a post
        if(empty($post->ID))
            return;

        //headers for MS Word
        header("Content-type: application/vnd.ms-
word");
        header('Content-Disposition:
attachment;Filename=' .
$post->post_name.'.doc');

        //html
        ?>
<html>
<body>
<h1><?php echo $post->post_title; ?></h1>
<?php
    echo apply_filters('the_content',
$post->post_content);
    ?>
</body>
</html>
<?php

    exit;
}

```

```
}

add_action('template_redirect', 'wpdoc_template_redirect');

?>
```

Note in the preceding example that we follow the three steps we used in the `add_rewrite_rule()` example to define our rule on activation and initialization and flush all rules on activation and deactivation.

We used `EP_PERMALINK | EP_PAGES` when defining our endpoint, which will add the endpoint to single post pages and page pages.<sup>1</sup> The full list of endpoint mask constants is as follows:

```
EP_NONE
EP_PERMALINK
EP_ATTACHMENT
EP_DATE
EP_YEAR
EP_MONTH
EP_DAY
EP_ROOT
EP_COMMENTS
EP_SEARCH
EP_CATEGORIES
EP_TAGS
EP_AUTHORS
EP_PAGES
EP_ALL
```

For more information on the Rewrite API, both the [Rewrite API Codex page](#) and the [WP\\_Rewrite class Codex page](#) are good sources of information. There is much more that can be done with the `WP_Rewrite` class that we didn't get into here.

## WP-Cron

A cron job is a script that is run on a server at set intervals. The WP-Cron functions in WordPress extend that functionality to your WordPress site. Cron jobs, sometimes called *events*, can be set up to run every few minutes, every few hours, every day, or on specific days of the week or month. Some typical uses of cron jobs include queueing up digest emails, syncing data with third-party APIs, and preprocessing CPU-intensive computations used in reports and comparative analysis.

There are three basic parts to adding a cron job to your app:

1. Schedule the cron event. This will fire a specific hook/action at the defined interval.
2. Hook a function to that action.
3. Place the code you actually want to run within the callback function.

This code can be added to a custom plugin file to schedule some cron jobs:<sup>2</sup>

```
//schedule crons on plugin activation
function sp_activation()
{
    //do_action('sp_daily_cron'); will fire daily
    wp_schedule_event(time(), 'daily', 'sp_daily_cron');
}

register_activation_hook(__FILE__, 'sp_activation');

//clear our crons on plugin deactivation
function sp_deactivation()
{
    wp_clear_scheduled_hook('sp_daily_cron');
}

register_deactivation_hook(__FILE__, 'sp_deactivation');
```

```
//function to run daily
function sp_daily_cron()
{
    //do this daily
}
add_action("sp_daily_cron", "sp_daily_cron");
```

The function `wp_schedule_event( $timestamp, $recurrence, $hook, $args )` has the following attributes:

`$timestamp`

Timestamp for first time to run the hook. You can typically set it to `time()`.

`$recurrence`

How often the event should run. You can pass `hourly`, `daily`, or `twicedaily`, or use the `cron_schedules` hook to add other intervals.

`$hook`

The name of the action to fire on each recurrence.

`$args`

Any arguments you'd like to pass along to the hook fired can be added to the end of the `wp_schedule_event()` call.

We like to give our cron events generic names based on the interval. By doing it this way, if we wanted to run another function daily, we could then just add `add_action('sp_daily_cron', 'new_function_name');` to our codebase.

## Adding Custom Intervals

By default, the `wp_schedule_event()` function accepts only intervals of hourly, daily, or twicedaily. To add other intervals, you need to use the `cron_schedules` hook:

```
//add a monthly interval to use in cron jobs
function sp_cron_schedules($schedules)
{
    $schedules['monthly'] = array(
        'interval' => 60*60*24*30, //really 30 days
        'display' => 'Once a Month'
    );
}
add_filter( 'cron_schedules', 'sp_cron_schedules' );
```

Unlike Unix-based cron jobs, WP-Cron doesn't support intervals based on day of the week. To do this, you can use a daily cron job and have the function called check the day of the week:

```
//run on Mondays
function sp_monday_cron()
{
    //get day of the week, 0-6, starting with Sunday
    $weekday = date("w");

    //is it Monday?
    if($weekday == "1")
    {
        //execute this code on Mondays
    }
}
add_action("sp_daily_cron", "sp_monday_cron");
```

You could write similar code to check for a specific day of the month (`date("j")`) or even specific months (`date("m")`).

## Scheduling Single Events

The preceding examples show how to execute code at some interval. You may also have times when you want to fire an event once at some point in the future. For example, you may want to schedule email delivery of new blog posts one hour after they are posted. This will give authors an hour to fix any issues with a blog post before it gets pushed around the world. The `wp_schedule_single_event()` function can be used in these cases in which we want to schedule an event to fire just once.

## Kicking Off Cron Jobs from the Server

In all of the previous examples, we assumed that events scheduled with `wp_schedule_event()` would actually run when they are scheduled. That's almost true.

On Unix systems, the cron service runs every minute (generally) to check whether there is a script to run. In WordPress, that check is done on every page load. So if no one loads your website in a given day, or only pages from a static cache are loaded, your cron jobs may not fire that day. They will fire at the next page load.

This setup is fine for casual WordPress sites, but our apps need reliability. Luckily, it is easy to disable the internal cron timer and set one up on your web server to fire when you need it to.

To disable the WordPress cron timer, simply add the following to your `wp-config.php` file:

```
define('DISABLE_WP_CRON', true);
```

You still add and manage events as we did earlier; this constant enables or disables the *check* for events that are ready to fire. We just need to manually hit the *wp-cron.php* file in our WordPress installation often enough to fire our scripts when needed.

If you have only daily scripts, you can add a cron job like this via the `crontab -e` command:

```
0 0 * * * wget -O - -q -t 1 http://yoursite.com/wp-cron.php?doing_wp_cron=1
```

### TIP

Information on how to use cron can be found at its [Wikipedia entry](#). Consult the [wget manual](#) on how to use wget.

The `0 0 * * *` part of the preceding entry tells cron to execute this script at 0 minutes on the 0th hour (midnight) every day of the week.

The `wget -O - -q -t 1 http://yoursite.com/wp-cron.php?doing_wp_cron=1` part uses the `wget` command to load up the *wp-cron.php* page in your WordPress install. The `-O -` tells `wget` to send output to `devnull`, and the `-q` enables quiet mode. This will keep cron from adding files to your server or emailing you the outputs of each cron run. The `-t 1` tells cron to try once. This will keep `wget` from hitting your server multiple times if the first try fails. If the call to *wp-cron.php* is failing, the rest of your website is probably failing too; hopefully you've already been notified.

Be sure to change `yoursite.com` to your actual site URL. And finally, the `?doing_wp_cron=1` on the end of the URL is needed since `wp-cron.php` will check for that `$_GET` parameter before running.

### CAUTION

Double-check that the URL to `wp-cron.php` is excluded from any caching mechanisms you may have installed on your site.

This one cron job will fire every day, and any daily cron jobs you scheduled inside WordPress will fire daily. If you need your crons to run more often, you can change the cron entry to run every hour or every few minutes. Note that a call to `wp-cron.php` is basically a hit to your website. A check every minute is effectively the same as an additional 1,440 users hitting your site. So schedule your cron jobs conservatively.

## Using Server Cron Only

If you aren't distributing your code or don't mind telling your users that they must set up server-side cron jobs, you don't need to schedule your cron events in WordPress at all. Just schedule a server-side cron job that calls a special URL to kick off your callback function. This is especially useful if you need to have more control over what times of day your crons run or otherwise just feel more comfortable managing your cron jobs in Unix instead of WordPress.

## NOTE

The information on scheduling server-side cron jobs in this section can be used to replace WP-Cron for recurring events. Single events set using `wp_schedule_single_event()` will still need to be handled using WP-Cron or some other mechanism.

If we were running our Monday cron job from earlier, we would update the code in WordPress:

```
//run on Mondays
function sp_monday_cron()
{
    //check that cron param was passed in
    if(empty($_REQUEST['sp_cron_monday']))
        return false;

    //execute this code on Mondays
}
add_action("init", "sp_monday_cron");
```

And your cron job entry would look like this:

```
0 0 * * 1 wget -O - -q -t 1 http://yoursite.com/?sp_cron_monday=1
```

## NOTE

Again, make sure that the URL to `?sp_cron_monday=1` is excluded from any caching mechanisms you may have installed on your site.

# WP Mail

The `wp_mail()` function is a replacement for PHP's built-in `mail()` function. It looks like this:

```
wp_mail( $to, $subject, $message, $headers, $attachments )
```

Its attributes are as follows:

*\$to*

A single email address, comma-separated list of email addresses, or array of email addresses the email will be sent to (using the “To:” field).

*\$subject*

The subject of the email.

*\$message*

The body of the email. By default, the email is sent as a plain-text message and should not include HTML. However, if you change the content type (see the following example), you should include HTML in your message.

*\$headers*

Optional array of mail headers to send with the message. This can be used to add CCs, BCCs, and other advanced mail headers.

*\$attachments*

A single filename or array of filenames to be attached to the outgoing email.

There are two major improvements `wp_mail()` makes over `mail()`:

- The `wp_mail()` function is hookable. The `wp_mail` filter will pass an array of all of the parameters passed into the `wp_mail()` function for you to filter. You can also filter the sending address using the `wp_mail_from` and `wp_mail_from_name` filters.
- The `wp_mail()` function can be passed a single filename or array of filenames in the `$attachments` parameters, which will be attached to the outgoing email. Attaching files to emails is very complicated, but `wp_mail()` makes it easy by wrapping around the `PHPMailer` class, which itself wraps around the default PHP `mail()` function.

## Sending Nicer Emails with WordPress

By default, emails sent through the `wp_mail()` function are sent from the administrator email address set on the General Settings page of the administrator dashboard, with “WordPress” used as the name. This is not ideal. You can change these values using the `wp_mail_from` and `wp_mail_from_name` filters.

Also by default, emails are sent using plain text. You can use the `wp_mail_content_type` filter to send your emails using HTML.

Finally, it is nice to add a styled header and footer to all of your outgoing emails. You can do this by filtering the email message using the `wp_email` filter.

The following code combines these techniques to pretty up the emails being sent by your WordPress app:

```

//Update from email and name
function sp_wp_mail_from($from_email)
{
    return 'info@schoolpress.me';
}
function sp_wp_mail_from_name($from_name)
{
    return 'SchoolPress';
}
add_filter('wp_mail_from', 'sp_wp_mail_from');
add_filter('wp_mail_from_name', 'sp_wp_mail_from_name');

//send HTML emails instead of plain text
function sp_wp_mail_content_type( $content_type )
{
    if( $content_type == 'text/plain')
    {
        $content_type = 'text/html';
    }
    return $content_type;
}
add_filter('wp_mail_content_type',
'sp_wp_mail_content_type');

//add a header and footer from files in the active theme
function sp_wp_mail_header_footer($email)
{
    //get header
    $headerfile = get_stylesheet_directory() .
"email_header.html";
    if(file_exists($headerfile))
        $header = file_get_contents($headerfile);
    else
        $header = "";
    //get footer
    $footerfile = get_stylesheet_directory() .
"email_footer.html";
    if(file_exists($footerfile))
        $footer = file_get_contents($footerfile);
    else

```

```
$footer = "";

//update message
$email['message'] = $header . $email['message'] . $footer;

return $email;
}

add_filter('wp_mail', 'sp_wp_mail_header_footer');
```

Sending emails from your server can present interesting network problems. Running a local SMTP server for sending emails can be time-consuming on top of the work of running a web server. Deliverability of your emails can be affected by spam filters that haven't whitelisted your apps IP range. The [Configure SMTP](#) plugin can be used to send your outgoing email through an external SMTP server like a Google Apps account. Services like Mandrill and SendGrid, each with its own WordPress plugin, also offer ways to send email from their trusted servers with additional tracking of open and bounce rates.

## File Header API

The comment block at the top of the main theme and plugin files are often referred to as *headers*. The File Header API consists of three functions, `get_plugin_data()`, `wp_get_theme()`, and `get_file_data()`, which allow you to parse these comment blocks.

As a reminder, here is what a plugin's file header may look like:

```
/*
Plugin Name: Paid Memberships Pro
```

```
Plugin URI: http://www.paidmembershipspro.com
Description: Plugin to Handle Memberships
Version: 1.7.3.2
Author: Stranger Studios
Author URI: http://www.strangerstudios.com
*/
```

You can pull this data into an array by calling the `get_plugin_data()` function:

```
get_plugin_data( $plugin_file, $markup = true, $translate =
true )
```

Its attributes are:

`$plugin_file`

The absolute path to the main plugin file where the header will be parsed.

`$markup`

A flag that, if set to `true`, will apply HTML markup to some of the header values. For example, the plugin URI will be turned into a link.

`$translate`

A flag that, if set to `true`, will translate the header values using the current locale and text domain.

The following code loops through the plugins directory and will show data for *most* of the plugins there. It actually takes quite a bit of logic to find all plugins in all formats. For that you can use the `get_plugins()` function, which will return an array of all plugins or take a look at the code for that function found in `wp-`

`admin/includes/plugin.php`. More information on `get_plugins()` can be found on the WordPress Codex's Function Reference page:

```
//must include this file
require_once(ABSPATH . "wp-admin/includes/plugin.php");

//remember current directory
$cwd = getcwd();

//switch to themes directory
$plugins_dir =ABSPATH . "wp-content/plugins";
chdir($plugins_dir);

echo "<pre>";

//loop through theme directories and print theme info
foreach(glob("*", GLOB_ONLYDIR) as $dir)
{
    $plugin = get_plugin_data($plugins_dir .
        "/" . $dir . "/" . $dir . ".php", false, false);
    print_r($plugin);
}

echo "</pre>";

//switch back to current directory just in case
chdir($cwd);
```

Similarly, you can use `wp_get_theme()` to get information out of a theme's file header:

```
wp_get_theme( $stylesheet, $theme_root ) +
```

Its attributes are as follows:

`$stylesheet`

The name of the directory for the theme. If not set, this parameter will be the current theme's directory.

`$theme_root`

The absolute path to the theme's root folder. If not set, the value returned by `get_raw_theme_root()` is used.

The following code loops through the themes directory and will show data for *most* of the themes there. It actually takes quite a bit of logic to find all themes. For that you can use the `wp_get_themes()` function, which will return an array of all `WP_Theme` objects, or take a look at the code for that function found in *wp-includes/theme.php*. More information on `wp_get_themes()` can be found on their [WordPress Codex page](#):

```
//remember current directory
$cwd = getcwd();

//switch to themes directory
$themes_dir = dirname(get_template_directory());
chdir($themes_dir);

echo "<pre>";

//loop through theme directories and print theme info
foreach(glob("*", GLOB_ONLYDIR) as $dir)
{
    $theme = wp_get_theme($dir);
    print_r($theme);
}

echo "</pre>";

//switch back to current directory just in case
chdir($cwd);
```

## Adding File Headers to Your Own Files

Both the `get_plugin_info()` and `wp_get_theme()` functions make use of the `get_file_data()` function. You can access the `get_file_data()` function directly to pull file headers out any file. This can help you to create your own drop-ins or sub-plugins (often referred to as *modules* or *add-ons*) for your plugins.

`get_file_data( $file, $default_headers, $context = "" )` has the following attributes:

`$file`

The full path and filename of the file to pull data from.

`$default_headers`

An array of the header fields to look for. The keys of the array should be the header names, and the values of the array should be regex expressions for parsing the label that comes before the “`:`” in the comment. You can usually just enter the header name as the regex as well.

`$context`

A label to differentiate between different kinds of headers. This parameter determines which `extra_{context}_headers` filter is applied to the default headers passed in:

```
//set headers for our files
$default_headers = array(
    "Title" => "Title",
    "Slug" => "Slug",
    "Version" => "Version"
);
```

```

//remember current directory
$cwd = getcwd();

//change to reports directory
$reports_dir = dirname(__FILE__) . "/reports";
chdir($reports_dir);

echo "<pre>";

//loop through .php files in reports directory
foreach (glob("*.php") as $filename)
{
    $data = get_file_data($filename,
$default_headers, "report");
    print_r($data);
}

echo "</pre>";

//change back to the current directory in case someone
expects the default
chdir($cwd);

```

## Adding New Headers to Plugins and Themes

The `extra_plugin_headers` and `extra_theme_headers` filters can be used to add further headers to the information returned by `get_plugin_data()` and `wp_get_theme()`, respectively. You can also use filters for your custom contexts to work with `get_file_data()`. The format is `extra_{context}_headers`.

[Example 7-4](#) adds an `Allow_Updates` header to plugins. If this header is found and the value is `no` or `false`, that plugin will not be flagged to update.

#### *Example 7-4. The Stop Plugin Updates plugin*

---

```
<?php
/*
Plugin Name: Stop Plugin Updates
Plugin URI: http://bwawwp.com/plugins/stop-plugin-updates/
Description: "Allow Updates: No" in a plugin's header keeps it
from updating.
Version: .1
Author: Stranger Studios
Author URI: http://www.strangerstudios.com
*/

//add AllowUpdates header to plugin
function spu_extra_plugin_headers( $headers ) {
    $headers['AllowUpdates'] = "Allow Updates";
    return $headers;
}
add_filter( "extra_plugin_headers", "spu_extra_plugin_headers"
);

/*
    loop through plugins
    check if updates are disallowed and if so remove it
from list
*/
function spu_pre_set_site_transient_update_plugins(
$update_plugins ) {
    //see if there are any plugins needing updates
    if ( !empty( $update_plugins ) && !empty( $update_plugins-
>response ) ) {
        //loop through plugins
        $new_plugins = array();
        foreach ( $update_plugins->response as $pluginpath =>
$plugin ) {
            //check if the plugin is allowed or not
            $plugin_data = ABSPATH . '/wp-content/plugins/' .
$pluginpath;
            $plugin_data = get_plugin_data( $plugin_data );
            if ( strtolower( $plugin_data['Allow Updates'] ) ==
"no" ||
                strtolower( $plugin_data['Allow Updates'] ) ==
>false" ) {
                    //change checked version and don't add to the new
```

```
response
    $update_plugins->checked[$pluginpath] =
$plugin_data['Version'];
}
else {
    //not blocked. add plugin to new response
    $new_plugins[$pluginpath] = $plugin;
}
}
$update_plugins->response = $new_plugins;
}

return $update_plugins;
}
add_action(
    'pre_set_site_transient_update_plugins',
    'spu_pre_set_site_transient_update_plugins'
);
?>
```

## Heartbeat API

The Heartbeat API, introduced in WordPress 3.9, is near-real-time Ajax communication to push and pull specific content to and from the WordPress database. If activated, the Heartbeat API will beat every 15 to 30 seconds by default on a loaded page to check whether something has happened; this is called the *Heartbeat pulse*. You can also set the pulse manually in code for up to 60 seconds.

### NOTE

Check out the *Heartbeat Control* plugin developed by Jeff Matson. It's great for easily adjusting how often Heartbeat pulses are sent to the server and where Heartbeat pulses are being sent from; you can even turn them off.

The Heartbeat API is more of a WordPress backend feature, and although it can be used on the front, it probably shouldn't be (or at least should be used very sparingly). As you can imagine, the Heartbeat API could be pretty labor intensive on server resources, depending what you are using it for and how many people are using it. This API is being used in WordPress core to do such things as post locking; you know that nice little administrative pop up on a post you're currently editing that lets you know that someone else wants to take over? *If this has happened to you, you might have thought to yourself, can't they see the little lock icon next to the post and the warning pop up saying that I'm actually working on it right now?* This is a great example of the Heartbeat API being used to check the status of who's working on a post and to take action accordingly. When the post listing page first loads, it knows who's working on what pages; then the Heartbeat API takes over to check every x seconds if a user is done with a post they were working on or if someone opened another post and started working on that. WordPress knows these actions from the loaded page by sending data asynchronously via the Heartbeat API to the server to be processed and returned to update the loaded page via JavaScript. So that person totally knew that you were working on that post and decided to just take it over anyways...smh!

The Heartbeat API works in three main steps:

1. Sending data to the server from the client via jQuery. This is done with the heartbeat-send event.
2. Processing the request on the server and providing a response via PHP. This is done with the

`heartbeat_received` filter.

3. Processing the response on the client via jQuery. This is done with the `heartbeat-tick` event.

In our example, we'll build a simple plugin that leverages the Heartbeat API to show staff at a grade school when a parent arrives to pick up their kid(s) from the car line. Almost every grade school we've dealt with has this archaic and dangerous method of teachers walking car to car in the heat, cold, rain, and snow to have parents sign a piece of paper to pick up their children. It's 2020, school board members; we can do better!

First let's build a basic plugin that displays an administrator screen on a monitor in the cafeteria where students wait patiently for their parents to arrive. This will be a shorthand version of a WordPress plugin concept to keep the code short and sweet to highlight the Heartbeat API functionality:

1. Make a folder in your plugin directory called *the-teacher-life-saver*.
2. Make a file in this folder called *the-teacher-life-saver.php*.
3. Review and copy the following code.
4. Save and activate the plugin, then test it on the administrator dashboard.

Example 7-5 creates an administrator dashboard widget that will automatically display content processed on the server via the WP Heartbeat API without manually refreshing the page.

---

*Example 7-5. The Teacher Life Saver plugin*

```
<?php
/**
 * Plugin Name: The Teacher Life Saver
 * Plugin URI: https://schoolpress.me/
 * Description: Alert teachers of parent arrivals
 * Version:      0.0.1
 */

// Action hook for creating a dashboard widget
function ttls_dashboard_widget() {
    global $wp_meta_boxes;

    // widget ID, widget name, callback function
    wp_add_dashboard_widget('ttls_widget', 'Student Pick Ups',
        'ttls_dashboard');
}

add_action('wp_dashboard_setup', 'ttls_dashboard_widget');

// Markup for the dashboard widget
function ttls_dashboard() {
    echo "<div id='ttls_message'></div>";
}

// Change the default pulse to every 15 seconds so we don't
// have to wait as long
function ttls_heartbeat_settings( $settings ) {
    $settings['interval'] = 15; // Anything between 15-60
    seconds
    return $settings;
}
add_filter( 'heartbeat_settings', 'ttls_heartbeat_settings', 1
);

// Enqueue heartbeat.js and our JavaScript functions
function ttls_heartbeat_init()
{
    // Only run this on the dashboard page (index.php)
    global $pagenow;
    if( $pagenow != 'index.php' )
        return;

    // Enqueue the Heartbeat API
    wp_enqueue_script('heartbeat');
```

```

// Load our JavaScript functions in the footer
add_action("admin_footer", "ttls_js_wp_footer");
}

add_action("admin_init", "ttls_heartbeat_init");

// JavaScript functions ran in the footer
function ttls_js_wp_footer()
{
?>
<script>
jQuery(document).ready(function() {

// Use heartbeat-send to send any keys/values in the data
array
    jQuery(document).on('heartbeat-send', function(e,
data) {
        data['client'] = 'check-for-parents';
});

// Use heartbeat-tick function check for data and take action
jQuery(document).on('heartbeat-tick', function(e, data) {
if(data['server'])
    document.getElementById("ttls_message").innerHTML =
data['server']
    + document.getElementById("ttls_message").innerHTML;
});
});
</script>
<?php
}
}

// Server-side function to receive and process the request
then return a response
function ttls_heartbeat_received($response, $data)
{
// Look for whatever data was passed from JS heartbeat-send
function
if($data['client'] == 'check-for-parents')
{
// Build whatever response you want
$r = '<p>';
$r .= date( 'm/j/y g:i a', current_time( 'timestamp', 0 ) );
$r .= " - Nina Messenlehner's father Brian Messenlehner ";

```

```

$r .= "has arrived in a 2007 White Hummer H2.";
$r .= '</p>';

return $r;
}

add_filter('heartbeat_received', 'ttls_heartbeat_received',
10, 2);
?>

```

If you did this correctly you should see whatever content you loaded into the `$response[server]` var in the `ttls_heartbeat_received` function pop up in your new dashboard widget around every 15 seconds. You can really do whatever processing you want on the server side and return anything you want in the response to be processed on the client side. In our example, we basically hardcoded a simple response to showcase how it works, but you could take it much further. Imagine a parent having a mobile app on their smartphone in conjunction with a geofence around the school parking lot. As soon as the parent pulls into the parking lot, their status could be updated; and the next time the Heartbeat pulse checks for new parents, it could return that parent's information to the monitor. In our example we returned very basic information, but we could include a lot more like the student name, parent name, parent photo, vehicle type, vehicle photo, license plate number, and anything else that might be important.

<sup>1</sup> Posts with `post_type` page.

<sup>2</sup> If you move this code into a subdirectory of your plugin, you need to update the `register_activation_hook()` and `register_deactivation_hook()` calls to point to the main plugin file.

# Chapter 8. Secure WordPress

---

Hackers beware! This chapter is packed full of tips and advice on how to make WordPress sites more secure and hopefully prevent them from falling prey to any malicious intent.

## Why It's Important

No matter what size website you are running, security is something that you do not want to overlook. Any size site can fall victim to hackers or malware. Being knowledgeable and proactive about WordPress security will help you be less vulnerable and hopefully avoid any attacks.

One of the most popular types of attacks is called a *brute-force attack*, in which a bot or script of code tries to gain access to your site by guessing the correct username and password combination. It may not sound that dangerous, but keep in mind that these bots are huge networks of computers making hundreds or even thousands of guesses every second! Even if these bots don't gain access to your WordPress administrator, they will often take your site down anyway through the sheer amount of resources it takes your server to respond to the malicious requests. This is called a *denial of service* (DoS)

attack, and can be caused by a targeted attack or by automated spammers and brute-force hacks.

In this chapter, we discuss the standard WordPress installation's built-in security features, in addition to other tips that you can easily follow to make your site more secure. We'll also highlight some plugins that can help with other issues, such as spam.

Some very bad things that can happen to you if you decide to *not* read the rest of this chapter. Here are some pretty frequent scenarios:

- You attempt to pull up your website but find that it's not there anymore. Downtime is bad! Hopefully you have a backup and can restore it quickly.
- You notice that you start showing up in search results for Viagra and other male enhancement drugs. This can be bad for business if your website is not specifically selling these drugs.
- Your application is sending out emails to all of your members with links to download a computer virus. Nobody wants that.
- Your application is hacked and the personal information of your members (their names, addresses, phone numbers, and email addresses) is exposed.
- Your website is hacked and is used to infect other websites with malware. This is the quickest way to get delisted from Google search results and other important directories.

## Security Basics

These are the simplest but most important security tips to consider. Pay attention here because it could save you a lot of time, money, and upset visitors/members.

## **Update Frequently**

The first and most important security tip is to always make sure you upgrade to the most recent version of WordPress as soon as a new version becomes available and also always update any plugins/themes that you have installed on your site. Many of the updates that are pushed out involve security updates; therefore, it is always important to upgrade your software in order to stay up to date and safe.

## **Don't Use the Username “admin”**

Another important item to take care of is making sure not to use “admin” as one of your user accounts. Many bots will automatically try to log in to your site with this username. Knowing that most people don’t change this account is half the battle; all they really need to focus on is guessing the password. When installing WordPress, the default username will be “admin” unless you specifically change it, and you *should* specifically change it! If you are already using WordPress under the username “admin,” you should create a new user account with an administrator role, log in with that new user, and delete the default administrator account. *Make sure you change over any posts or pages created by your administrator account to this new account.*

## **Use a Strong Password**

Choosing a secure password is also very important, especially for your administrator accounts. Don't use one word or one name.

Jumble your password up and make it not connected to you personally.

Make sure your password is a combination of upper- and lowercase letters as well as numbers and special characters. A good password should also be at least 10 characters long; the more characters you use, the stronger your password will be. If you are having trouble coming up with one yourself, just mash on your keyboard a bit or use a service like [Random.org](https://Random.org). Make sure you memorize it or copy it somewhere and secure it properly. WordPress will tell you whether you are using a strong password; please take this into consideration.

## Examples of Bad Passwords

Following are some examples of bad passwords:

- password
- password123
- pa55w0rd
- 123456789
- qwerty
- batman
- mustang
- letmein

Using any variation of password or single words, numbers, or names is a bad idea:

- usmarine (Brian was in the Marines)
- brianmessenlehner (Brian's first and last name)
- jason&kim050507 (Jason's name, his wife's name, and their anniversary)
- Dalya-Brian (kids' names)
- ThaiShortiMaxx (pets' names)
- IAMAWESOME! (everybody knows this, so it could be easy to guess)

Anybody who knows anything about us or our families could potentially guess passwords like these.

## Examples of Good Passwords

Following are some examples of strong passwords:

- U\$s(#8H27@!
- !lik32EaTF1\$h&CHIp5
- #Uk@nN0tBr3akTh1s\$h1t!!!
- [0mG-LoL-R0Fl-T0T3\$CraY]!

It can be a pain in the neck and take an extra second or two entering a good password, but it's well worth it if it can prevent your website/application from being hacked.

# Hardening WordPress

The work of making a website more secure is often referred to as *hardening*. The WordPress Support section “[Hardening WordPress](#)” has similar information to what’s in this chapter, plus other things we don’t cover. Read this chapter first, then see “Hardening WordPress” for more information.

Let’s go over a few techniques for making it harder for your application to be hacked.

## Don’t Allow Admins to Edit Plugins or Themes

By default, WordPress allows administrators to edit the source code of any plugin or theme directly in the web browser. You should disable this functionality so that if a hacker is able to log in to one of your administrator accounts, they can’t add malicious code via the administrator user interface for editing code. To disable this functionality, add this code to your *wp-config.php* file:

```
define( 'DISALLOW_FILE_EDIT', true );
```

## Change Default Database Tables Prefix

The standard WordPress installation uses *wp\_* as a prefix for all tables in the database. By simply changing this prefix to something else, you will make your site far less vulnerable to hackers who attempt SQL injections and assume you’re using the generic *wp\_* prefix. On a brand-new WordPress installation, you have the option to

use any table prefix you want; you should change the default `wp_` prefix to something custom.

To do this on a WordPress site that is already up and running, follow these steps:

1. Make a database backup just in case you mess this up!
2. Open `wp-config.php` and change `$table_prefix = wp_;` to `$table_prefix = anyprefix_;`.
3. Update the existing table names in your database to include your new prefix with the following SQL commands using phpMyAdmin or any SQL client such as MySQL Workbench:

```
rename table wp_commentmeta to
anyprefix_commentmeta;
rename table wp_comments to
anyprefix_comments;
rename table wp_links to
anyprefix_links;
rename table wp_options to
anyprefix_options;
rename table wp_postmeta to
anyprefix_postmeta;
rename table wp_posts to
anyprefix_posts;
rename table wp_terms to
anyprefix_terms;
rename table wp_term_relationships to
anyprefix_term_relationships;
```

```
rename table wp_term_taxonomy to
anyprefix_term_taxonomy;
rename table wp_usermeta to
anyprefix_usermeta;
rename table wp_users to
anyprefix_users;
```

### NOTE

You will need to run a similar `rename` SQL query for any custom tables added by your app or plugins you are using.

Using SQL commands or a SQL client, update any of the instances of `wp_` in the `prefix_options` and `anyprefix_usermeta` tables and change any values like `wp_` to `prefix_`:

```
update anyprefix_options set option_name = replace(
option_name, 'wp_', 'anyprefix_');
update anyprefix_usermeta set meta_key = replace(
meta_key, 'wp_', 'anyprefix_');
```

Test out your site and make sure everything is working as it should.

If you don't feel comfortable making these changes manually, there are plugins available that can change your table prefix for you:

- [Change Table Prefix](#)
- [Brozzme DB Prefix & Tools Addons](#)

## Move wp-config.php

The WordPress *wp-config.php* file stores valuable information like your database location, username, and password and your WordPress authentication keys. Since these values are stored in PHP variables and they are not displayed to the browser, it is not likely that anybody could gain access to this data, but it could happen. You can move *wp-config.php* to one level above your WordPress install, which in most cases should be a nonpublic directory. If it doesn't find it in your root directory, WordPress automatically looks one level up for *wp-config.php*. For example, move */username/public\_html/wp-config.php* to */username/wp-config.php*.

You can also store *wp-config.php* as any filename in any directory location. To do this, make a copy of *wp-config.php*, name the copy whatever you want, and move it to any directory above your root install of WordPress. In your original *wp-config.php* file, remove all of the code and add an `include` to the relative path and filename of the copy you made. For example, copy */username/public\_html/wp-config.php* to */username/someotherfolder/stuff.php*. Change the code in the *wp-config.php* file to

```
include ('/username/someotherfolder/stuff.php')  
;
```

## Hide Login Error Messages

Normally, when you're trying to log in to your site, WordPress displays an error message if you've entered the wrong username or password. Unfortunately this lets hackers know exactly what they are doing wrong or right when attempting to access your site.

Luckily there is a simple fix for this: add a line of code into your theme *functions.php* file or in a custom plugin that will hide or alter those messages:

```
add_filter( 'login_errors', function ( $message ) {
    return "Invalid username or password.";
} );
```

### NOTE

The preceding code uses an [anonymous function](#) as the callback in the second parameter of the `add_filter()` call. This requires PHP version 5.3 or higher. You could also just define a named function above the `add_filter()` call, but then this wouldn't be “one line” of code.

## Hide Your WordPress Version

Many bots scour the internet in search of WordPress sites to target specifically by the WordPress version they are running. These bots look for sites with known vulnerabilities they can exploit. By default, WordPress displays the following code within the `<head></head>` of every page:

```
`<meta name="generator" content="WordPress 3.8.1" />`
```

You can easily hide the version of WordPress you are using by implementing the following code:

```
add_filter( 'the_generator', '__return_null' );
```

## NOTE

There are various ways to detect whether a site is using WordPress and what version. For example, if no script version is specified, the WordPress version is appended to the URLs of JavaScript files. There are other, even more subtle ways in which a determined hacker could detect the version of WordPress you are using. Still, every little bit helps to thwart largely automated attacks that are constantly floating around the internet.

## Don't Allow Logins via wp-login.php

Some bots are smarter than others. We just discussed hiding your WordPress version from some bots, but sometimes all a bot needs to know is that you're using WordPress. This is easy if it sends a POST request to *wp-login.php*. Once a bot knows *wp-login.php* exists, it can start trying to log in to your site.

We like to redirect *wp-login.php* to the home page, which prevents bots from specifically trying to log in using this file. Follow these steps to make an alternative login page and hide the default *wp-login.php* login page:

1. Add the following rewrite rule to your *.htaccess* file:

```
RewriteRule ^new-login$ wp-login.php
```

Note that */new-login/* will be the URL you can use to actually log in to *wp-admin*. You can change this to whatever you want.

2. In your theme *functions.php* file or in a custom plugin, add this code:

```

function schoolpress_wp_login_filter(
    $url, $path, $orig_scheme ) {
    $old = array( "/(wp-login\.php) /"
);
    $new = array( "new-login" );
    return preg_replace( $old,
    $new, $url, 1 );
}
add_filter( 'site_url',
'schoolpress_wp_login_filter', 10, 3 );

function
schoolpress_wp_login_redirect() {
    if ( strpos( $_SERVER[ "REQUEST_URI" ],
'new-login' ) === false ) {
        wp_redirect( site_url()
);
        exit();
    }
}
add_action( 'login_init',
'schoolpress_wp_login_redirect' );

```

If you don't want to write any custom code, you can use the following plugins to achieve similar results:

- [iThemes security](#)
- [WP Admin](#)

## Add Custom .htaccess Rules for Locking Down wp-admin

If you are the only user that needs to log in to the backend of your application, or if you have only a handful of backend users, you can restrict access to the backend by certain IP addresses. Create a new *.htaccess* file in the *wp-admin* directory of your WordPress installation and add the following code, replacing *127.0.0.1* with your actual external IP address. Go to *http://ipchicken.com/* if you are not sure of your external IP address:

```
order deny,allow
allow from 127.0.0.1 #(repeat this line for multiple IP
addresses)
deny from all
```

If you suspect that certain IP addresses hitting your application are bots or malicious users, you can block them by their IP addresses by using the following code:

```
order allow,deny
deny from 127.0.0.1 #(repeat this line for multiple IP
addresses)
allow from all
```

If people really want to get around their banned IP address, they will use a proxy server.

If you think your IP address or that of your backend users may change often or you have far too many backend users to manage all of their IP addresses, you can add a separate username and password to access the *wp-admin* directory. This adds a nice second layer of authentication because all of your backend users will be required to

enter an htaccess username and password and their standard WordPress username and password:

```
AuthType Basic
AuthName "restricted area"
AuthUserFile /path/to/protected/dir/.htpasswd
require valid-user
```

Notice the `AuthUserFile` line; you will need to create a `.htpasswd` file somewhere in a directory above or outside of your WordPress install. In this file, you will need to add a username and password. The password can't just be plain text; use a tool like [htaccess password generator](#) to create an encrypted password.

So the username/password for `letmein/Pr3tTyPL3a$3!` after encryption should be `letmein:E5Dj7cUaQVcN`.

Add the entire encrypted string `letmein:E5Dj7cUaQVcN`. to your `.htpasswd` file; and when users try to go to `/wp-admin`, they will be prompted for a username and password. Make sure to let your backend users know what this username and password is and tell them not to share it with anybody.

## SSL Certificates and HTTPS

When accepting sensitive information through a web form—for example, a credit card number—you should encrypt that information by loading and submitting the form over SSL or HTTPS. First, some definitions:

## *SSL*

Stands for “Secure Sockets Layer” and is the technology that encrypts data that is transferred to and from a web page.

## *HTTP*

Stands for “Hypertext Transfer Protocol.” This is the standard *protocol* for serving web pages without encryption.

## *HTTPS*

Stands for “HTTP Secure.” This is the protocol for serving web pages with SSL encryption.

There are many options when it comes to configuring SSL on your server and installing an SSL certificate. The bottom line is that *all sites* (not just ecommerce sites) should be set up to serve *all traffic* (not just sensitive traffic) over HTTPS these days. We’ll cover a few options for doing this in the following sections.

## **Installing an SSL Certificate on Your Server**

First, make sure that you have SSL enabled on your web server. How to do that will depend on your specific host and web server.

DigitalOcean has great system administrator documents in general and a particularly good article on setting up SSL with Apache.

After enabling the SSL service on your host, you’ll need an SSL certificate to use with it. You can use self-signed certificates for testing purposes, but modern-day browsers will show some fairly dire warnings when you browse to a site using a self-signed certificate.

Figure 8-1 shows the warning displayed to Chrome users.



## Your connection is not private

Attackers might be trying to steal your information from **yourdomain.com** (for example, passwords, messages, or credit cards). [Learn more](#)

NET::ERR\_CERT\_AUTHORITY\_INVALID

**Subject:** yourdomain-com

**Issuer:** yourdomain-com

**Expires on:** May 18, 2018

**Current date:** Mar 16, 2019

**PEM encoded chain:**

-----BEGIN CERTIFICATE-----

```
MIIIEhjCCAwgAwIBAgICFYMwDQYJKoZIhvcNAQELBQAwgCMxCzAJBgNVBAYTAi0t  
MRIwEAYDVQQIDAlTb21lU3RhdGUxETAPBgNVBAcMCFNvbWVDaXR5MRkwFwYDVQQK  
DBBTb21lT3JnYW5pemF0aW9uMR8wHQYDVQQLDBZTb21lT3JnYW5pemF0aW9uYWxV  
bm10MSIwIAYDVQODDBlzaXRlcyclzdHJhbndlcnN0dWRpb3MtY29tMS0wKwYJKoZI  
hvcNAQkBFh5yb290QHNpdGVzLXN0cmFuZ2Vyc3R1ZG1vcy1jb20wHhcNMTEwNTE4
```

[Advanced](#)

[Back to safety](#)

*Figure 8-1. Chrome SSL warning*

For production environments, you'll want to use a public-key certificate from a certificate authority, or CA. You can purchase public-key certificates, though they're usually bundled or offered as an add-on to your web-hosting package. You can also use public-key certificates (SSL certificates) purchased from third parties. A good CA certificate will be trusted by all modern web browsers, which is what gives you the green or golden padlock icon on your website instead of a broken or red padlock.

There are good options now for paid or free CA certificates. What you're really doing when you use a CA certificate is confirming that you actually own the domain on which you are using the certificate. Ownership of the domain is usually confirmed via email to an address on the domain. Or, in the case of Let's Encrypt, the confirmation is done through automated scripts run from the server in question.

### LET'S ENCRYPT

Since the publication of the first edition of this book, a new service called Let's Encrypt has come online that offers public-key certificates that are easy to install and 100% free. Let's Encrypt certificates are installed from the command line using a tool called Certbot. Certbot is available for most web server platforms. If your web server is configured correctly and connected to the internet, Certbot will automatically validate itself with the Let's Encrypt certificate authority, issue the certificate, and install the certificate on your server. You can then set up a cron job to renew the certificate every three months or so.

Even if you aren't managing your own server, you may still be able to use a Let's Encrypt certificate with your web host. While the paid options are likely to be more visible in your control panel, there are often Let's Encrypt options when setting up SSL or you may be able to submit a support request to have your host set up a free Let's Encrypt SSL for your site.

*Again, every website should install an SSL certificate.* In the first edition of this book, we covered methods to serve the admin and checkout pages over HTTPS while the rest of the site was served over unsecured HTTP. We no longer recommend this kind of hybrid setup. The internet has moved to a place where it is assumed that *all websites* are served entirely over HTTPS. This is part of a larger “security by default” movement.

There are a number of reasons to set up HTTPS on your entire site:

- Security by default. You might imagine that only your login and checkout pages really need to be secure, but what happens when your site is updated to show a login form in the sidebar? Now every page on your site needs to be secure. If your entire site is served over HTTPS, you won’t accidentally introduce an unencrypted form anywhere on the site.
- Internet consumers are trained to look for that padlock (see Figure 8-2). Both savvy and nonsavvy users will feel better seeing it. Further, modern browsers will show some pretty scary warnings if parts of your site are not served over HTTPS.
- Google and other search engines have started boosting sites that are served entirely over HTTPS in their search rankings.
- There is no longer a CPU hit to your server when you use HTTPS. The web server stack has been updated to better handle HTTPS and frankly expect it, so you can no longer use page load times as an excuse to disable HTTPS on your site.

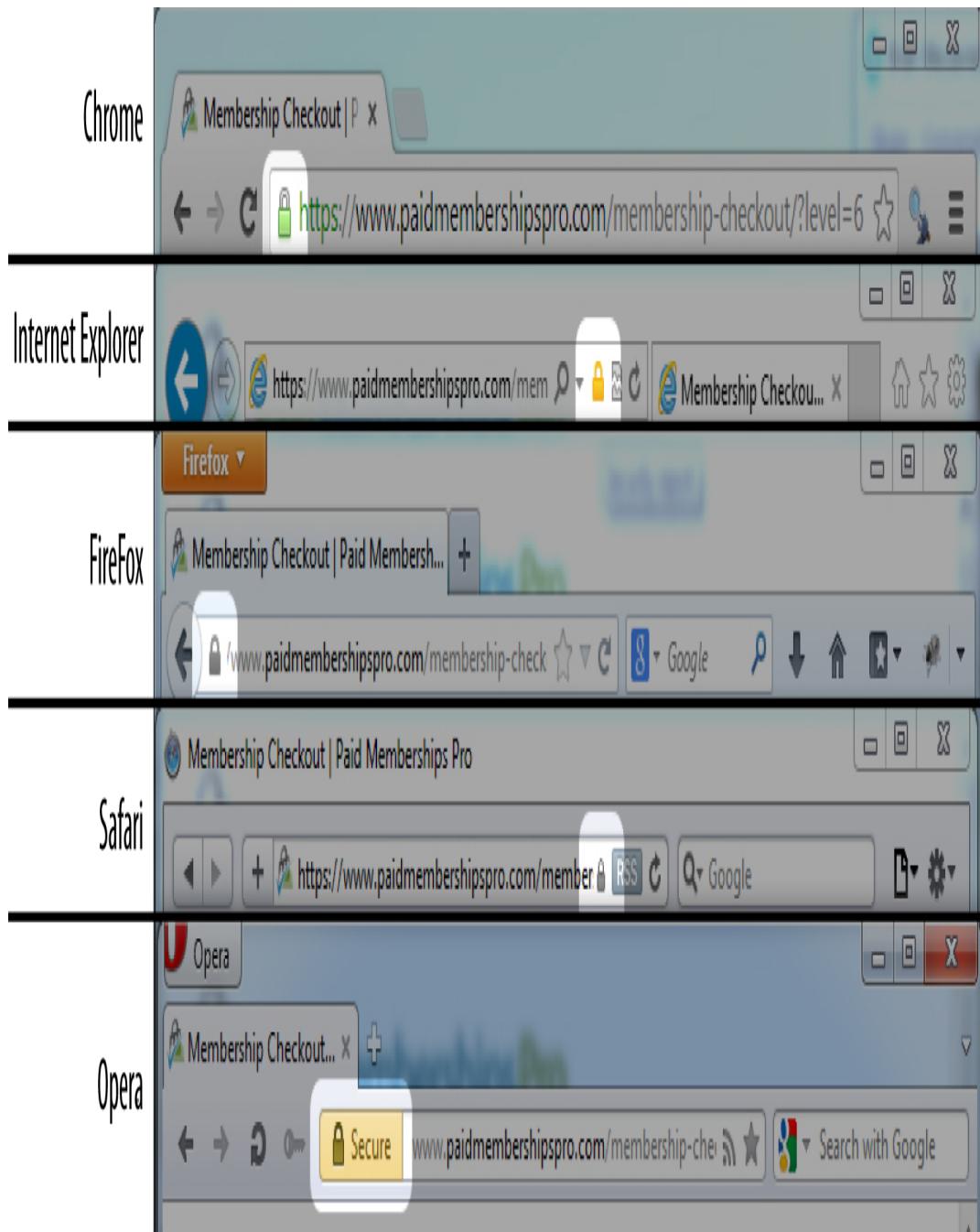


Figure 8-2. Various padlocks across browsers

## USING ONE DIRECTORY FOR HTTPS AND HTTP TRAFFIC

Besides using a CA certificate, the other thing to do when setting up SSL is to have your HTTPS directory point to your HTTP directory

through a *symbolic link*, or *symlink* for short. A symlink is like a shortcut in a Windows PC. The symlink points to another directory rather than being a directory of its own.

The end result of using a symlink for your HTTPS directory is that the same *.php* source files will be loaded when people visit *https://yoursite.com* as when they visit *http://yoursite.com*. Your server will make sure that the traffic through the HTTPS link is encrypted and both WordPress and any ecommerce plugin you may be using will make sure that the correct secure page is shown to the user when being served over SSL.

Assuming your HTTP directory is called *html* and you want your HTTPS directory to be called *ssl\_html*, you would issue the following Linux command to create a symlink to that directory: `ln -s http ssl_http`.

## **WordPress Login and WordPress Administrator over SSL**

Serving your checkout page over SSL is the minimum you can do to secure the private data passed to and from your site. You can also set up WordPress to use SSL on the login page, in the administrator dashboard, across the entire site, or only on select pages.

For SSL logins in WordPress, you set the `FORCE_SSL_LOGIN` constant to `true` in your *wp-config.php* file. Place the following line of code above the “That’s all, stop editing! Happy blogging.” comment at the end of the file:

```
define('FORCE_SSL_LOGIN', true);
```

To use SSL on the login page *and* in the administrator dashboard, use the following `FORCE_SSL_ADMIN` constant instead:

```
define('FORCE_SSL_ADMIN', true);
```

### NOTE

The `FORCE_SSL_ADMIN` constant supersedes the `FORCE_SSL_LOGIN` constant. You should set only one or the other constant to `true`. If `FORCE_SSL_LOGIN` is `false` and `FORCE_SSL_ADMIN` is `true`, your login page will still be served over SSL.

## Debugging HTTPS Issues

Now we're going to write a quick little function to filter URLs generated by WordPress to use the same protocol as the current page. Remember earlier that we talked about how URLs like `https://yoursite.com/some-page` (HTTP) that show up on a page like `https://yoursite.com/checkout` (HTTPS) will cause your browser to show a security warning:

```
function my_https_filter($s) {
    if(is_ssl())
        return str_replace("http:", "https:", $s);
    else
        return str_replace("https:", "http:", $s);
}
add_filter('bloginfo_url', 'my_https_filter');
add_filter('wp_list_pages', 'my_https_filter');
add_filter('option_home', 'my_https_filter');
add_filter('option_siteurl', 'my_https_filter');
```

```
add_filter('logout_url', 'my_https_filter');
add_filter('login_url', 'my_https_filter');
add_filter('home_url', 'my_https_filter');
```

The `is_ssl()` function provided by WordPress will return `true` if the current page was loaded over HTTPS. The `is_ssl()` function checks specifically if the `$_SERVER['HTTPS']` global is set to `on` or `1` or if the `$_SERVER['SERVER_PORT']` global is set to `443`. Some server setups behind load balancers or reverse proxies will load HTTPS pages without setting those globals properly in PHP. You can fix this by adding the following code to your `wp-config.php` file.

```
if (isset($_SERVER['HTTP_X_FORWARDED_PROTO']))
    && $_SERVER['HTTP_X_FORWARDED_PROTO'] ==
'https') {
    $_SERVER['HTTPS'] = 'on';
}
```

Our https filter uses the `str_replace()` function to swap “`http:`” for “`https:`”, or vice versa. We set this filter to run on a number of built-in WordPress hooks used at various places throughout the WordPress codebase where URLs are generated.

When you output URLs in other places of your custom application code, be sure to use the `home_url()` function to make sure the URL is generated correctly and the `my_https_filter` is run on it.

## Avoiding SSL Errors with the “Nuclear Option”

The `my_https_filter()` function ensures links that show up on a page use the correct protocol. However, sometimes raw `http://...` URLs may be hardcoded into your posts, or maybe a plugin you use doesn't use the built-in WordPress functions like it should when outputting same-site URLs or loading JavaScript or CSS files.

Figure 8-3 shows the Chrome Developer Tools Console, which can help locate errors.

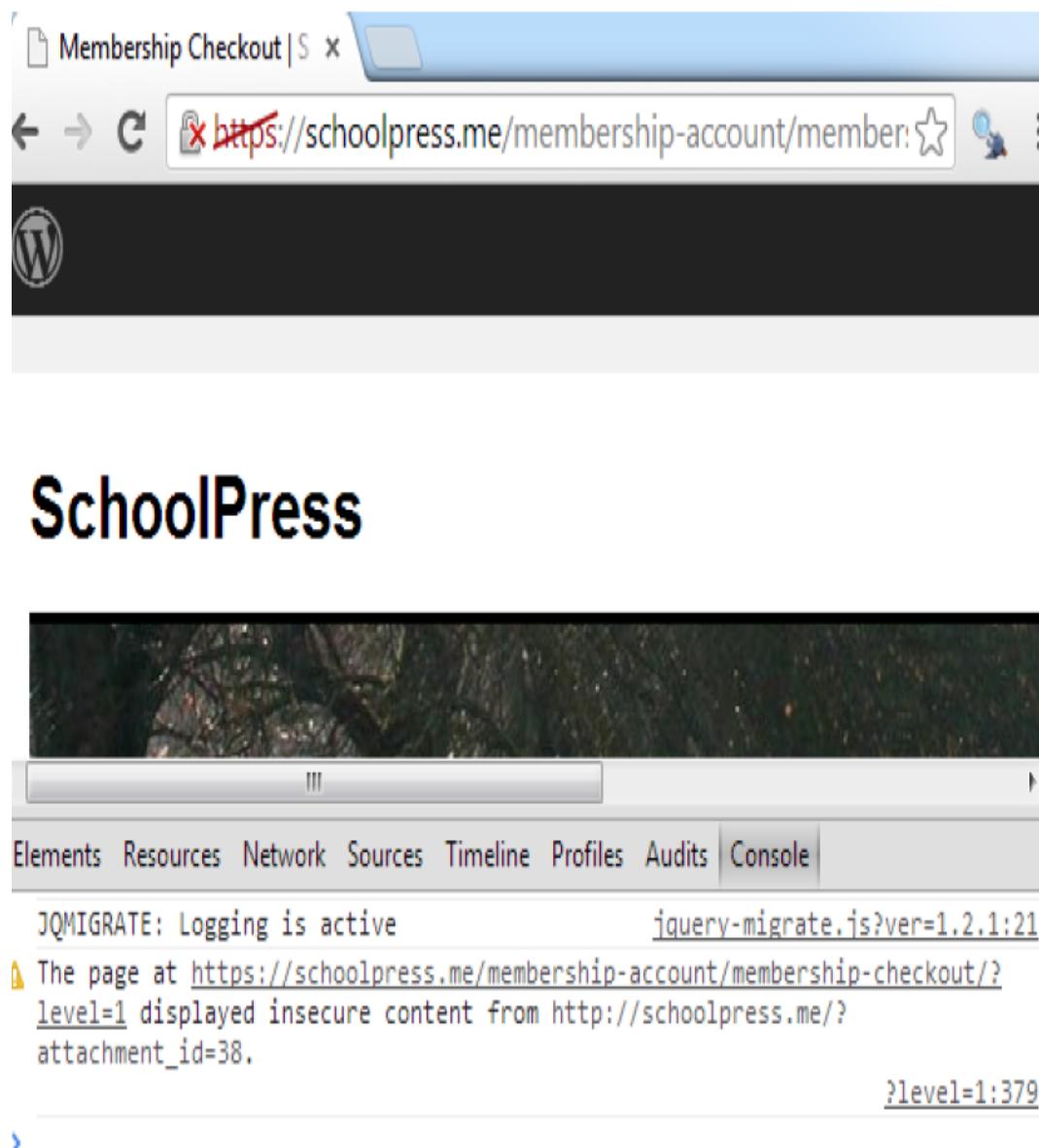


Figure 8-3. An SSL error in the Chrome Developer Tools Console

In these cases, you can try to find each case of a bad URL and fix the link in your posts or code to use a relative URL or the proper WordPress function to make sure it will output on the frontend using the proper protocol. However, it's sometimes easier to use what we call the Nuclear Option:

```
constant('MY_SITE_DOMAIN', 'yoursite.com');

function my_NuclearHTTPS() {
    ob_start("my_replaceURLsInBuffer");
}

add_action("init", "my_NuclearHTTPS");

function my_replaceURLsInBuffer($buffer) {
    global $besecure;

    //only swap URLs if this page is secure
    if(is_ssl())
    {

/*
okay swap out all links like these:
* http://yoursite.com
* http://any.subdomain.yoursite.com
* http://any.number.of.sub.domains.yoursite.com
*/
    $buffer = preg_replace(
        '/http\/\/([a-zA-Z0-
9\.\-]*).str_replace('.','\.',MY_SITE_DOMAIN).'')/i',
        'https://$1',
    $buffer
    );
}

return $buffer;
}
```

First, we need to make sure we define a constant MY\_SITE\_DOMAIN and set it to the second-level domain (SLD) for

your site. Your `site_url()` set in WordPress may be `http://<www.yoursite.com`, but we are interested here in just the `yoursite.com` part of that.

Then, `my_NuclearHTTPS()` fires on the `init` hook and uses the PHP function `ob_start()` to turn on output buffering. Output buffering means that all output generated by PHP (e.g., via echo function calls or inline HTML) goes into a buffer string instead of straight to the browser. Then, when PHP is finished generating all output (or if you call the `ob_end_flush()` function first), the buffer string is passed to a callback function, which is `my_replaceURLsInBuffer()` in this case.

The `my_replaceURLsInBuffer()` function filters the buffer string, swapping out “`http:`” for “`https:`” on *every* link. The regular expression magic we’re doing in the `preg_replace()` call there makes sure that links to any subdomain using the same domain (why we needed to set the `MY_SITE_DOMAIN` constant) will also be filtered.

So you might have caught on by now why we call this the “Nuclear Option.” Instead of finding the source of bad URLs in your app and fixing them, we just fix all of the URLs at once before sending the output to the browser. There will be a small performance hit here, depending on how large your HTML output is. But this method can be useful in a pinch, especially if you are using many third-party plugins that you can’t or don’t want to fix to output site URLs properly.

## NOTE

The [Really Simple SSL plugin](#) includes many of the HTTPS fixes we've mentioned, as well as other tools to help you get HTTPS working properly on your WordPress site.

## Back Up Everything!

It is important to make regularly scheduled backups of your site's content (your database) as well as the *wp-content* folder. This makes it much easier to restore your site in the event that it does fall victim to a hacker. We recommend scheduling a backup at the very least once a week, but depending upon how much new content you are adding, you may feel that you need to increase or decrease the frequency. Of course, a daily backup is always the best choice.

## NOTE

### Do You Know Where Your Backups Are?

Do you know if you can really use your backups to recover your site? Every few months, try to rebuild your site from your backups. This ensures that they're really working, that you're backing up everything you need to, and that you can quickly restore your site from backups. There are plenty of horror stories told by people who thought they had a backup plan only to find that the backups were corrupt, incomplete, or otherwise not useful for restoring the site.

## Scan, Scan, Scan!

Scanning or monitoring your application is essential to know whether you have been attacked. If your application is ever hacked, it is important to know immediately, so you can quickly address the issue.

Be proactive about protecting your web application against malware. There are several services that will scan your web applications for you so you can take a more hands-off approach. We recommend using [Sucuri](#). Not only will Sucuri find malware and alert you if your application has been infected, but it will also clean it up for you. Tony Perez, the COO of Sucuri, is also a former US Marine and a martial arts master, so why wouldn't you want Sucuri to have your back? Sucuri also has a great security plugin for WordPress.

## Useful Security Plugins

In the following sections are some more useful and powerful WordPress plugins to help you increase security for your application and also help you to recover quickly if you fall victim to a malicious attack.

### Spam-Blocking Plugins

Spam is a problem for every website on the internet. These plugins will help.

#### AKISMET

This plugin is used to block comment spam from getting through to your site. It was developed by Automattic, also the creators of WordPress, and therefore comes standard with any new WordPress

install. Although the plugin will be installed on your site, you will need to activate it by registering for an API key at <https://akismet.com/>. An API key is free if your site is for personal use; however, there is a small charge for business websites. The way Akismet works is each time a comment is posted to your site, Akismet will run it through a series of tests to ensure it is a real comment; if it is identified as spam, it is automatically moved to the spam folder in your dashboard. This saves you tons of time having to sort through all of your comments and determine which ones are spam and which are legitimate comments.

## BAD BEHAVIOR

The Bad Behavior plugin works to block link spam from your site and functions best when run in conjunction with another spam service. It looks not only at the content of the spam, but also at the method through which the spam is being delivered by the spammer and the software being used, and blocks that as well.

## Backup Plugins

Backups are very helpful to have in the event that your site is compromised. Here are a few popular backup plugins.

## BACKUPBUDDY

As we mentioned in Chapter 3, BackupBuddy is a premium plugin that lets you make backups of all your WordPress site content for safekeeping, restoring, or moving your site. Backups can be scheduled and the files emailed to you or sent to a storage site such as Dropbox or an FTP server. This plugin's restore option easily restores

themes, widgets, and plugins. With BackupBuddy you can use the WordPress dashboard to move your site to a new server or domain, a handy feature if you work on a Dev server and move the sites to a Production environment upon launch.

## VAULTPRESS

VaultPress is another plugin created by the team at Automattic and offers users the opportunity to have all of their site content backed up in real time on cloud servers. Once installed, this plugin will automatically detect any changes to the content on your site as well as site settings and then update the backup copy with those changes. The plugin also features a one-click database restore in the event that your site ever becomes compromised. This is a premium plugin, meaning there is a fee for service, and different levels are offered. The premium version of the plugin also includes a daily security scan of your site to detect any issues as well as fixes for those issues.

## Firewall/Scanner Plugins

The plugins that follow are useful for detecting and mitigating the kinds of automated attacks that plague every website on the internet.

## WORDFENCE

WordFence works as a type of firewall for your site by scanning incoming traffic and then blocking all kinds of different malicious requests. You can also perform an on-demand scan of your site and detect any areas of vulnerability in your site's security. Upgrade to the paid version to use some of the premium tools and use a more up-to-date database of malware and vulnerabilities.

## **ALL IN ONE WP SECURITY & FIREWALL**

The All In One WP Security & Firewall plugin has a firewall and scanner similar to WordFence's. It also has tools to harden your login and user security. One important security issue this plugin helps with is changing your database table prefix, which can be tricky if you are not that familiar with the standard database structure.

## **EXPLOIT SCANNER**

Maintained by Automattic, the Exploit Scanner plugin will scan through all the files on your site and then alert you if it finds anything that looks like it could be a potential threat.

## **Login and Password-Protection Plugins**

The plugins here deal specifically with limiting access to the WordPress login page and dashboard in general.

## **LIMIT LOGIN ATTEMPTS**

The Limit Login Attempts plugin is great for fighting off brute-force attacks, such as someone running an automated script that will try to log in to WordPress using random combinations of words. By default, WordPress will allow an unlimited amount of login attempts, and this plugin limits the number of those attempts. If someone tries  $x$  times to log in and fails each time, they will be blocked from attempting to log in again for a set amount of time.

## **ASKAPACHE PASSWORD PROTECT**

[AskApache Password Protect](#) is different from other WordPress security plugins in that it works at the network level to prevent attacks, rather than at the site level. You choose a unique username and password that then protect your login page and entire *wp-admin* folder. This plugin does require the use of an Apache web server and web host support for *.htaccess* files.

## Writing Secure Code

You want to make sure any custom code you write is secure and not hackable. If you take notice and apply the following methods, you should be in pretty good shape against attacks.

### Check User Capabilities

Each of your users has unique standard or custom roles and capabilities. If you are writing some code that provides custom functionality for your application's administrators, make sure to give access to administrators, and only administrators. There are a few built-in WordPress functions for telling you whether a user has certain roles or capabilities. All of these functions are located in *wp-includes/capabilities.php* and return a Boolean of whether the user has the passed-in role name or capability. You can pass in any default or custom-made roles or capabilities.

#### **USER\_CAN( \$USER, \$CAPABILITY )**

Whether a particular user has a particular role or capability.

*\$user*

A required integer of a user ID or an object of the user.

*\$capability*

A required string of the capability or role name.

## **CURRENT\_USER\_CAN( \$CAPABILITY )**

Whether the current user has a particular role or capability.

*\$capability*

A required string of the capability or role name.

## **CURRENT\_USER\_CAN\_FOR\_BLOG( \$BLOG\_ID, \$CAPABILITY )**

Whether the current user has a particular role or capability for a particular site on a multisite network.

*\$user*

A required integer of a blog ID.

*\$capability*

A required string of the capability or role name.

In the following code, we don't want to let ordinary users into the backend of our application. We want them to interact only with the custom UI we created within the theme on the frontend, so we will redirect anybody who is not an administrator and may wander to `/wp-admin` back to the frontend:

```
function schoolpress_admin_check() {
    global $user_ID;
    if ( ! user_can( $user_ID, 'administrator' ) ) {
```

```
        wp_redirect( site_url() );
    }
}

add_action( 'admin_init', 'schoolpress_admin_check' );
```

## NOTE

Another common practice in many WordPress plugins is to test for the *manage\_options* capability rather than the administrator role. On a default WordPress install, only the administrator has this capability anyway, but checking for *manage\_options* instead of *administrator* will ensure that your check works on sites with custom roles.

For a complete reference of standard default WordPress roles and capabilities, see [Chapter 6](#) or the [WordPress Codex page](#).

## Custom SQL Statements

Sometimes the built-in WordPress functions that interact with the database may not be enough for your needs, and depending on what you are building, you may want to write custom SQL statements.

When writing your own SQL statements, you need to make sure you do so in a way that will not allow for any potential SQL injections. First, always use the `$wpdb` object and make sure to escape and prepare all custom SQL statements.

As we talked about in [Chapter 3](#), the `$wpdb` object can be used to access any standard or custom tables in your WordPress database and provides easy-to-use methods for doing so. One very important thing to remember is that when writing custom queries with any dynamic

values being passed in, you need to use the `esc_sql()` function or the `prepare()` method to sanitize and escape those dynamic values. By sanitizing and escaping dynamic values, you are making sure those values are not made up of invalid characters or malicious SQL code that can hijack your query (SQL injections).

The `esc_sql()` and `$wpdb->prepare()` functions are covered in detail in [Chapter 3](#).

## Data Validation, Sanitization, and Escaping

*Do not trust your users!* Again: *do not trust your users!* Don't be that web application, website, or blog that spreads malware.

Validate, sanitize, and escape every piece of data going into and coming out of your database. You want to make sure that the data your users are submitting to your database is in the format it should be in; the database doesn't care what the data is as long as the data being submitted is of the same datatype.

For example, suppose that you have a custom form used to collect user data with a textbox for date of birth (DOB). You plan on storing the DOB as user meta in the `meta_value` column of the `wp_usermeta` table. The `meta_value` column has a datatype of `longtext`, meaning the value can be super-duper long<sup>1</sup> and the database isn't going to care what value you store there. It's up to you as the developer to make sure the data being stored as DOB is a date and nothing else.

So what exactly is the difference between validation, sanitization, and escaping?

### *Validating*

The process of making sure the data received from the end user is in the correct format you expect it to be in. You want to validate data before saving it into the database.

### *Sanitizing*

The process of cleaning data received from the end user before saving it to the database or using it in your app.

### *Escaping*

The process of cleaning data you may already have before displaying it to the end user, saving it to the database, or passing it off to an API.

Now you know!

You want to validate and sanitize any data submitted to your app through form submissions, URL parameters, or API calls. You want to escape any data before putting it into your database or echoing it out to the screen. When pulling data out of your database, you want to sanitize it just to be safe in case somehow you are storing unsanitized data.

PHP has validation and sanitization functions, but WordPress has its own helper functions built in. This is a book about WordPress, so let's talk about some of those functions.

## NOTE

Most sanitization and escaping helper functions are located in *wp-includes/formatting*.

### **ESC\_URL( \$URL, \$PROTOCOLS = NULL, \$\_CONTEXT = 'DISPLAY' )**

Checks and cleans a URL by checking whether it has the proper protocol, stripping invalid characters, and encoding special characters. Use this if displaying a URL to an end user:

*\$url*

A required string of the URL that needs to be cleaned.

*\$protocols*

An optional array of whitelisted protocols. Defaults to array( http, https, ftp, ftps, mailto, news, irc, gopher, nntp, feed, telnet, mms, rtsp, svn ) if not specifically set.

*\$context*

An optional string of how the URL is being used. Defaults to display, which sends the URL through wp\_kses\_normalize\_entities() and replaces & with &#038; and ' with &#039;.

### **ESC\_URL\_RAW( \$URL, \$PROTOCOLS = NULL )**

This function calls the esc\_url() function but passes db as the value for the \$\_context parameter. Do not use this function for

displaying URLs to the end user; only use it in database queries.

### **ESC\_HTML( \$TEXT )**

Escape HTML blocks in any content. This function is a nice little wrapper for the `_wp_specialchars()` function, which basically converts a number of special characters into their HTML entities:

*\$text*

A required string of the text you want to escape HTML tags on.

### **ESC\_JS( \$TEXT )**

Escapes strings in inline JavaScript. Escaped strings need to be wrapped in single quotes for this to work:

*\$text*

A required string of the text you want to escape single quotes, HTML special characters ( "`<` `>` `&` ), and fix line endings on.

### **ESC\_ATTR( \$TEXT )**

Escapes HTML attributes and encodes such characters as `<`, `>`, `&`, `"`, and `'`. This is important to use when including values in form input elements such as ID, name, alt, title, and value:

*\$text*

A required string of the text you want to escape HTML attributes on.

### **ESC\_TEXTAREA( \$TEXT )**

Escaping for `textarea` values. Encodes text for use inside a `<textarea>` element:

*\$text*

A required string of the text you want to escape HTML on.

### **SANITIZE\_OPTION( \$OPTION, \$VALUE )**

This function can be used to sanitize the value of any predefined WordPress option. Depending on what option is being used, the value will be sanitized via various functions:

*\$option*

A required string of the name of the option.

*\$value*

A required string of the unsanitized option value you wish to sanitize.

### **SANITIZE\_TEXT\_FIELD( \$STR )**

Sanitizes any string input by a user or pulled from the database. Checks for invalid UTF-8; converts single < characters to entity; strips all tags; removes line breaks, tabs, and extra whitespace; and strips octets:

*\$str*

The required string you want to sanitize.

### **SANITIZE\_USER( \$USERNAME, \$STRICT = FALSE )**

This function cleans a username of any illegal characters:

*\$username*

A required string of the username to be sanitized.

*\$strict*

An optional Boolean that, if set to `true`, will limit the username to specific characters.

### **SANITIZE\_TITLE( \$TITLE, \$FALLBACK\_TITLE = "" )**

Sanitizes a title stripping out any HTML or PHP tags, or returns a fallback title for a provided string:

*\$title*

A required string of the title to be sanitized.

*\$fallback\_title*

An optional string to use if the title is empty.

### **SANITIZE\_EMAIL( \$EMAIL )**

Sanitizes an email address by stripping out any characters not allowed in an email address:

*\$email*

The email address to be sanitized.

### **SANITIZE\_FILE\_NAME( \$FILENAME )**

Sanitizes a filename, replacing whitespace with dashes. Removes special characters that are illegal in filenames on certain operating

systems and special characters requiring special escaping to manipulate at the command line. Replaces spaces and consecutive dashes with a single dash. Trims period, dash, and underscore from beginning and end of filename:

*\$filename*

Required string of the filename to be sanitized.

**WP\_KSES( \$STRING, \$ALLOWED\_HTML,  
\$ALLOWED\_PROTOCOLS = ARRAY () )**

This function makes sure that only the allowed HTML element names, attribute names, and attribute values plus only sane HTML entities will occur in the string you provide. You have to remove any slashes from PHP's magic quotes before you call this function:

*\$string*

A required string that you want filtered through kses.

*\$allowed\_html*

A required array of allowed HTML elements.

*\$allowed\_protocols*

An optional array of allowed protocols in any URLs in the string being filtered. The default allowed protocols are http, https, ftp, mailto, news, irc, gopher, nntp, feed, telnet, mms, rtsp, and svn. This covers all common link protocols, except for javascript, which should not be allowed for untrusted users.

**WP\_KSES\_POST( \$DATA )**

This function sanitizes the data passed in, allowing the same HTML tags and protocols that are allowed in the post content section of the edit post page. If you want to use stricter rules than you would for authors, editors, and admins on your site, use the `wp_kses()` function with specific tags and protocols passed in. If your field is meant for administrators and you just want to sanitize a field the same way the post content is sanitized, the `wp_kses_post()` function is a good shortcut.

`$data`

A required string that you want filtered through `kses`.

The following code validates and sanitizes an email address:

```
// pretend a user added an email address "jason @
stranger$studios.com"
$user_email = 'jason @ stranger$studios.com';

// we can check if this is a valid email
$valid_email = is_email( $user_email );

// we know it's not because it's set to nothing from
is_email()
if ( ! $valid_email )
    echo 'invalid email<br />';

// let's try again with sanitizing the email
$user_email = 'jason @ stranger$studios.com';

// use sanitize_email() to try to fix any invalid email
$user_email = sanitize_email( $user_email );

$valid_email = is_email( $user_email );

if ( ! $valid_email )
    echo 'invalid email<br />';
```

```
else
    echo 'valid email: ' . $user_email;
```

Notice that in this example that the `sanitize_email()` function removes both the spaces and dollar sign in the invalid email. While the returned email address is technically valid, it's not Jason's real email address since the function doesn't understand leet-speak well enough to swap the \$ with an s. Also note that the returned value won't always be a valid email address. If there is no @ sign, no text before the @ sign, or no domain behind the @ sign, then the returned value will be an invalid email.

Additional information on validating, sanitizing, and escaping data can be found in the [WordPress Codex](#).

## Nonces

*Nonce* means “number used once,” and using nonces is critical to protecting your application from cross-site request forgery (CSRF) attacks. Normally your server-side scripts for form processing are processing forms from your own site. People visit your site, log in, and submit a form to perform some action on your site. However, if your server-side code were simply looking for `$_POST` values to determine what to do, those values could be submitted from *any* form, even forms on other websites.

The first line of defense is to check that a user is really logged in and has the capabilities to do the requested action. However, this isn't enough to stop CSRF attacks, because you might be logged in on your WordPress site (e.g., in a background tab) while some malicious

code on another site/tab kicks off the form request with the correct `$_POST` variable to send a spammy message to your friends or initiate account deletion or something else you don't want to do.

What's needed is a way to make sure that the request comes from the WordPress site and not another site. This is what a nonce does. The basic outline of using a nonce is as follows:

1. Generate a nonce string every time a page is loaded.
2. Add the nonce string as a hidden element on the form.
3. When processing a submitted form, generate the nonce the same way and check that it matches the one submitted from the form.

Because the nonce is generated using a combination of the secret salt keys in your `wp-config.php` and the server time, it is difficult for attackers to guess the nonce string for their spoofed forms.

Nonces are useful for nonform links and Ajax calls as well. The process is basically the same:

1. Generate a nonce string every time a page is loaded.
2. Add the nonce string as a parameter to the URL.
3. When processing the request, generate the nonce the same way and check that it matches the one submitted through the URL.

Whether protecting your forms, links, or Ajax requests, WordPress has a few helper functions to make this process very easy to implement.

## **WP\_CREATE\_NONCE( \$ACTION = -1 )**

This function will create a random token that can only be used once, and which is located in *wp-includes/pluggable.php*:

*\$action*

An optional string or int that describes what action is being taken for the nonce created. You should always set an action to be more secure:

```
function schoolpress_footer_create_nonce() {
    $nonce = wp_create_nonce('random_nonce_action');
    $url = add_query_arg( array( 'sp_nonce' =>
$nonce ) );
    echo '<p><a href="' . $url . '">Verify this
Nonce</a></p>';
}
add_action( 'wp_footer',
'schoolpress_footer_create_nonce' );
```

## **WP\_VERIFY\_NONCE( \$NONCE, \$ACTION = -1 )**

This function is used to verify that the correct nonce was used within the allocated time limit. If the correct nonce is passed into this function and everything checks out OK, the function will return a value that evaluates to `true`.<sup>2</sup> If not, it will return `false`. This function is located in *wp-includes/pluggable.php*:

*\$nonce*

A required string of the nonce value being used to verify.

*\$action*

An optional string or int that should be descriptive to what is taking place and should match the action from when the nonce was created.

```
function schoolpress_init_verify_nonce() {
    if ( isset( $_GET['sp_nonce'] ) )
        && wp_verify_nonce( $_GET['sp_nonce'],
        'random_nonce_action' ) ) {
        echo 'You have a valid nonce!';
    } else {
        echo 'You have an invalid nonce!';
    }
}
add_action( 'init', 'schoolpress_init_verify_nonce' );
```

## CHECK\_ADMIN\_REFERER( \$ACTION = -1, \$QUERY\_ARG = '\_WPNONCE' )

This function calls the `wp_verify_nonce()` function, so it verifies nonces but also checks to see that the *referrer*, or the page that got you to the current page, is from the same website. This function is located in *wp-includes/pluggable.php*:

`$action`

An optional string, but you should specify a nonce action to be verified.

`$query_arg`

An optional string of the query argument that has the nonce as its value.

```
// checking the same nonce "sp_nonce" that was created
earlier
function schoolpress_init_check_admin_referer() {
    if ( isset( $_GET['sp_nonce'] ) ) &&
```

```

        check_admin_referer( 'random_nonce_action',
'sp_nonce' ) ) {
    echo '<p>You have a valid nonce!</p>';
} else {
    echo '<p>You have an invalid nonce!
</p>';
}
add_action( 'init',
'schoolpress_init_check_admin_referer' );

```

## WP\_NONCE\_URL( \$ACTIONURL, \$ACTION = -1 )

This function also utilizes the `wp_create_nonce()` function and adds a nonce to any URL. If you create any actions based off of a query string, you should always tie a nonce to your URL with this function:

`$actionurl`

A required string of the URL to which to add a nonce action.

`$action`

An optional string for the action name. You should always set this.

This function is located in `wp-includes/functions.php`:

```

// simple url with querystring example
function schoolpress_footer_nonce_url() {
    $url = wp_nonce_url(
        add_query_arg( array( 'action' =>
'get_users' ) ),
        'get_users_nonce'
    );
    echo '<p><a href="' . esc_url( $url ) . '">Get
Users</a></p>';
}

```

```

}

add_action( 'wp_footer', 'schoolpress_footer_nonce_url' );

// querystring action
function schoolpress_footer_nonce_url_action(){
    // check if querystring action is get_users and for
    // the nonce
    if ( isset( $_GET['action'] )
        && 'get_users' == $_GET['action']
        && check_admin_referer( 'get_users_nonce' ) )
    {
        echo 'Your action: ' . esc_html(
        $_GET['action'] );
        // or get your users and display them
        here...
    }
}
add_action( 'init', 'schoolpress_footer_nonce_url_action' );

```

Notice in this example that we used the `esc_html()` function when echoing the action that was passed into the query string. Normally, we haven't been using the escape functions in our example code because they make it harder to read and understand what the code is doing. However, this *is* the security chapter and carelessly echoing URL parameters is one of the most common ways of introducing cross-site scripting vulnerabilities into your site.

Notice also, we use the `esc_url()` function when echoing the nonce URL into the link. Even when building a URL using the functions provided by WordPress, you must escape the URL before sending it to output. The `esc_url()` function will prevent itself from being run multiple times and ruining the URL.

**WP\_NONCE\_FIELD( \$ACTION = -1, \$NAME =  
"\_WPNONCE", \$REFERER = TRUE , \$ECHO = TRUE )**

This function retrieves or displays a hidden nonce field in a form. The `wp_create_nonce()` function is baked into it, so you should always use this nice helper function when dealing with forms.

The nonce field is used to validate that the contents of the form came from the location on the current site and not somewhere else. The nonce does not offer absolute protection, but should protect against most cases. It is very important to use a nonce field in forms.

The `$action` and `$name` parameters are optional, but if you want to have better security, it is strongly suggested to set those two parameters. It is easier to just call the function without any parameters, because validation of the nonce doesn't require any parameters, but since crackers know what the default is, it won't be difficult for them to find a way around your nonce and cause damage.

The input name will be whatever `$name` value you gave. The input value will be the nonce creation value. This function is located in `wp-includes/functions.php`:

`$action`

An optional string for the action name. You should always set this.

`$name`

An optional string for the nonce name. You should always set this.

`$referer`

An optional Boolean of whether to set the referer field for validation. The default value is `true`.

`$echo`

An optional Boolean of whether to display or return a hidden form field. The default value is `true`.

```
<?php
// simple submission form example
function schoolpress_footer_form() {
    ?>
    <form method="post">
        <?php // create our nonce
        wp_nonce_field( 'email_list_form',
'email_list_form_nonce' );
        ?>
        <h3>Join our email list</h3>
        Email Address: <input type="text"
name="email_address">
        <input type="submit" name="submit_email"
value="Submit" />
    </form>
    <?php
}
add_action( 'wp_footer', 'schoolpress_footer_form' );

// form action
function schoolpress_footer_form_action() {
    if ( isset( $_POST['submit_email'] ) )
        && isset( $_POST['email_address'] )
        && check_admin_referer( 'email_list_form',
'email_list_form_nonce' ) ) {
        echo 'You submitted: ' . esc_html(
$_POST['email_address'] );
        // or process your form here...
    }
}
add_action( 'init', 'schoolpress_footer_form_action' );
?>
```

## **CHECK\_AJAX\_REFERER( \$ACTION = -1, \$QUERY\_ARG = FALSE, \$DIE = TRUE )**

When using Ajax, you should still be using nonces. This function allows you to do a nonce and referer check while processing an Ajax request. This function is located in *wp-includes/pluggable.php*:

*\$action*

An optional string of the nonce action being referenced.

*\$query\_arg*

An optional string of where to look for nonce in *\$\_REQUEST*.

*\$die*

An optional Boolean of whether you want the Ajax script to die if an invalid nonce is found.

Throughout the book, you may have noticed code snippets that didn't use nonces or sanitize data. We did this to try to keep the code examples short and sweet, but you should *always* use nonces and sanitize your data. Any custom form submission or URL with custom query strings should utilize nonces, and every time you write *\$\_POST['anything']* or *\$\_GET['anything']*, they should be wrapped in a sanitization or escaping function.

---

<sup>1</sup> In technical terms, “super-duper long” is equal to about 4 GB of data.

<sup>2</sup> The *wp\_verify\_nonce()* function will return 1 if the nonce is under 12 hours old. If the nonce is between 12 and 24 hours old, it will return 2. If it is older than 24 hours old, it will return *false*. This way you can test whether the result evaluates to *true* or, to check for a slightly fresher nonce, you could check if it is equal to 1 exactly.



# Chapter 9. JavaScript Frameworks and Workflow

---

JavaScript is a major component of any modern web app. With the rise of *Node.js* to run JavaScript on the server and powerful frameworks like React, JavaScript is quickly becoming the primary component of any modern web app.

In 2012, the ratio of PHP to JavaScript code in version 3.6 of WordPress was about 6:7 PHP and 1:7 JavaScript. In 2018, the Gutenberg plugin that introduced the new block editor in WordPress 5.0 has those numbers reversed and then some with only 1:9 of the code in PHP versus 8:9 JavaScript. The block editor obviously is built on top of the server-side PHP, but it's very telling that large new features in WordPress are being built primarily in JavaScript. As the block editor becomes integrated into other aspects of the WordPress dashboard, expect the amount of JavaScript in core WordPress to go up.

Why the big move to JavaScript? On the frontend side of things, rendering a website with JavaScript can be much lighter than rendering it with PHP. As you navigate around the typical website, loading all of the HTML DOM is pretty wasteful. The header, footer, menu, and other pieces of the site may not change at all. With JavaScript, you can simply load the new part of the website, change the class on the items in your menu, and voilà: new page. This is a

much more app-like experience and perfect for using web apps over mobile networks where bandwidth is more scarce.

### NOTE

Using Ajax to update pages instead of loading new pages is sometimes referred to as building a single-page application (SPA).

Before diving into the world of JavaScript, we feel we need to express how difficult it has been to write this chapter and keep it up to date. The programming language and platform commonly referred to as JavaScript is a complicated collection of standards, frameworks, and tools.

Technically, each separate browser has its own version of JavaScript with varying support for different features. The browser vendors, along with other stakeholders, work on the Ecma International Technical Committee 39 (TC39) to publish a standard for JavaScript. The standard is actually called ECMAScript, and JavaScript is just one implementation of it.<sup>1</sup> The latest version of ECMAScript is ECMAScript2018 (or ES2018 or ES9), but “modern JavaScript” is often referred to as ES6. This version of ECMAScript was released in 2015; ES6 was the first to include many of the new features that JavaScript developers are excited about.

Confused yet? We really wanted to cut through all of this nonsense and just explain JavaScript in as straightforward a way as possible,

but you'll come across some of these terms while trying to build your apps and we wanted you to at least recognize them.

In this chapter, we will define some of the most common terms and tools you'll run into while programming JavaScript for WordPress. Then we will dive deeper into the specific methods we and other WordPress developers have been using for the past few years. Finally we explore a few of the newer JavaScript frameworks and workflows that have become popular in the WordPress community.

#### NOTE

The term “JavaScript,” when used in this chapter and throughout the book, refers to any code written in any version of JavaScript that runs in the client browser, including jQuery and Ajax calls done in jQuery.

## What Is ECMAScript?

ECMAScript is the standard published by the Ecma International TC39. Available on their website, the [TC39 scope](#) is as follows: “Standardization of the general purpose, cross platform, vendor-neutral programming language ECMAScript. This includes the language syntax, semantics, and libraries and complementary technologies that support the language.”

ECMAScript’s most popular implementation is JavaScript. For all intents and purposes, you can think of ECMAScript as the name for the standards, and JavaScript as the name of the programming language implemented by web browsers.<sup>2</sup>

## **What Is ES6?**

ES6 is the version of the ECMAScript standard published in 2015.

Why are we still talking about it in 2018 and beyond? The ES6 standard included some popular language features that are still being evangelized today: the `let` and `const` statements for variables, default parameter values in function definitions, and arrow functions. This version of JavaScript coincided with a general push to use “modern JavaScript” and so “ES6” is sometimes used as shorthand for “modern JavaScript.”

All current browsers have nearly full support for ES6, with the most popular features definitely supported. You can safely code ES6 JavaScript without the use of a compiler for browser support.

## **What Is ES9?**

ES9 is the version of the ECMAScript standards published in 2018.

As of this writing, it is the latest version of the standard.

## **What Is ESNext?**

ESNext is the term for the future version of ECMAScript to be published sometime in 2019.

## **What Is Ajax?**

*Ajax* (Asynchronous JavaScript and XML) is a way of using JavaScript to query the server after a page has already loaded.

Historically, XML data would be returned and then processed by the browser using more JavaScript. These days, we more typically send back JSON-encoded data or straight up HTML to be incorporated into the app. In this chapter, we will cover executing an Ajax call via the jQuery `ajax()` method and also through the new Heartbeat API for WordPress.

## What Is JSON?

*JSON* (JavaScript Object Notation) is a machine- and human-readable format for transmitting data. It is especially useful when working with JavaScript, since a properly encoded JSON statement can be evaluated by JavaScript with no extra processing. To work with JSON in PHP, we use the `json_encode()` and `json_decode()` functions that have been part of PHP core since version 5.2.

## jQuery and WordPress

jQuery is a popular JavaScript library that makes doing many things with JavaScript a lot easier. One of the things that is easier to do with jQuery is Ajax calls.

WordPress comes installed with its own version of jQuery, which is used in the administrator dashboard for various UI and Ajax-related scripting. Because jQuery is already on your server, including it in the frontend of your WordPress app is a breeze.

## NOTE

As of this writing, WordPress comes bundled with jQuery 1.12.4, whereas the latest jQuery version is 3.3.1. Most WordPress themes and plugins use the version of jQuery bundled with WordPress, and that is what we cover here. The transition to newer versions of jQuery is proving tricky. You can follow the discussion through the [WordPress core trac ticket](#).

The jQuery JavaScript file is located at `/wp-includes/lib/js/jquery.js`. Typically, you would add a link like the following to the `<head>` tag of your website to load jQuery:

```
<script lang="JavaScript" src="/wp-
includes/lib/js/jquery.js" />
```

This works if added to your theme's `header.php` or through the `wp_head` hook; however, the proper way to include a `.js` file in your WordPress site is to use the `wp_enqueue_script()` function. Add the line `wp_enqueue_script('jquery');` to an `init` function called by the inside your main plugin file, like so:

```
function sp_enqueue_scripts() {
    wp_enqueue_script( 'jquery' );
}
add_action( 'init', 'sp_enqueue_scripts' );
```

The first parameter of the `wp_enqueue_script()` function is a label for the JavaScript file to enqueue. WordPress already knows what `jquery` is and where it's located,<sup>3</sup> so that is the only parameter you need to enqueue it.

## Enqueuing Other JavaScript Libraries

To enqueue other JavaScript libraries that WordPress doesn't already know about, pass the full list of parameters. Your main app plugin may include code like the following to load jQuery and any number of required JavaScript libraries. Also, while it is possible to enqueue your scripts on the `init` hook, you should use the

`wp_enqueue_scripts` and `admin_enqueue_scripts` hooks instead. The `wp_enqueue_scripts` hook fires on the frontend just before enqueueing scripts, while the `admin_enqueue_scripts` hook fires in the dashboard just before enqueueing scripts:

```
<?php
//frontend JavaScript
function sp_wp_enqueue_scripts() {
    wp_enqueue_script( 'jquery' );
    wp_enqueue_script(
        'schoolpress-plugin-frontend',
        plugins_url( 'js/frontend.js', __FILE__ ),
        array( 'jquery' ),
        '1.0'
    );
}
add_action( "wp_enqueue_scripts", "sp_wp_enqueue_scripts" );

//admin JavaScript
function sp_admin_enqueue_scripts() {
    wp_enqueue_script(
        'schoolpress-plugin-admin',
        plugins_url( 'js/admin.js', __FILE__ ),
        array( 'jquery' ),
        '1.0'
    );
}
add_action( 'admin_enqueue_scripts',
```

```
'sp_admin_enqueue_scripts' );  
?>
```

Using `wp_enqueue_scripts` and `admin_enqueue_scripts` lets you load different JavaScript files on the frontend and backend of your site. You could add other checks in here to make sure that jQuery is loaded only on certain pages, which could improve load times on those pages that don't need jQuery loaded. Common methods include checking attributes of the global `$post` or checking `$_REQUEST` values used in the administrator like `$_REQUEST['page']` or `$_REQUEST['post_type']`.

Remember, the first parameter of the `wp_enqueue_script()` function is a reference label. The second parameter of the `wp_enqueue_script()` function tells WordPress where the script is located. The `plugins_url()` function is used to figure out the URL relative to the current file `__FILE__`. This works when this code is included in the main plugin file. You would pass `dirname(__FILE__)` as the parameter to this call if the file you are editing is in a subdirectory of the plugin.

The third parameter of the `wp_enqueue_script()` function allows you to state dependencies for your script. By passing `array('jquery')` for our `frontend.js` and `admin.js` scripts, we make sure that jQuery is loaded first.

## Where to Put Your Custom JavaScript

Again, we will run into situations where we need to decide where to put a certain bit of code. Should it go into the theme code or the plugin code? Here are the general rules we use when deciding where a particular bit of JavaScript code will go:

- If the code will be used only once and is generally specific to the page it is used on, it can be coded directly into that page within a `<script>` tag.
- If the JavaScript is used more than once (a function or module) and is related to theme functionality or UI, it is placed in a JavaScript file within the theme (e.g., `/themes/schoolpress/js/schoolpress.js`).
- If the JavaScript is used more than once on your app's administrator screens, it is placed in an `admin.js` file inside your plugin (e.g., `/plugins/schoolpress/js/admin.js`).
- If the JavaScript is used more than once on the frontend of your app, but is not part of the theme UI, it's placed in a `frontend.js` file inside your plugin (e.g., `/plugins/schoolpress/js/frontend.js`).
- If splitting some JavaScript code into its own file, to be loaded on specific pages, will result in a needed increase in performance, that code will be placed in a separate JavaScript file.<sup>4</sup>
- If you are using advanced JavaScript frameworks or a large portion of your app is coded in JavaScript, you will need more structure to your JavaScript files. Follow the conventions of the framework you are using or make intelligent choices about how to split up your JavaScript files.

These rules are specific to how we like to develop and are only a suggestion. Some developers will cringe particularly hard at the thought of adding JavaScript code inside `<script>` tags instead of placing all JavaScript inside `.js` files.<sup>5</sup> The current cycle of web development is moving toward more frontend code and thus more JavaScript. Putting all your JavaScript code in `.js` files is the safe way to develop JavaScript-enabled WordPress apps.

The important thing is that *you* understand how your JavaScript files and code are organized so that working on your site is intuitive.

## Ajax Calls with WordPress and jQuery

There are a number of ways to make Ajax calls with WordPress. Vanilla JavaScript has included the `XMLHttpRequest()` function since 2000, and you can still use that function directly. Current browsers support newer JavaScript features like `promises`, `async` and `await`, and the `fetch()` method, all of which can be used to make sane-looking Ajax queries in vanilla JavaScript. Angular, Vue.js, React, and other JavaScript frameworks have their own methods of doing Ajax-like things. The most popular method of making Ajax calls in WordPress is with jQuery. We use jQuery in the examples that follow.

Ajax calls in WordPress require two components: the JavaScript code on the frontend to kick off the Ajax request, and the PHP code in the backend to process the request and return HTML or JSON-encoded data. Say you want to adjust your signup page to automatically check

if the username entered has already been used. You could warn the person signing up before they hit the submit button and allow them to change the username they picked, saving a bit of grief.

The first thing we need to do is add a quick JavaScript to the head of our pages to define our `ajaxurl`. This is the URL that all Ajax queries will run through. It looks like this:

```
<script type="text/JavaScript">
var ajaxurl = '/wp-admin/admin-ajax.php';
</script>
```

In the WordPress dashboard, this script will be embedded by default. But for frontend Ajax, we'll need to embed it ourselves. Here's the code to define the `ajaxurl` variable for frontend Ajax:

```
function my_wp_head_ajax_url()
{
?>
<script type="text/JavaScript">
var ajaxurl = '<?php echo admin_url("admin-ajax.php");?>';
</script>
<?php
}
add_action('wp_head', 'my_wp_head_ajax_url');
```

Now, the variable `ajaxurl` is available to the rest of the JavaScript on our frontend pages and can be used in our Ajax calls. Here is the JavaScript code to add to the bottom of the registration page to perform the username check:

```
<?php
//our JavaScript for the page
function my_wp_footer_registration_JavaScript()
```

```

{
    //make sure we're on the registration page
    if(empty($_REQUEST['action']) || $_REQUEST['action'] != 'register')
        return;
?>
<script>
    //wait til DOM is loaded
    jQuery(document).ready(function() {
        //var to keep track of our timeout
        var timer_checkUsername;

        //detect when the user_login field is changed
        jQuery('#user_login').bind('keyup change',
function() {
            //use a timer so check is triggered 1 second
            //after they stop typing
            timer_checkUsername = setTimeout(function()
            {checkUsername();}, 1000);
        });
    });

    function checkUsername()
    {
        //make sure we have a username
        var username = jQuery('#user_login').val();
        if(!username)
            return;

        //check the username
        jQuery.ajax({
            url: ajaxurl,type:'GET',timeout:5000,
            dataType: 'html',
            data:
"action=check_username&username="+username,
            error: function(xml){
                //timeout, but no need to scare the
                user
            },
            success: function(response){
                //hide any flag we may have already shown
                jQuery('#username_check').remove();

                //show if the username is good (1) or taken
                (0)
                if(response == 1)
                    jQuery('#user_login').after(

```

```

        '<span id="username_check"
class="okay">Okay</span>'
    );
else
    jQuery('#user_login').after(
        '<span id="username_check"
class="taken">Taken</span>'
    );
}
}

</script>
<?php
}

add_action('wp_footer',
'my_wp_footer_registration_JavaScript');
?>
```

The preceding code is hooked into `wp_footer` so the JavaScript will be added to the end of the HTML output. We first check that `$_REQUEST['action'] == "register"` to make sure we're on the default WordPress registration page.

If you're using a plugin like Paid Memberships Pro that has its own registration page, you'll want to use a check such as `if (!is_page("membership-checkout"))` to figure out which page you are on. You'll also need to make sure that the `#user_login` check in your JavaScript code is updated to use the ID used for the username field on the registration page.

In the code, we use `jQuery(document).ready()` to detect that the DOM is loaded and then use `jQuery('#user_login').bind('keyup change', ...)` to detect when a user has either typed inside the field or otherwise changed it. When this happens, we use `setTimeout()`

to queue up a username check in one second. If the user types again before the timer runs, it is reset to wait one second again. The effect is that one second after the user stops typing or changing the field, the `checkUsername()` function is kicked off.

In the `checkUsername()` function, we have the `jQuery.ajax()` call. Before we do that, though, we check the value from the username field to see whether it's empty. In the `jQuery.ajax()` call, we set the URL to `ajaxurl`, which should have been set via `wp_head` earlier.

We set the type of call to GET. You can also use the POST method. The DELETE and PUT methods are also available, but may not be supported by all browsers. Use the same logic you would when deciding which type to use on a `<form>` you are submitting to decide which method to use in an Ajax call. If you are “getting” data like we are in this example, GET makes sense. If you are submitting data to be saved, you can use the POST method.

We set a timeout of 5,000 (5 seconds) here. After this, the request will be canceled and the defined error action will be kicked off. Set the timeout value based on the reasonable amount of time it might take your server to process this particular request. If you set it too low, you'll prematurely cancel requests. If you set it too high, people will be waiting a long time for requests that may have hung up on the server side.

We set the datatype to `html` here. This tells jQuery to take the output and place it into a string. A datatype of `json` will evaluate the output and place it into a JavaScript object variable. There are a few other datatypes including `xml`, `jsonp`, `script`, and `text`. The jQuery documentation addresses when you would use these and how jQuery processes each datatype.

We set the data to

```
"action=check_username&username='"+username,
```

which will pass our defined action and the username as parameters to the `wp-admin-ajax.php` script and our service-side code.

Then we set a handler in case of errors and in case of success. In case of error, you could alert the user, but since this isn't a critical function, we just go about our business. In case of success, we remove the old `#username_check` element and append an "OK" or "Taken" message after the username field.

#### NOTE

jQuery hosts the full API documentation for the `jQuery.ajax()` on its [website](#).

Now let's see the backend code. Here is the code you would put into `functions.php`, your custom plugin, or a `.php` in your plugin's `/services/` directory to listen for the Ajax request and send back a 1 or 0 if a username is available or not:

```

<?php
//detect Ajax request for check_username
function wp_ajax_check_username() {
    global $wpdb;
    $username = $_REQUEST['username'];

    $taken = username_exists( $username );

    if ( $taken )
        echo "0"; //taken
    else
        echo "1"; //available
}
add_action( 'wp_ajax_check_username',
'wp_ajax_check_username' );
add_action( 'wp_ajax_nopriv_check_username',
'wp_ajax_check_username' );
?>

```

## NOTE

The `username_exists()` function returns the ID of a user if it exists or false if it doesn't exist. The function is documented in a [WordPress Codex page](#).

There are two hooks for defining your Ajax functions in WordPress, and in this example we use both of them:

`wp_ajax_{action}`

Runs for logged-in users

`wp_ajax_nopriv_{action}`

Runs for nonusers

On the registration page, users are by definition not logged in, so we need to use the `wp_ajax_nopriv_` hook. But we may also want to use this check on the add new user screen in the administrator, so we'll also hook into `wp_ajax_` to handle that case.

If you have an Ajax service that will be used only by users, just use the `wp_ajax_` hook. If you need your service available for users and nonusers, you'll need to use both hooks.

Also, notice how the `action` parameter we're looking for (`check_username`) is added to the hook in the action definition. This hook will fire only if `$_REQUEST['action'] == "check_username".`

## Managing Multiple Ajax Requests

When working with Ajax requests, it's important to keep track of them. If not, you can put undue stress on your server and the client's browser, leading to a lockup of their session or the entire site.

For example, in the preceding code, we wait one second after the username field is updated before kicking off the Ajax request to check if the username is available. But once the request goes out, the user might keep on typing, kicking off another Ajax request. If your server isn't able to get back within one second, those requests might start to build up on each other.

Now, our username checker might not have too much potential to get out of hand, but it's possible in a lot of situations. A simple example

would be one in which an Ajax request is kicked off when a user clicks a button. If the user clicks the button 20 times, that could be 20 requests on your server. So keep track of them.

Generally, you want to do one of two things when managing your Ajax requests:

- Keep a user from submitting a request if another request of the same type is still processing.
- Cancel any existing request of the same type if a new request is submitted.

The option you use depends on what the Ajax request is doing.

Generally, if you’re “getting” data, you’ll want to cancel earlier requests and submit the fresher one. If you are “posting” data, you should ignore the new request until the old one is completed.

Depending on your app and the request at hand, there will be many ways to disable or cancel requests. Since the “complete” callback in jQuery’s `ajax()` method is called whether the request is successful or errors out, you can use it to re-enable a button or other element that’s being used to kick off a specific Ajax request:

```
//Option #1: Disabling a button while an Ajax request is
processing
jQuery('#button').click(function() {
    //disable the button
    jQuery(this).attr('disabled', 'disabled');

    //do the ajax request
    jQuery.ajax({
        url: ajaxurl,type:'GET',timeout:5000,
        dataType: 'html',
```

```

        error: function(xml) {
            //error stuff
        },
        success: function(response) {
            //success stuff
        }
        complete: function() {
            //enable the button again
            jQuery('#button').removeAttr('disabled');
        }
    });
});

```

Similarly, here is some code that will cancel an old request when a new one comes in:

```

//Option #2: Cancel an older request when a new one comes in
var ajax_request;
jQuery('#button').click(function() {
    //cancel any existing requests
    if(typeof ajax_request !== 'undefined')
        ajax_request.abort();

    //do the ajax request
    ajax_request = jQuery.ajax({
        url: ajaxurl,type:'GET',timeout:5000,
        dataType: 'html',
        error: function(xml) {
            //error stuff
        },
        success: function(response) {
            //success stuff
        }
    });
});

```

## Heartbeat API

Earlier in this chapter, we built an Ajax call triggered by a form field being updated. Sometimes you will want certain updates to happen on their own periodically as your web app is running. For example, you may want to check for new comments on a discussion forum and automatically pull in fresh comments as they are posted. With JavaScript, you typically do this by polling the backend every few seconds using an Ajax call kicked off by the `setInterval()` function. Alternatively, you can use the WordPress Heartbeat API.

The Heartbeat API has been available since WordPress 3.6 and can be used to facilitate quasi-real-time updates in your app. Every 15 seconds (or less if you change the settings), your app will send a heartbeat request from the client to the server and back. During this round trip, you can do things like autosave app states or load fresh content. In WordPress 3.6, the Heartbeat API is being used for autosaving posts, locking posts, and giving login expiration warnings. In this section, we cover how you can use the Heartbeat API for your app.

Like anything else, the Heartbeat API can seem complicated, but at its heart,<sup>6</sup> it's simply a bunch of data passed back and forth from the client to the server through periodic Ajax calls. Using hooks, you can tap into the data being sent or received to get the information you need to and from the server.

Here is a minimal example demonstrating the Heartbeat API. The only thing this code does is send a message `marco` to the server. If the server sees that message, it sends `polo` back to the client. Both

messages are logged to the JavaScript console, so every 15 seconds, you should see the following in your console:

```
Client: marco  
Server: polo
```

We can break down the use of the Heartbeat API into three segments: initialization, client-side JavaScript, and server-side PHP.

## INITIALIZATION

```
//enqueue heartbeat.js and our JavaScript  
function hbdemo_init()  
{  
    /*  
     * //Add your conditionals here so this runs on the  
     * pages you want, e.g.  
     *     if(is_admin())  
     *         return; //don't run  
     * this in the admin  
     */  
  
    //enqueue the Heartbeat API  
    wp_enqueue_script('heartbeat');  
  
    //load our JavaScript in the footer  
    add_action("wp_footer", "hbdemo_wp_footer");  
}  
add_action('init', 'hbdemo_init');
```

This first function enqueues the *heartbeat.js* file and sets up an action to put our JavaScript code in the footer via the `wp_footer` hook. If you want this heartbeat code to run only on certain pages (very likely), you would put your checks here.

## CLIENT-SIDE JAVASCRIPT

```

<?php
//our JavaScript to send/process from the client side
function hbdemo_wp_footer()
{
?
<script>
    jQuery(document).ready(function() {
        //hook into heartbeat-send: client will send the
message
        // 'marco' in the 'client' var inside the data array
        jQuery(document).on('heartbeat-send', function(e,
data) {
            console.log('Client: marco');

            //need some data to kick off Ajax call
            data['client'] = 'marco';
        });

        //hook into heartbeat-tick: client looks for a
'server'
        //var in the data array and logs it to console
        jQuery(document).on('heartbeat-tick', function(e,
data) {
            if(data['server'])
                console.log('Server: ' +
data['server']);
        });

        //hook into heartbeat-error to log errors
        jQuery(document).on('heartbeat-error',
            function(e, jqXHR, textStatus, error) {
                console.log('BEGIN ERROR');
                console.log(textStatus);
                console.log(error);
                console.log('END ERROR');
            });
        });
    </script>
?
}
?>

```

This second function dumps our JavaScript into the footer. In the JavaScript code, we use `jQuery(document).ready()` to run

our code after the DOM has loaded. Then, we hook into three JavaScript events triggered by the Heartbeat API:

1. The heartbeat-send event is fired directly before the heartbeat sends data back to the server. To send your data, add a value to the data array passed through the event.
2. The heartbeat-tick event is fired when the server replies. To see what data the server has sent, look for it in the data array that is passed through the event.
3. The heartbeat-error event is fired if there is an error in the `jQuery.ajax()` call used to send the data to the server. You can include code here for debugging or degrade nicely if Ajax doesn't seem to be working in your production environment.

## SERVER-SIDE PHP

```
//processing the message on the server
function hbdemo_heartbeat_received($response, $data)
{
    if ($data['client'] == 'marco')
        $response['server'] = 'polo';

    return $response;
}
add_filter('heartbeat_received',
'hbdemo_heartbeat_received', 10, 2);
```

This third PHP function in the previous example runs on the `heartbeat_received` hook and processes the data from the client. We can add data to go back to the client by updating the `response` variable.

Now let's try a more realistic example. SchoolPress has a section of the assignments page showing how many assignments have been submitted and how many are left. Let's use the Heartbeat API to update this number if new assignments have been posted.

In our template, the assignment count will be displayed something like this:

```
?>
<div>
    Submitted:
    <span id="assignment_count">
        <?php echo count($assignment->submissions); ?>
    </span>
    /
    <?php echo count($course->students); ?>
</div>
<?php
```

## INITIALIZATION

```
function sp_init_assignments_heartbeat()
{
    //Ignore if we're not on an assignment page.
    if(strpos($_SERVER['REQUEST_URI'], "/assignment/")
== false)
        return;

    //enqueue the Heartbeat API
    wp_enqueue_script('heartbeat');

    //load our JavaScript in the footer
    add_action("wp_footer",
    "sp_wp_footer_assignments_heartbeat");
}

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

Things are pretty similar to our minimal example so far. We're just making sure we don't run this code on nonassignment pages by checking for */assignment/* in the URI:

## CLIENT-SIDE JAVASCRIPT

```
<?php
function sp_wp_footer_assignments_heartbeat()
{
    global $post;      //post for current assignment
?>
<script>
jQuery(document).ready(function() {
    //heartbeat-send
    jQuery(document).on('heartbeat-send', function(e, data) {
        //make sure we have an array for SchoolPress data
        if(!data['schoolpress'])
            data['schoolpress'] = new Array();

        //send to server the post_id of this assignment and
        current count
        data['schoolpress']['assignment_post_id'] = '<?php echo
$post->ID;?>';
        data['schoolpress']['assignment_count'] =
jQuery('#assignment-count').val();
    });

    //heartbeat-tick
    jQuery(document).on('heartbeat-tick', function(e, data) {
        //update assignment count
        if(data['schoolpress']['assignment_count'])
            jQuery('#assignment-count').val(data['schoolpress']
['assignment_count']);
    });
});
</script>
<?php
}
```

Notice that we're storing our data in a `schoolpress` array within the data array. We'll store all Heartbeat-related data in this array as a kind of namespacing to make sure our variable names don't conflict with any other plugins that might be using the Heartbeat API. Each time the heartbeat sends data to the server, we send along the assignment's post ID and the current count value.

### NOTE

It's important that you send *something* to the server through the heartbeat. If there is no data to send, the heartbeat won't bother polling the server at all.

## SERVER-SIDE PHP

```
//processing the message on the server
function sp_heartbeat_received_assignment_count($response,
$data)
{
    //check for assignment post id
    if(!empty($data['schoolpress']
['assignment_post_id']))
    {
        $assignment = new Assignment(
            $data['schoolpress']
['assignment_post_id']
        );
        $response['schoolpress']['assignment_count']
= count(
            $assignment->submissions
        );
    }

    return $response;
}
```

```
add_filter('heartbeat_received',
    'sp_heartbeat_received_assignment_count', 10, 2);
```

Here we check for the assignment\_post\_id value passed from the client. If it's found, we load up the assignment and return the count of submissions as assignment\_count, which our frontend JavaScript will be looking for.

This code could be updated to detect changes in the assignment count (by comparing the number sent from the client to the number found server-side); in those cases, it would pass back a message notifying the teacher to refresh to view the new submissions. Or we could send some data about the new submissions themselves and push them into the list on the page.

Finally, if you want to speed up or slow down the heartbeat, you can override the settings using the following code:

```
function sp_heartbeat_settings($settings = array())
{
    $settings['interval'] = 20; //20 seconds vs. 15 second default
    return $settings;
}
add_filter('heartbeat_settings', 'sp_heartbeat_settings');
```

Note that as of this writing, the API will only let you use a value between 15 and 60 seconds. Anything faster or slower will be set to 15 or 60 seconds, respectively. This limitation is actually a good idea for the Heartbeat API, since at any given time, multiple plugins and processes may be using that same heartbeat. If you need a certain poll to occur faster than once every 15 seconds, you should probably set it

up as a separate Ajax call using your own `setInterval` or `setTimeout` calls in JavaScript.

You can think of the Heartbeat API as a more casual way of doing polling between client and server. If you need something more hardcore (and polling your server every second is pretty hardcore), you should roll your own Heartbeat-like system.

## **WordPress Limitations with Asynchronous Processing**

Most WordPress applications execute PHP scripts through an Apache or NGINX server. When optimized, you can serve a lot of small, simultaneous connections on these setups, which is perfect for asynchronous JavaScript applications. However, the servers themselves, and perhaps more important, the general overhead of loading WordPress on server-side calls, means that a WordPress service running on Apache or NGINX will never be as fast as a smaller JavaScript service running on something like Node.js, which was built specifically to handle asynchronous JavaScript calls.

That said, you can still get a lot done with WordPress and the architecture behind it. Our suggestion is always to build it the obvious way first and selectively pull out parts of your application for scaling later when performance becomes an issue.

Does your app have a user base consisting solely of the 30 people in your company? Then you'll probably be fine using WordPress for your real-time JavaScript coding.

Do you plan to have thousands of users, with dozens of simultaneous connections? You'll need some beefy hardware, but you'll also probably be fine keeping everything in WordPress.

Do you plan to have millions of users, with tens of thousands of simultaneous connections? If so, you need some top-notch engineers, so hopefully you have the money for them. In any case, you'll either be pushing WordPress to its limits or using other platforms to serve your real-time interactions.

We explore these kinds of scaling questions in [Chapter 14](#).

## JavaScript Frameworks

One thing limiting a move to JavaScript is that all of our handy functions and data structures are native to PHP. As more development is done on the JavaScript side of the WordPress platform, there is a greater need for some kind of framework to help organize the JavaScript development.

### Backbone.js

Backbone.js is a framework for JavaScript consisting of models, views, and collections of models. This setup is very similar to the MVC frameworks used for server-side PHP development. In traditional MVC frameworks, the *C* stands for “controller.” With Backbone.js, the controlling of an app is handled within the views and outside the JavaScript framework itself.

Backbone.js was added to WordPress core in version 3.5 and has been used extensively in the Media Library and Theme Customizer updates. The WordPress core team and most WordPress developers have moved on or skipped over Backbone.js in favor of more recent JavaScript libraries and frameworks.

It is still worth understanding how Backbone.js could be used for JavaScript development with WordPress. If you are using Backbone.js to render the frontend of your app, the main intersection point with WordPress will be when your collections and models are saved to the database through the backend.

Imagine an interface on the SchoolPress site for adding student groups to an assignment. There may be an input box for naming the group and a button labeled Add Group to add the group. Using the traditional Ajax technique outlined in this chapter, the turn of events would look like this:

1. The user enters a new group name.
2. The user clicks the Add Group button.
3. The group name is sent to the server via Ajax.
4. The server (WordPress) processes the name, adds the new group, and returns some data.
5. The client uses JavaScript to parse the response and update the list of groups on the frontend.

With a Backbone.js app, you mirror the list of groups more thoroughly in the model and collection you would set up in

JavaScript. You could use a similar workflow as the typical Ajax app, but a more appropriate workflow for a Backbone.js app would be:

1. The user enters a new group name.
2. The user clicks the Add Group button.
3. The group name is used to create a new instance of the group model and added to the group collection in Backbone.js.
4. The collection will be coded to update the server (WordPress) through Ajax whenever the collection changes.
5. A representation of the current collection of groups is sent to the server.
6. WordPress updates the internal representation of the collection in the database to match what was sent.

So instead of first updating things in the backend and the backend telling the frontend what to look like, with Backbone.js things are first updated on the frontend, and the frontend tells the backend how to save the data.

You can find an example of some SchoolPress functionality coded both with the traditional Ajax technique and then using Backbone.js at <http://bwawwp.com/backbonejs-example/>.

Here are some resources to learn more about Backbone.js, and how to use it with WordPress:

- [Official Backbone.js site](#)
- [“Backbone.js and WordPress Resources” by Peter R. Knight](#)

## React

React is a JavaScript library for building interactive UIs. Unlike some of the other libraries covered here, React is not a full application framework. Instead, it focuses on the Views part of a typical MVC framework.

In React, you define views for each state of your application. React then monitors the state of your app and updates your UI as the data changes. For example, a counter showing the total number of users on your site (properly built in React) would automatically update if you added a new user on the same screen or even if another user signed up from their own device.

React is component-based. A component could be something as simple as a single text field or button. Subcomponents can be used together to create more complex components. You can see many of the React components used in WordPress in [the Gutenberg GitHub repository](#). Individual WordPress features, like the editor itself, will have [their own library of supporting components](#).

You can also use React to create native mobile apps using the appropriately named React Native library. These are not write-once-run-anywhere web apps; they're apps that run natively on mobile devices. The same React code running your web app can't be used for a mobile app. However, you can use the same JavaScript programming language and React concepts to build UIs for web or mobile depending on your needs.

The new block editor UI and components are built with React. We will cover how to build and extend blocks in [Chapter 11](#), including the recommended JavaScript development workflow, which reinforces some of the concepts touched on in this chapter. Before that, we encourage you to learn about the WordPress REST API, which we examine in [Chapter 9](#).

---

- 1 Other implementations of ECMAScript include JScript and ActionScript.
- 2 We don't know why ECMA is capitalized in ECMAScript but not in Ecma International, but that is how things are spelled out on the official sites.
- 3 To see how all of the bundled JavaScript files are registered in core WordPress, view the `wp_default_scripts()` function in */wp-includes/script-loader.php*.
- 4 Standalone JavaScript files can be cached or served through a content delivery network (CDN). JavaScript embedded in dynamic PHP files cannot be cached as easily.
- 5 Current-day Jason is one such developer. If he could go back in time, he would tell 2012 Jason to make sure all the JavaScript in the Paid Memberships Pro plugin is stored in its own `.js` files. This is particularly important for plugins and themes that will be distributed on sites outside your control. Having your JavaScript in its own files makes it easier for sites to optimize by merging JavaScript into one file or serving it in the footer instead of the header of your site. Having JavaScript in its own file also avoids issues where other JavaScript embedded into a page breaks, causing your JavaScript to never load. Even so, we still think it's OK to use JavaScript within your `.php` files in certain cases where you have full control over your app and are trying to keep things simple.
- 6 Hehe.

# Chapter 10. WordPress REST API

---

In version 4.4 of WordPress, the REST API was introduced to the project, making WordPress more extensible than ever. The WordPress REST API makes it possible for developers to push and pull data to and from WordPress from outside WordPress. This allows you to create all kinds of cool and useful software applications that may not necessarily use the traditional WordPress UI but are powered by or integrated with WordPress. It allows you to communicate with other software applications regardless of the application programming language. Before we dig into the specifics of using the WordPress REST API, let's quickly describe what we mean generally by "REST API."

## NOTE

This chapter covers much of the same content as WordPress's [detailed handbook on using the REST API](#). In some parts, we go into more detail or use examples that make more sense in the context of this book. Note that some things we cover briefly here are covered in more detail in the WordPress handbook.

## What Is a REST API?

You might already know what a REST API is and even be surprised that one wasn't built in to WordPress sooner. Let's try to break it down Barney style—you know, that annoying big purple dinosaur that kids were once obsessed with, and that made parents want to gouge out their eyes and ears.

### NOTE

Some refer to the WordPress REST API as the JSON REST API. We will leave off the “WordPress” in this chapter and just write REST API.

Let's cover some of the basics.

## API

An *application programming interface* (API) is a collection of functions and tools built in to a software application used by developers to interact with that software application either internally or externally, usually via the internet between web applications. Depending on how the API was set up, you could use it to push and pull content and data into or out of another application, thus effectively integrating two or more applications.

The REST API allows you to interact with the WordPress database from outside of your WordPress website.

## REST

REST (for *representational state transfer*) is an architectural style for defining HTTP-based communications between web applications. There are guiding principles for building a REST API, and an interface must meet each of the following constraints for it to be *RESTful*.

- Client-server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand

## JSON

JSON, as we've mentioned previously, is an open standard text format for sending readable text in data objects like arrays and name-value pairs. The response from a REST API HTTP Request should be in JSON format. You should be able to parse the JSON in an HTTP Response and use that data within your application.

## HTTP

HTTP is a standard for requests and responses to and from a client and a server. Every time you visit a website in your web browser you make an HTTP Request and receive the HTTP Response or the web page your browser loaded. Pushing and pulling data to and from WordPress via the REST API happens over the same protocol, but instead of returning the web page, the response is the raw data in

JSON format. With a default WordPress installation, you can make HTTP requests to access public content such as posts and pages, in JSON format, if you type the appropriate URL or API *route* directly into your browser.

There are four main types of HTTP request methods for working with RESTful APIs like WordPress.

- POST
- GET
- PUT
- DELETE

People may also refer to these as *CRUD* (Create, Read, Update, and Delete) actions. Whatever you call these self-explanatory verbs, they are used via HTTP requests to perform their particular actions and then return the results of those actions in the HTTP Response.

Try appending */wp-json/wp/v2/posts* to the end of your WordPress website domain, and you should be looking at your posts in JSON format. This is an example of an HTTP GET request method, and its response. The process of sending an HTTP request and retrieving an HTTP response is known as passing *HTTP messages*.

There are three parts to every HTTP message: the request, the headers, and the message body.

## REQUEST

The *request* (or *request line*) is sent from the client to the server and includes the request method, a request URL, and the HTTP version:

*Request method*

GET, POST, PUT, DELETE

*Request URL*

The URL of the API route

*HTTP version*

The version of HTTP

These combine as seen in the following:

```
GET URL HTTP/1.1
```

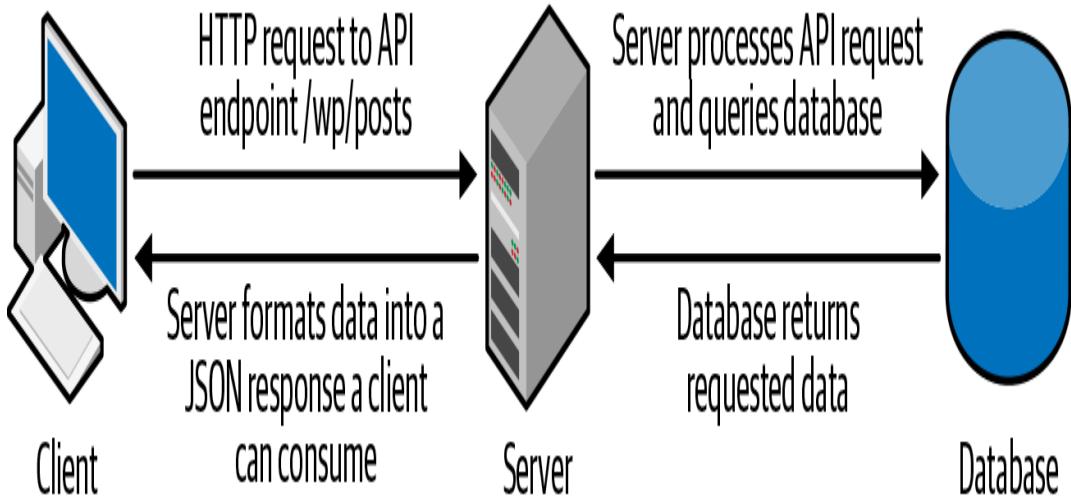
The *status line* is returned from the request in the response, and includes the HTTP version and a status code:

```
HTTP/1.1 200 OK
```

**NOTE**

If you Google “HTTP status codes and what they mean” you can research all available HTTP status codes that can be returned in an HTTP response. To *really* understand HTTP, check out [\*HTTP: The Definitive Guide\*](#) (O’Reilly), by Brian Totty, David Gourley, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy.

Figure 10-1 shows a request going to a server and a response going to the client.



*Figure 10-1. The request line*

## HEADERS

The HTTP header sent in an HTTP message includes metadata that can be passed into the HTTP request and received from the HTTP response. There are various standard request and response fields that can be used in the header, including your own custom header fields. The headers are generally used to pass information (like authentication and permissions) back and forth from the client to the server. [Wikipedia](#) has a full list of standard request and response header fields.

In Chrome and other WebKit browsers, you can view the headers for the current page or any API calls made by opening the Developer Tools and then clicking the Network tab. On the list on the left, click the document or Ajax call and then click the Headers subtab. Here are the headers for that Wikipedia page to which we just linked, as shown in [Figure 10-2](#).

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

# List of HTTP header fields

From Wikipedia, the free encyclopedia

**HTTP header fields** are components of the header section of request and response messages in the Hypertext Transfer Protocol (HTTP). They define the operating

**HTTP**  
Persistence • Compression • HTTPS • OAuth

Elements Console Sources Network Performance Memory Application Security Audits □ 1 X

View: Group by frame Preserve log Disable cache Offline No throttling ▾

Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Name Headers Preview Response Cookies Timing

List\_of\_HTTP\_header\_fields General

Request URL: [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)

Request Method: GET

Status Code: 304

Remote Address: 208.80.154.224:443

Referrer Policy: no-referrer-when-downgrade

Response Headers

age: 73549  
backend-timing: D=127568 t=1557248118759080  
cache-control: private, s-maxage=0, max-age=0, must-revalidate  
content-encoding: gzip  
content-language: en  
content-type: text/html; charset=UTF-8  
date: Wed, 08 May 2019 13:21:08 GMT  
last-modified: Mon, 06 May 2019 17:42:10 GMT  
p3p: CP="This is not a P3P policy! See <https://en.wikipedia.org/wiki/Special:CentralAutoLogIn/P3P> for more info."  
server: mw1270.eqiad.wmnet  
server-timing: cache;desc="hit-front"  
status: 304

1 / 26 requests | 718 B / 2.3 KB trans...

Console What's New X

Highlights from the Chrome 74 update

*Figure 10-2. Using Chrome Developer Tools to view the headers for a web page*

### NOTE

The Status Code header is one you will be looking at a lot. A typical web page that loads with no errors shows status 200 “OK.” The Wikipedia page in [Figure 10-2](#) shows status 304 “Not Modified.” Wikipedia is using a browser caching scheme that won’t waste resources transmitting a page you refreshed unless the page has been modified. In [Chapter 14](#) we discuss how to set up browser caching with the W3 Total Cache plugin.

## MESSAGE BODY

The message body isn’t always required but can be passed with a request or response. The REST API typically returns JSON in the message body containing the WordPress data you queried in your HTTP request. Depending on your request method you may not receive a message body in your response.

## Why Use the WordPress REST API?

Well, with the REST API you can do just about anything you might need to do from within the traditional WordPress dashboard, but from anywhere, and it’s also a standard and secure way to do so. Following are examples of what you might use the REST API for:

### *Allowing public access to WordPress data*

The API is on by default in the latest versions of WordPress, allowing anyone to easily grab any of your public content such as posts, pages, media, and comments. Although not public by

default, you can make any CPT and any of its data publicly available via the REST API as well. You can also create custom endpoints that are open to the public to retrieve whatever data you specify.

#### *Allowing CRUD access to manage WordPress data*

Precisely the opposite of what we just discussed. If you want a third-party app to be able to manage WordPress data, you need to have the authorization to do so. The REST API allows you to authenticate a few different ways so you can create, read, update, and delete on a WordPress user's behalf. Once you've properly authenticated, you can manage all your WordPress data from outside WordPress. We cover the current authentication methods later in this chapter.

#### *Exchanging data between WordPress websites*

This is a pretty typical use case; imagine the possibilities of easily exchanging data between two or more WordPress websites.

#### *Headless WordPress websites*

The term *headless* just means that you aren't using the WordPress frontend at all but instead are using the backend. You may want to set up a headless WordPress installation strictly to store data you will use in another application via the REST API.

#### *WordPress-powered mobile apps*

At AppPresser we do this all day every day. Utilizing the REST API, you can build mobile apps that use WordPress as the data source. If you are building a standalone app and don't need a website, you could use a headless WordPress installation to power your app. If you have an existing website, you can build a mobile app to complement it and share its data. We cover building mobile apps with WordPress in more detail in [Chapter 16](#).

### *Asynchronous data synchs*

Developers commonly run into issues when they're building new WordPress websites from snapshots of Production WordPress databases (usually done to preserve all original content, and also to give developers real data to work with while building the site). Content like posts, comments, and users could be created on the Production website while the new site is still being developed. This can be a real pain when it's time to launch the new website. Depending on the project, you might want to do a content freeze on the Production website and import any new data into the Development website, which then will become the Production website. Or maybe it makes sense to move any new data on the Development database, like settings or new post-type posts and meta, into the Production website. If you've done many new WordPress site launches from existing WordPress sites, you probably know where we're going. By the time you're ready to launch the new website, you might have data that you need in both the Production and Development websites. Now you are stuck writing custom scripts that cherry-pick the data that you need from either environment. Hmm, maybe use the REST API to update the Development website with any new or updated data in real time as it happens on the Production website. A solution like this could save you lots of time manually synching your data prior to launching a new website.

### *Virtual reality and gaming*

Imagine immersive VR environments with custom content delivered via the REST API. For example, say you're building a race car game and you want the billboards on the road to display custom content.

### *Alexa and Google Home*

Alexa, make me a sandwich.<sup>1</sup> We're sure it's possible using the REST API. ;)

## Using the WordPress REST API V2

Now that you understand why you might want to use the REST API and more about some of the technologies behind APIs in general, let's go into using the REST API specifically.

### Discovery

In current versions of WordPress, the REST API is enabled by default, but you will still want check to make sure that it is actually enabled and also check what content you will have access to. The easiest way to see what you are working with is to add `/wp-json/` or `?rest_route=/2` on the end of any WordPress domain in a web browser or Postman—for example, `localhost/wordpress/wp-json/`.

You should be looking at a JSON response of the schema returned by all the API endpoints registered on the site.

### Authentication

Authentication is important if you want to use the REST API for more than just pulling public content from a WordPress website. The REST API provides options for authentication that can be utilized depending on what you are building.

#### COOKIE AUTHENTICATION

Cookie authentication is the default and, as of this writing, the only authentication method built in to core WordPress. You can also use other authentication methods via third-party plugins or build your own if you choose or need to.

For cookie-based authentication, a user just needs to be logged in to WordPress. When any API requests are made, they are governed by what actions that logged-in user can perform.

The REST API also uses nonces to address cross-site request forgery (CSRF). Because of this, someone trying to be malicious can't just use your cookie to do whatever they want; they would also need a valid nonce. The recommended method for using the REST API for custom themes and plugins is to use the built-in *JavaScript API* in which the nonces are automatically created for you. You can also make manual requests, but you would need to pass the nonce with each request that you make.

Cookie authentication makes it super easy to build cool features and functionality into the frontend of your website via a theme or plugin so your users can interact with WordPress data without going to the backend.

As an example, this is how the built-in JavaScript client creates the nonce:

```
wp_localize_script( 'wp-api', 'apiSettings', array(
    'root' => esc_url_raw( rest_url() ),
    'nonce' => wp_create_nonce( 'wp_rest' )
) );
```

This code places into the HTML header a JavaScript object that can be used in Ajax calls to an API endpoint. Pass the nonce with your Ajax call and it will be used to authenticate your API calls. The nonce will also protect your site from CSRF attacks. We cover nonces and related security concerns in detail in [Chapter 8](#).

Following is an example Ajax call using jQuery:

```
$ .ajax( {
    url: apiSettings.root + 'wp/v2/posts/1',
    method: 'POST',
    beforeSend: function ( xhr ) {
        xhr.setRequestHeader( 'X-WP-Nonce',
            apiSettings.nonce );
    },
    data: {
        'title' : 'Hello World'
    }
} ).done( function ( response ) {
    console.log( response );
} );
```

The code here updates the title of a post with ID 1. The nonce is passed in the request header X-WP-Nonce. If you create custom endpoints, you will not need to validate the nonce; the API will validate it for you.

## BASIC AUTHENTICATION

Basic Authentication is a quick-and-dirty solution for developing apps for external clients that integrate with WordPress data. First things first, though; you don't want to use this method of authentication in a production environment or any WordPress website

you wouldn't want to get hacked (which should be all of them). Basic Authentication should only be used for development purposes, because it requires a base64-encoded username and password be sent through the header of every REST API request. Base64-encoded strings can be intercepted and decoded, so you see the problem this could cause you.

To quickly be able to test the REST API with an external-to-WordPress application you are building, you should check out the [WP-API Basic-Auth plugin](#).

If your website is set up to serve page requests *only* over HTTPS and the API end point you are accessing is also set up to serve requests only over HTTPS, this makes Basic Authentication more secure. The same encrypted string will still be sent with every request, but if your password is sufficiently strong (see [Chapter 8](#)), it should be uncrackable.

You can also alter the Basic Authentication code to only work for specific endpoints. This means you can allow Basic Authentication for one of your endpoints without enabling it for your entire WordPress site. The WP SSO plugin we introduce later in this chapter uses Basic Authentication this way. Even so, you should consider other authentication methods before using Basic Authentication for your app's APIs.

Here is an example of a REST API request using Basic Authentication:

```
$.ajax( {
    url: 'localhost/wp-json/wp/v2/posts/1',
    method: 'POST',
    beforeSend: function ( xhr ) {
        xhr.setRequestHeader( 'Authorization',
            'Basic ' + btoa( username +
        ':' + password ) );
    },
    data: {
        'title' : 'Hello World'
    }
} ).done( function ( response ) {
    console.log( response );
} );
```

## JSON WEB TOKENS

JSON Web Tokens (JWT) is a JSON-based open standard for creating access tokens that can be used to authenticate API access to a web service. This is a secure way to share information between a web service and a client-side application. A JWT is just a JSON object made up of a header, a payload, and a signature. The JWT creation process starts when a user signs in to an authentication service (think wp-admin login) and a JWT is created for them to use in a third-party application. When the web service (think REST API) receives a JWT from a third-party application, it checks that it's valid and, if authenticated, the client can use the API to perform any CRUD actions that token's corresponding user would have access to make.

In our opinion, probably the quickest and easiest way to start using the REST API securely is to use the [JWT Authentication for WP REST API Plugin](#). This plugin extends the REST API to use JWTs as an API authentication method. Configuring the plugin is pretty

straightforward; try this if you're looking for the quickest way to securely use the REST API.

## OAUTHP AUTHENTICATION

Open Authorization (OAuth), released in 2007, is an open standard for token-based authentication and authorization. OAuth is the preferred method for developing apps for external clients that integrate with WordPress data. OAuth basically acts as a middle man between the user and WordPress. It allows a user to authenticate without ever exposing their password in any requests. An *OAuth Flow* is the process of obtaining a token that is then used to authorize specific account information for a user to be shared with an application.

Currently there are two versions of OAuth, 1.0 and 2.0. The major difference between the two versions is that 2.0 requires that API requests are authorized via HTTPS or SSL/Transport Layer Security (TLS). There are also two OAuth WordPress plugins that the REST API Team has developed to work with each version of OAuth.

<https://github.com/WP-API/OAuth1>

This plugin actually uses OAuth 1.0a because it does not require SSL for any endpoints, while OAuth 1.0 does require SSL for some. Because WordPress does not require SSL or that your website be HTTPS, this is currently the more developed and popular plugin to use.

<https://github.com/WP-API/OAuth2>

This plugin may also be an option for you, but please note that it is still an early beta version.

You could always build your own from scratch or leverage an existing library, but why on earth would you want to spend all that time reinventing the wheel? Both of these OAuth plugins use what is referred to as *three-legged authentication*, where each leg is a separate role that is involved with the authentication process:

### *Client*

The third-party application you want to communicate with WordPress.

### *Server*

The WordPress installation that the third-party application will make API requests on.

### *Resource owner*

The end user that has a login to the WordPress install and is using the third-party application.

OAuth uses token credentials that are issued by the *server*, when the resource owner authenticates using their credentials. These tokens are used by the *client* to gain access to the server. These token credentials can be revoked by the server at any time by the resource owner, but will also eventually expire, which will require the resource owner to reauthenticate.

What follows is a detailed outline of the OAuth flow:

1. The client sends a signed request to the server to obtain a request token, also known as temporary credentials. This request is sent to the Temporary Credentials endpoint URI, and it contains the following:

- `oauth_consumer_key`: provided by the server
- `oauth_timestamp`
- `oauth_nonce`
- `oauth_signature`
- `oauth_signature_method`
- `oath_callback`
- `oauth_version` (optional)

2. The server verifies the request, and if it's valid, grants a request token that contains the following:

- `oauth_token`
- `oauth_token_secret`
- `oauth_callback_confirmed`

3. The client then sends the resource owner (the user) to the server to authorize the request. This is done by constructing a request URI by adding `oauth_token` (obtained in the previous step) to the Resource Owner Authorization endpoint URI.

4. The resource owner (the user) authorizes at the server by providing credentials.

5. If the `oauth_callback` URI was provided in the first step, the server redirects the client to that URI and appends the following as query strings:

- `oauth_token` obtained in the second step
- `oauth_verifier` used to ensure that the resource owner who granted access is the same

returned to the client

6. If the `oauth_callback` URI was not provided in the first step, then the server displays the value of the `oauth_verifier` so that the resource owner can inform the client manually.
7. After receiving `oauth_verifier`, the client requests the server for token credentials by sending a request to the Token Request endpoint URI. This request contains the following:
  - `oauth_token` obtained in the second step
  - `oauth_verifier` obtained in the previous step
  - `oauth_consumer_key` provided by the resource provider (the server), before starting the OAuth handshake
  - `oauth_signature`
  - `oauth_signature_method`
  - `oauth_nonce`
  - `oauth_version`
8. The server verifies the request and grants the following, known as token credentials:
  - `oauth_token`
  - `oauth_token_secret`
9. The client then uses the provided token credentials to access whatever data it needs on the server.

If you are new to OAuth, you may be looking at the process here and thinking that setting up OAuth will take a bunch of work. Because we are using a WordPress plugin developed by the REST API team, this process is streamlined into a few easy steps. So don't worry about fully understanding the entire OAuth flow process. If you are interested in understanding the entire process in detail, go ahead and dissect the plugin. Ah, the beauty of open source!

You can download this OAuth1 plugin directly from the [WordPress plugin repository](#). Let's get this plugin set up on your Dev site. Go ahead and install, then activate, this plugin like you would any other WordPress plugin.

## Routes and Endpoints

In [Chapter 9](#), we learned how to use the `/wp-admin/admin-ajax.php` URL to access our Ajax services. When using the REST API, each API method has its own URL including a *route* and *endpoint*.

### WHAT IS A ROUTE?

Routes point to a resource such as `/wp-json/wp/v2/posts`, this is the http address for the “posts” route. You can use endpoints to perform CRUD operations if they exist for a route.

### WHAT IS AN ENDPOINT?

Endpoints call or trigger a particular method or function when you request a particular route. Each route can have multiple different endpoints.

An example of requesting a route using different endpoints:

*GET /wp/post/1*

This route could have a method or function `get_post()` that returns the requested post.

*POST /wp/posts/1*

This route could have a method or function `update_post()` that updates the requested post.

We visit the same route */wp/post/1*, but the CRUD operation determines what method or function to run and that is the endpoint.

## WHAT IS A NAMESPACE?

A namespace is a “prefix” to your API routes that identify and protect the endpoints from collisions.

The core namespace is `wp`. This namespace is reserved and should not be used in creating custom endpoints. If another plugin or theme registered the same route after you registered a route, it would make the previously registered route void.

## Requests

Internally, when you request an API endpoint the API utilizes `WP_REST_Request`, a core class used to implement a REST request object.

This request object contains many useful methods to process the request. Using the method `get_posts()` seen in the example that

follows the list of parameters, you might visit `/wp/posts?hello=world`. The `get_params()` method would list the `hello` parameter in its array with the value `world`.

The following methods are available:

`add_header`

Appends a header value for the given header.

`canonicalize_header_name`

Ensures header names are in the standard format.

`from_url`

Retrieves a `WP_REST_Request` object from a full URL.

`get_attributes`

Retrieves the attributes for the request.

`get_body`

Retrieves the request body content.

`get_body_params`

Retrieves parameters from the body.

`get_content_type`

Retrieves the content-type of the request.

`get_default_params`

Retrieves the default parameters.

`get_file_params`

Retrieves multipart file parameters from the body.

*get\_header*

Retrieves the given header from the request.

*get\_header\_as\_array*

Retrieves header values from the request.

*get\_headers*

Retrieves all headers from the request.

*get\_json\_params*

Retrieves the parameters from a JSON-formatted body.

*get\_method*

Retrieves the HTTP method for the request.

*get\_param*

Retrieves a parameter from the request.

*get\_parameter\_order*

Retrieves the parameter priority order.

*get\_params*

Retrieves merged parameters from the request.

*get\_query\_params*

Retrieves parameters from the query string.

*get\_route*

Retrieves the route that matched the request.

*get\_url\_params*

Retrieves parameters from the route itself.

*has\_valid\_params*

Checks whether this request is valid according to its attributes.

*offsetExists*

Checks whether a parameter is set.

*offsetGet*

Retrieves a parameter from the request.

*offsetSet*

Sets a parameter on the request.

*offsetUnset*

Removes a parameter from the request.

*parse\_body\_params*

Parses the request body parameters.

*parse\_json\_params*

Parses the JSON parameters.

*remove\_header*

Removes all values for a header.

*sanitize\_params*

Sanitizes (where possible) the params on the request.

*set\_attributes*

Sets the attributes for the request.

*set\_body*

Sets body content.

*set\_body\_params*

Sets parameters from the body.

*set\_default\_params*

Sets default parameters.

*set\_file\_params*

Sets multipart file parameters from the body.

*set\_header*

Sets the header on request.

*set\_headers*

Sets headers on the request.

*set\_method*

Sets HTTP method for the request.

*set\_param*

Sets a parameter on the request.

*set\_query\_params*

Sets parameters from the query string.

*set\_route*

Sets the route that matched the request.

*set\_url\_params*

Sets parameters from the route.

## Responses

The sibling of `WP_REST_Request` is `WP_REST_Response`, a core class used to implement a REST response object.

This class is used to prepare your response data, HTTP status code, and any response headers:

`add_link`

Adds a link to the response.

`add_links`

Adds multiple links to the response.

`as_error`

Retrieves a `WP_Error` object from the response.

`get_curies`

Retrieves the CURIEs (compact URIs) used for relations.

`get_links`

Retrieves links for the response.

`get_matched_handler`

Retrieves the handler that was used to generate the response.

`get_matched_route`

Retrieves the route that was used.

`is_error`

Checks whether the response is an error; that is, a response code  $\geq 400$ .

`link_header`

Sets a single link header.

`remove_link`

Removes a link from the response.

`set_matched_handler`

Sets the handler that was responsible for generating the response.

`set_matched_route`

Sets the route (regular expression for path) that caused the response.

## Adding Your Own Routes and Endpoints

Using the built-in REST API routes and endpoints is powerful, but things get really fun when you build your own routes and endpoints. New endpoints can be used to expose CPTs and other custom data your application uses to third-party services or to JavaScript your application is running in the frontend.

During the writing of this book, we built a plugin called [WP Single Sign-On](#) to demonstrate some features of the REST API, including adding your own. *Single Sign-On* (SSO) means allowing multiple separate sites to be accessed using the same login credentials. There are many methods of doing SSO in general, and other plugins and third-party services to do it with WordPress in particular. The solution we came up with was meant to be simple and straightforward, making it easier to teach with and use for reference. The same plugin shows how to create a custom REST API route and how to use a custom REST API route.

```
register_rest_route( $namespace, $route, $args,  
$override );
```

The function to register new REST API routes is

`register_rest_route()`. This function must be called during the `rest_api_init` action hook.

*\$namespace*

A URL segment that comes between the core prefix<sup>3</sup> and the route you are adding. This string should be unique to your plugin or app.

*\$route*

A string for the base URL of the route you are adding.

*\$args*

An array of options for the endpoint. You can also pass an array of arrays if adding multiple methods for this endpoint. The most common arguments you will pass in are `methods` to specify the HTTP methods to enable for this endpoint, `callback` to specify the callback function to handle requests to the endpoint, and `args` to define query parameters that can be passed into the endpoint.

*\$override*

Should this new route override existing routes with the same name? If `true`, the new route will override the existing route. If `false`, the routes will be merged together, with the newer options taking precedence. Defaults to `false`.

To demonstrate how to add your own route, we dig in to the WP Single Sign-On plugin we've mentioned. This plugin contains both

code to set up a REST API endpoint and handle requests to it as well as code to make calls to that REST API endpoint in PHP.

## Setting Up the WordPress Single Sign-On Plugin

To test the WP Single Sign-On plugin in action, download and activate it on two separate WordPress sites. Go to Settings → WP SSO on each site. Set up one site as the *host* and the other as the *client*. You will need to copy the *Host URL* from the host site and paste it into the settings for the client site. The Host URL is the URL for a custom REST API endpoint created by the plugin!

Once the plugin is on both sites, you'll be able to log in to the client site using a username and password from a valid user on the host site. Users (e.g., the administrators) on the client site can still log in using their existing usernames and passwords for that site as well. The plugin only checks with the host site if a login was otherwise going to fail on the client site.

## Adding the /wp-sso/v1/check Route

First we need to register our new route. We will hook into the `rest_api_init` action and then use the `register_rest_route()` function to define our route.

```
function wpssso_register_routes() {
    $options = wpssso_get_options();

    // Make sure host option is enabled.
    if ( ! $options['host'] ) {
        return;
    }
```

```

register_rest_route(
    'wp-sso/v1',
    '/check',
    array(
        'methods' =>
        WP_REST_Server::READABLE,
        'callback' =>
        'wpssso_check_authentication_endpoint',
    )
);
add_action( 'rest_api_init', 'wpssso_register_routes' );

```

This function first gets an array of options for the WordPress Single Sign-On plugin. If the `host` option is not enabled, the route is not registered. The route we're going to add is what makes a host a host.

The namespace we've chosen is `wp-sso/v1`. We're using the slug of the plugin and also adding another segment to the URL, `v1`. If we ever need to make breaking changes to our API, we can keep the old API as `v1` and create new routes using a `v2` prefix. This way, users of our API can opt into the new version. The endpoint we add is called `check`, which will be setup to check whether the username and password passed in via Basic Authentication works for a valid user on the host site.

The arguments we pass define the available methods and the callback function to handle requests to the route. The value `WP_REST_Server::READABLE` is an alias for GET. The callback function is `wpssso_check_authentication_endpoint()`:

```

function wpssso_check_authentication_endpoint() {
    global $current_user;

    if ( ! empty( $current_user->user_login ) ) {
        $r = array(
            'success' => true,
            'message' => sprintf( 'Logged in as %s',
$current_user->user_login ),
            'user_login' => $current_user->user_login,
            'user_email' => $current_user->user_email,
            'first_name' => $current_user->first_name,
            'last_name' => $current_user->last_name,
        );
    } else {
        $r = array(
            'success' => false,
            'message' => 'Not logged in.',
            'user_email' => null,
            'user_login' => $current_user->user_login,
            'first_name' => null,
            'last_name' => null,
        );
    }
}

$r = rest_ensure_response( $r );

return $r;
}

```

The callback function here tests whether a user is logged in. If the user is logged in, we set the success value to true and pass along the username, email, first name, and last name of the user. If the user is not logged in, we set the success value to false and pass along null values for the user data.

Note that the return array in the previous example is passed through the `rest_ensure_response()` function. This function converts our array into a `WP_REST_Response` object. As you'll see later on, when we make requests to this new endpoint, we get back JSON containing the array of values we set up in the callback function.

## Bundling Basic Authentication with Our Plugin

For our `$current_user` checks to work, we need these API requests to be authenticated. To do that, we've bundled a Basic Authentication class in `/includes/basic-auth.php`. This authentication handler is the same as that linked to in the WordPress REST API Handbook, but we've 1) prefixed the function names to avoid conflicts, and 2) added the following code so Basic Authentication is enabled only for HTTPS traffic to our specific endpoint:

```
// Don't run if not using SSL
if ( ! is_ssl() ) {
    return $user;
}

// Don't run unless using our route.
if ( ! empty( $_REQUEST['rest_route'] ) ) {
    $rest_route = '/' . rest_get_url_prefix() .
$_REQUEST['rest_route'];
} else {
    $rest_route = $_SERVER['REQUEST_URI'];
}
if ( $rest_route != '/' . rest_get_url_prefix() . '/wp-
sso/v1/check' ) {
    return $user;
}
```

## Using the Endpoint We Set Up to Check User Credentials

WordPress sites running our Single Sign-On plugin and configured as a host will have the REST API endpoint described earlier such that when accessed by an authenticated user, they will return an array with details about that user. Client sites will access this API endpoint during login to check whether a user trying to log in exists on the host site. Let's see what that code looks like.

In the main *wp-sso.php* file there is a function `wpssso_authenticate()` that hooks into the `wp_authenticate` action hook. The `wpssso_authenticate()` function checks to make sure both a username and password have been entered and checks first whether those credentials will work on the local site. If they won't work on the local site, it uses them to make a basic authenticated request to the WordPress Single Sign-On endpoint on the host site. The request looks like this:

```
$url = $options['host_url'];
$args = array(
    'headers' => array(
        'Authorization' => 'Basic ' . base64_encode( $username .
        ':' . $password ),
    ),
);
$response = wp_remote_get( $url, $args );
```

The host URL is taken from options. We also set up an array of headers to pass along to the GET request. The `$username` and

`$password` variables that are passed into the `wp_authenticate` hook are encoded and included in the authorization header. Since we enabled Basic Authentication for our endpoint on the host site, the host site will attempt to authenticate the request using the username and password sent. We use the `wp_remote_get()` function to make the request.

If the credentials don't work on the host site, you will get a 500 error back from your request with the message "incorrect password."

If the credentials do work on the host site, the response will look something like this:

```
array(6) {
    ["headers"]=>
        ...
}
["body"]=>
string(162) "{\"success":true,...}"
["response"]=>
array(2) {
    ["code"]=>
        int(200)
    ["message"]=>
        string(2) "OK"
}
["cookies"]=>
array(0) {
}
["filename"]=>
NULL
["http_response"]=>
...
}
```

The rest of the `wpsso_authenticate()` function checks that the status on the response is 200 and then uses the `json_decode()` function on the response body to pull out the user data. If a user with the same email address exists on the local site, that user's password is updated. If no user exists yet with that password, a new user is created.

## Popular Plugins Using the WordPress REST API

Many plugins use the REST API, both to support their own user interfaces and to make the data and functionality of the plugins available to other apps. Following are some plugins that work with the WP REST API particularly well and can be used for reference.

### WooCommerce

WooCommerce has a fully featured API that can do pretty much anything you would want to do with WooCommerce short of processing checkouts. You can perform CRUD operations against products, orders, coupons, and customers. You can access reports and tax rate information. Be sure to check out the [detailed technical documentation for the WooCommerce REST API](#).

The WooCommerce REST API uses Basic Authentication over SSL. But instead of using the same username and password you would use to log in to the WordPress administrator dashboard, WooCommerce has a page in its settings to generate sets of API keys for specific

users. The API keys can be limited to read, write, or read/write permissions.

The keys are labeled as a consumer key and consumer secret, but when using them in your Basic Authentication API requests, you set them as the username and password, respectively.

## EXAMPLE: HIDE SALE BANNERS FOR PAYING CUSTOMERS

The `Customer` object returned by the WooCommerce API has a useful property called `is_paying_customer`. In some instances, you might want to show certain banners or ads to noncustomers only. Following is a quick example of how you could do that in JavaScript through the WooCommerce API.

First, here is the PHP code to register, localize, and enqueue our JavaScript. This is similar to the cookie authentication example shared earlier in this chapter, but now we are also passing the current user's ID and enqueueing our JavaScript:

```
function my_hide_sale_banner_script() {
    global $current_user;

    wp_register_script(
        'hide-sale-banner',
        plugins_url( 'js/hide-sale-banner.js',
        __FILE__ ),
        array( 'jquery' )
    );

    wp_localize_script( 'hide-sale-banner',
    'HSBSettings', array(
```

```

        'root' => esc_url_raw( rest_url() ),
        'nonce' => wp_create_nonce( 'wp_rest' ),
        'current_user_id' => $current_user->ID,
    ) );
}

wp_enqueue_script( 'hide-sale-banner' );
}

add_action( 'wp_enqueue_scripts',
'my_hide_sale_banner_script' );

```

Here is the JavaScript code that will get the current customer information using the WooCommerce API; if the user is already a paying customer, remove the banner element from the page.

```

jQuery(document).ready(function() {
    jQuery.ajax( {
        url: HSBSettings.root + 'wc/v3/customers/' +
        HSBSettings.current_user_id,
        method: 'GET',
        beforeSend: function ( xhr ) {
            xhr.setRequestHeader( 'X-WP-Nonce',
HSBSettings.nonce );
        },
    } ).done( function ( customer ) {
        if ( customer['is_paying_customer'] ) {
            jQuery('#sale-banner').remove();
        }
    } );
} );

```

Pretty neat. Other uses for the WooCommerce API could include generating coupons on the fly, using the sales report statistics to update a “X customers served!” banner, or updating product information on the fly as details are changed in other systems.

## BuddyPress

The BuddyPress team has been working on getting full WP REST CRUD coverage for the myriad BuddyPress datatypes. As of this writing, it's about 90% complete.<sup>4</sup>

Work on the BuddyPress REST API is being done through a feature plugin. To use the BuddyPress REST API, install and activate this plugin alongside BuddyPress. The plan is to merge the REST API support into the 5.0 version of BuddyPress.

Though the BuddyPress API is currently very much under active development, you can still do some amazing things with it in its present form. There are endpoints for Activities, Groups, XProfile Fields and Groups, Members, Notifications, Components, and more.

Most of the endpoints require authentication to return data (e.g., by using the WordPress REST API: OAuth 1.0a Server plugin).

### **EXAMPLE: HIGHLIGHT ACTIVITY FROM SPECIFIC USERS**

By default the */wp-json/buddypress/v1/activity* endpoint returns the most recent activity across all users. You can pare it down further by passing in a specific `user_id` as a parameter, and you can use the `per_page` parameter to change the number of results returned. So an authenticated request to a URL such as *https://yoursite.com/wp-json/buddypress/v1/activity?user\_id=1&per\_page=1* will return the one most recent activity item for the user with ID 1.

You could use code like the following, then, to grab the most recent activity item for that user and display it more prominently at the top of the page:

```
function my_highlight_admin_activity_script() {
?>
<style>
    li.activity.highlighted {
        background-color: #FFFFCC;
    }
</style>
<script>
    fetch( '/wp-json/buddypress/v1/activity?
user_id=1&per_page=1' )
        .then( function( response ) {
            return response.json();
        })
        .then( function( activity ) {
            const elements = document.querySelectorAll(
                "a[href='" + activity[0].link + "']");
            elements.forEach( function( element ) {
                const wrapping_div =
                    element.closest( 'li.activity' );
                wrapping_div.classList.add(
                    'highlighted' );
            });
        });
</script>
<?php
}
add_action( 'wp_footer',
'my_highlight_admin_activity_script' );
```

Adding the preceding code to a plugin or theme's *functions.php* file would add the JavaScript into the footer of all frontend pages. Typically, you would put this kind of JavaScript code into its own *.js* file and enqueue it, similar to this BuddyPress example. We wanted to show a few different ways of doing things.

The previous example also uses the `fetch()` method instead of `jQuery.ajax` to get the data from the BuddyPress API. We also use some vanilla JavaScript selector commands instead of the jQuery variants. This code should work on most modern browsers without your needing to load the jQuery API.

The BuddyPress REST API activity endpoint does not require authentication. So the fetch did not need to be authenticated. We simply hit the `/wp-json/buddypress/v1/activity?`  
`user_id=1&per_page=1` endpoint, passing in `user_id=1` to get activity for just that user and `per_page=1` to limit the results to the latest activity item.

## Paid Memberships Pro

Paid Memberships Pro has a simple but powerful REST API that you can use to answer a very important question: is a user a paying member?

There are other endpoints supported by Paid Memberships Pro and more in development, but the `/wp-json/wp/v2/users/2/pmpo_membership_level` endpoint added by the plugin is useful for connecting the member database stored within WordPress and Paid Memberships Pro to other applications.

## NOTE

Unlike the other plugins mentioned in this section, which create their own new routes for their API, Paid Memberships Pro extends the existing `/users/` route to include a new endpoint for checking a user's membership level.

You could imagine a Zapier or If This Then That (IFTTT) recipe that would check a user's membership using this endpoint before deciding between sending user data into different customer relationship management (CRM) silos. This one endpoint also makes it possible to manage your paying members in WordPress while controlling access to an application built on any platform that can make authenticated API calls.

### EXAMPLE: CHECK WHETHER A CERTAIN EMAIL ADDRESS HAS A MEMBERSHIP

To use the `pmpro_membership_level` endpoint we first need to ascertain a user's ID. To do that we can perform a search using the built-in `users` endpoint. We then take that ID and use it to query for the user's membership-level data.

Note that the example here uses Basic Authentication. As of this writing, Paid Memberships Pro does not handle its own authentication. You will need to have the Basic Authentication plugin active or change this:

```
function my_check_host_site_membership() {  
    global $current_user;  
  
    // Change these values.
```

```

$host_site_url = 'https://hostsite.com/';
$restricted_post_id = 2;
$apiuser = 'apiuser';
$apipassword = 'apipassword';

// We're only blocking the specific post ID
$queried_object = get_queried_object();
if ( empty( $queried_object ) )
|| $queried_object->ID != $restricted_post_id ) {
    return;
}

// If not logged in, redirect to the host site
if ( ! is_user_logged_in() ) {
    wp_redirect( $host_site_url );
    exit;
}

// Check for membership at host site.
$url = esc_url(
$host_site_url
. '/wp-json/wp/v2/users/?search='
. urlencode( $current_user->user_email )
);
$args = array(
    'headers' => array(
        'Authorization' => 'Basic '
        . base64_encode( $apiuser . ':' . $apipassword ),
    ),
);

// Make sure our first request was successful.
$response = wp_remote_get( $url, $args );
if ( empty( $response ) || $response['response'][ 'code' ] != '200' ) {
    wp_redirect( $host_site_url );
    exit;
}

// Make sure we found a user.
$response_body = json_decode( $response['body'] );

```

```

    if ( empty( $response_body ) ) {
        wp_redirect( $host_site_url );
        exit;
    }

    // The result from the user search is an array. Grab
    // the first one.
    $host_user = $response_body[0];

    // Check the user's membership at the host site.
    $url = esc_url(
    $host_site_url
    . '/wp-json/wp/v2/users/'
    . $host_user->id
    . '/pmpro_membership_level'
    );
    $response = wp_remote_get( $url, $args );

    // Make sure the second request was successful.
    if ( empty( $response ) || $response['response'][
    ['code']] != '200' ) {
        wp_redirect( $host_site_url );
        exit;
    }

    // Check for a membership level
    $membership_level = json_decode( $response['body'] );
    if ( empty( $membership_level ) ) {
        wp_redirect( $host_site_url );
        exit;
    }

    /*
        If we get here, $membership_level will
        contain information
            about the user's level. We could check for a
        specific level
            or just stop here to let users of all levels
        view this page.
    */
}

```

```
add_action( 'template_redirect',
'my_check_host_site_membership' );
```

In this example, you would change the host site URL to point to the home page of the WordPress site where Paid Memberships Pro is active with members. Users are also redirected to this URL. The \$restricted\_post\_id is the ID of the post you want to lock down for members. You could update this example to check for post meta or an array of posts or some other factor you want to restrict. With Basic Authorization enabled on the Paid Memberships Pro site, The \$apiuser and \$apipassword would just need to be the username and password of a user with the edit\_users capability.

The REST API is a powerful tool for enabling faster, more flexible functionality and applications to be built on top of WordPress. One example of an internal feature that relies heavily on the REST API is the new block editor, Gutenberg. We examine Gutenberg, blocks, and creating CPTs in the next chapter.

---

<sup>1</sup> Remove this in the audiobook version.

<sup>2</sup> The /?rest\_route=/ is for WordPress sites that don't have permalinks enabled.

<sup>3</sup> wp-json by default.

<sup>4</sup> Of course, the last 10% of any project takes at least as much time as the previous 90%.

# Chapter 11. Project Gutenberg, Blocks, and Custom Block Types

---

When the new WordPress editor was proposed in January 2017, Matt Mullenweg wrote:

*The editor will endeavor to create a new page and post building experience that makes writing rich posts effortless, and has “blocks” to make it easy what today might take shortcodes, custom HTML, or “mystery meat” embed discovery.*

Less than two years later, the block editor (aka the Gutenberg editor) was included in WordPress version 5.0, bringing with it a new way to edit posts and a new way to develop experiences with WordPress.

## NOTE

The original project to build the block editor was codenamed “Gutenberg.” We now prefer the term *block editor* or simply *WordPress editor*, but many people and posts you run across will refer to the editor itself as Gutenberg.

The Gutenberg team has put together the [Block Editor Handbook](#) to help both users and developers get up to speed with using Gutenberg. The Block Editor Handbook is a tight, well-written, and evolving piece of documentation that you should read. Like, right now go read

it, and then come back here. We will reference specific sections of this handbook later on.

In this chapter, we briefly cover the general features of the block editor, build a minimal block as a starting point, and then dive a bit deeper into the features most relevant to application development.

The block editor is built primarily in JavaScript (using the UI), and managed as a Node.js project. Though the Gutenberg handbook will help you code a block, it can be hard to tell what parts of the example code are JavaScript, React, or Gutenberg. Understanding how the various technologies work together will help you as you work to extend WordPress. We have tried our best to make it clear in our examples where the code comes from.

## The WordPress Editor

The current WordPress editor is built around the unit of blocks. These blocks, representing paragraphs, headings, lists, images, and more complex components, are placed in a series. Some blocks, like the group and column blocks, can have other blocks nested inside them. You can drag and drop blocks within the editor to reorder them (see Figure 11-1).

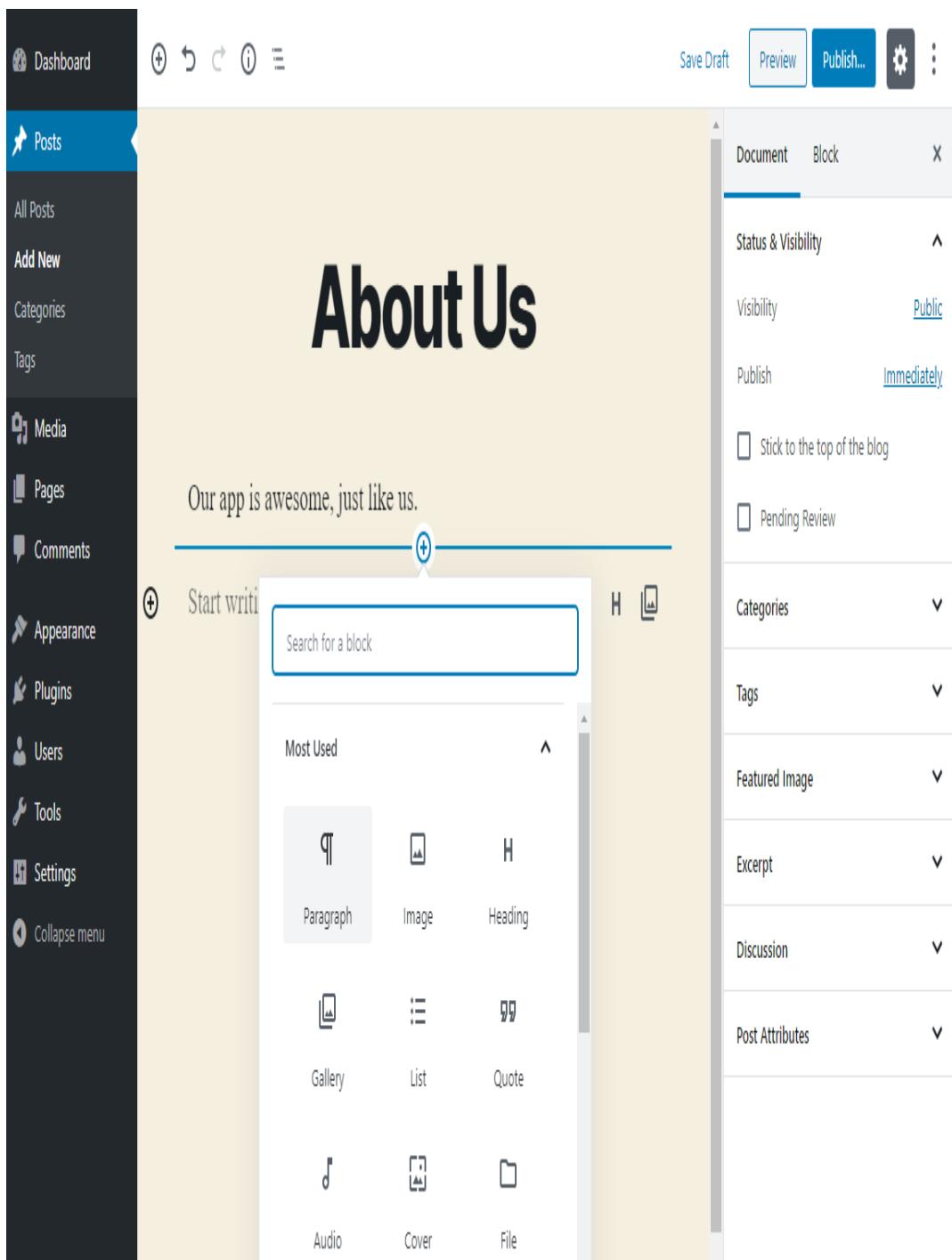


Figure 11-1. The WordPress Block Editor

Blocks can be added by clicking the + in the UI or by typing / followed by the block name within the empty space of the editor. Pressing the enter key within a paragraph block creates a new

paragraph block. Some blocks will automagically appear when you type certain Markdown-based characters into an empty block. The \* character will start an unordered list. The ## characters will start a secondary heading.<sup>1</sup>

One block can be edited at a time. You click on a block to bring it into focus and switch into its editor view. While in focus, the Block tab in the right sidebar contains settings specific to that block. Clicking another block or in the empty area of the editor will put the focus block into the frontend view.

The WordPress editor mirrors the frontend as much as possible. Some blocks will use placeholders or have slightly different styling to make things easier to edit, but developers are instructed to have the editor views of their blocks resemble the frontend view of their blocks as much as possible.

## The Classic Editor Plugin

If you have a site with plugins or features that require the classic editor, or you are just a big fan of the old TinyMCE-based editor, you can install [the Classic Editor plugin](#) on that site. This plugin allows you to selectively or globally use the classic editor to edit any post or page. According to the *README* file, “Classic Editor is an official WordPress plugin, and will be fully supported and maintained until at least 2022, or as long as is necessary.”

We suggest using the Classic Editor plugin as a last resort. While the plan is for that plugin to be supported through 2022, the block editor

will become an increasingly integral part of the WordPress dashboard and other plugins and themes built on top of it.

## Using Blocks for Content and Design

Most blocks available are content and design focused. They help you create rich design layouts without resorting to coding. It's outside the focus of this book to overview the currently popular divider, carousel, or gallery blocks, but, suffice it to say, there are many to choose from.

## Using Blocks for Functionality

Some blocks are more tied to the functionality of your WordPress site or app. The Paid Memberships Pro plugin includes a membership block to restrict any nested blocks based on membership level. Over time, plugins with complicated frontend UIs will ship with blocks to allow you more control over how that UI is laid out and configured. You can imagine fine-tuning the directory layout in BuddyPress through blocks and block settings.

## Creating Your Own Blocks

Using blocks is fun. Creating your own blocks is even funner...more fun...sometimes fun. It can be difficult, but let's start with the smallest possible block and build up from there.

### Minimal Block Example

The bare minimum required to add a block to the editor is to make a call to `wp.blocks.registerBlockType()` in some JavaScript run on the editor page:

```
wp.blocks.registerBlockType( 'bwawp/minimal', {
    title: 'Minimal Block Example',
    category: 'common',
    edit() {
        return 'Minimal block editor content.';
    },
    save() {
        return 'Minimal block frontend content.';
    },
} );
```

The full docs and list of options can be found in the [Block Registration section of the Block Editor Handbook](#).

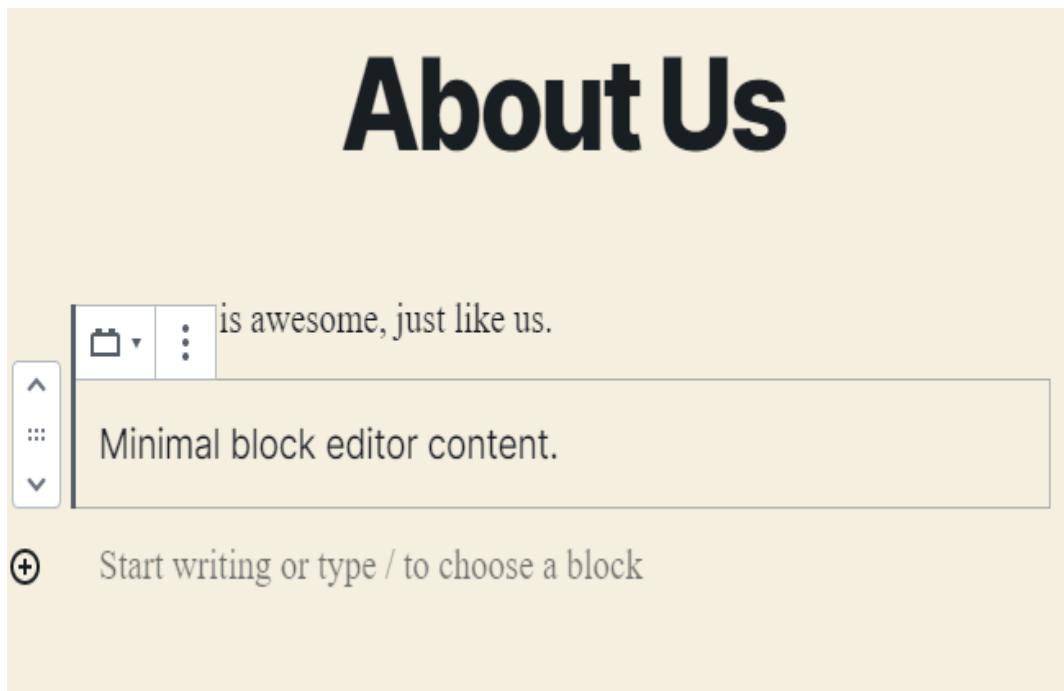
Our minimal block passes in the name of the block and an array of arguments. Names must be unique and should be prefixed with the namespace or slug of the plugin loading the block. Names must start with a letter and can contain only *lowercase* letters, numbers, and dashes.

Our minimal block also passes in a title and category. The title can be any descriptive string, to help users find it in the block list. Later, we will cover how to add a new category to group related blocks. The categories included in core WordPress are *common*, *formatting*, *layout*, *widgets*, and *embed*.

The workhorses of the `registerBlockType` attributes are the `edit` and `save` attributes, for which we pass in functions to

compute and return the structure of your block. You'll use the `edit` function when the block is rendered in the editor (see [Figure 11-2](#)). The `save` function is used when the post is saved, before blocks are serialized into the `post_content`.

In the minimal block example, we just return a simple string. Note that any HTML included in these strings is going to be sanitized and converted into HTML entities. To generate markup with your blocks, you are going to have to use the `wp.element.createElement()` function.



*Figure 11-2. Minimal block in the editor*

We can build a plugin with just one folder, and two files to load our block:

- `block.js`
- `minimal-block-example.php`

Here is code for *minimal-block-example.php* that will enqueue this *block.js* file:

```
/**  
 * Plugin Name: Minimal Block Example  
 */  
function enqueue_min_block() {  
    wp_enqueue_script(  
        'minimal-block',  
        plugins_url( 'block.js', __FILE__ ),  
        array( 'wp-blocks' )  
    );  
}  
add_action( 'enqueue_block_editor_assets',  
    'enqueue_min_block' );
```

This is similar to how we enqueued other JavaScript in [Chapter 9](#). In this case, we are hooking into the

`enqueue_block_editor_assets` action instead of the regular `wp_enqueue_scripts` action. Note also that our block JavaScript has `wp-blocks` as a dependency. As we add functionality to our blocks, we will need to include other Gutenberg-related packages here.

### NOTE

This example demonstrates the minimum requirements for a plugin as well. The only value you need in your header is the “Plugin Name.” The rest are optional. Some are highly recommended, as we cover in [Chapter 3](#).

From here, return to the docs at [WordPress.org](#) and go through the “Writing Your First Block Type” tutorial. This covers adding styles,

using editable fields, adding toolbars and settings, and creating dynamic blocks that update on the fly.

## Using Custom Blocks to Build App Experiences

Imagine our SchoolPress teachers want to be able to create new homework assignments. They could use Microsoft Word or Adobe Acrobat, or just type things up in a regular WordPress post. But we want something that integrates with the rest of the app we are building and will store assignment responses into a database for reporting and other functionality.

We could build a form with custom PHP. We could build something more dynamic in React that stores data into WordPress through the WordPress REST API. We could use the Advanced Custom Fields plugin along with your homework CPT to lay out the data needed. As with many tasks in WordPress development, or development in general, there are many reasonable options.

One option that makes some sense is to create a homework CPT and custom blocks and block templates. By building on the block editor, we start with a UI that is already familiar to our users. We can code our custom blocks and block templates so that homework assignments will always include data in the structured format our WordPress-based app will need.

### Enabling the Block Editor in Your CPTs

As of WordPress 5.3, CPTs will default to using the Classic Editor. At some point, all CPTs may be forced into using the new block editor. For now, to enable the block editor in your CPTs, say so explicitly when registering your CPT.

Here is the homework CPT example again, this time with the settings to use the block editor:

```
register_post_type(
    'homework',
    array(
        'labels' => array(
            'name' => __( 'Homework' ),
            'singular_name' => __( 'Homework' )
        ),
        'public' => true,
        'has_archive' => true,
        'supports' => array( 'title', 'editor' ), // New
        'show_in_rest' => true, // New
    )
);
```

The property 'supports' => array( 'title', 'editor' ) adds support for the title field and the block editor. Since the block editor uses the REST API to update posts, we need to also add REST API support by adding 'show\_in\_rest' => true.

## Block Categories

When you're adding multiple related blocks, it is helpful to put them into the same category so they will appear together in the block list. Setting the category of a block is as simple as changing the

category attribute in the registerBlockType() arguments in your block JavaScript. WordPress needs to know about the new block category, though. To do that, use the `block_categories` filter:

```
// Add a Homework block category.
function my_block_categories( $categories, $post ) {
    return array_merge(
        $categories,
        array(
            array(
                'slug'  => 'homework',
                'title' => 'Homework',
            ),
        )
    );
}
add_filter( 'block_categories', 'my_block_categories' ), 10,
2 );
```

Each category in the array of `$categories` is an array with two keys: `slug` and `title` (see [Figure 11-3](#)).

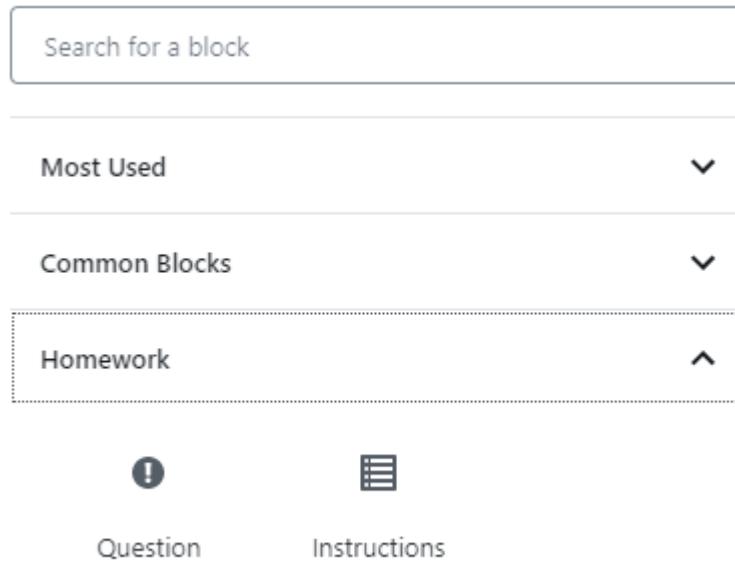


Figure 11-3. The Homework Block Category with the Homework and Instructions blocks

## The Homework Blocks

To support our homework app, we add a couple of blocks. The *homework/instructions* block will go at the top of our homework and include a due date set via block settings. Multiple *homework/question* blocks will go below. Each question block has a `RichText` field to enter the question and a question type set via the block settings. The question types could be true/false, multiple choice, or essay.

The full code for these blocks can be found in the [Chapter 11 folder](#) of this book's GitHub repository.

## Limiting Blocks to Specific CPTs

We want our instructions and question blocks to be used only on posts with the homework post type. To accomplish this, we can add

some code to check the post type of the current post being edited and unregister our post types if it's not a homework post.

At the bottom of your *block.js* file, add the following code:

```
// Deregister instructions on non-homework posts.
wp.domReady( function() {
    if( wp.data.select('core/editor').getCurrentPostType()
!= 'homework' ) {
        wp.blocks.unregisterBlockType(
'homework/instructions' );
    }
});
```

The `wp.domReady()` function is similar to jQuery's `document.ready()` function, waiting until the full page has loaded before running the code within it. We need to do this because the post type of the current post being edited won't be available until then. This is why you need to unregister the block instead of checking the post type before registered. To use the `wp.domReady()` function, make sure your block JavaScript is enqueued with `wp-dom-ready` as a dependency.

The current post type is pulled out of the WordPress data module using the command

`wp.data.select('core/editor').getCurrentPostType()`. To do this, make sure your block JavaScript is enqueued with `wp-edit-post` as a dependency.

Tons of useful information is available using similar methods listed in the [Data Module Reference](#). For more on how the data module is

built and how to set up your own data stores, check out the [data package documentation](#).

## Limits CPTs to Specific Blocks

Earlier, we showed how to keep our blocks from being used in other post types. WordPress also provides a `allowed_block_types` filter. This can be used to allow only select block types across your whole site or network. Or you can check the `post_type` on the `$post` parameter passed in and allow only select block types when editing posts of that type.

The following code limits homework posts to using a select few core blocks along with our instructions and question blocks:

```
// Only allow certain blocks on homework posts.
function my_allowed_block_types( $allowed_blocks, $post ) {
    if ( $post->post_type == 'homework' ) {
        $allowed_blocks = array(
            'core/block',
            'core/image',
            'core/paragraph',
            'core/heading',
            'core/list',
            'homework/instructions',
            'homework/question',
        );
    }
    return $allowed_blocks;
}
add_filter( 'allowed_block_types', 'my_allowed_block_types',
10, 2);
```

By default `$allowed_blocks` is null, telling WordPress to allow all blocks. Since blocks are added in JavaScript, after all PHP has loaded, the filter can't know what custom blocks will be added. But if you set a list of allowable blocks using code like that shown in the previous example, it will make sure only blocks in those list get loaded.

## Block Templates

When you register a CPT, you can pass a `template` attribute that will specify a group of blocks to be added to new posts of that type by default. Here is a version of our callback to register the homework post type that defaults to an instructions block up top, followed by one true/false question and one essay question:

```
// Register the homework CPT.
register_homework_post_type(
    'homework',
    array(
        'labels' => array(
            'name' => __( 'Homework' ),
            'singular_name' => __( 'Homework' )
        ),
        'public' => true,
        'has_archive' => true,
        'supports' => array( 'title', 'editor' ),
        'show_in_rest' => true,
        'template' => array(
            array( 'homework/instructions' ),
            array( 'homework/question',
                array( 'content' =>
                    'True/false 1.',
                    'question_type'
                ) => 'true_false'
            )
        )
    )
)
```

```
        ) ,
    array( 'homework/question',
           array( 'content' => 'Essay
question.' ,
=> 'essay'
           )
      )
) ;
```

With the above code in place, new homework posts will use our template and start with the instructions block and a few question blocks already in place. You can click on each block to edit their content (Figure 11-4).

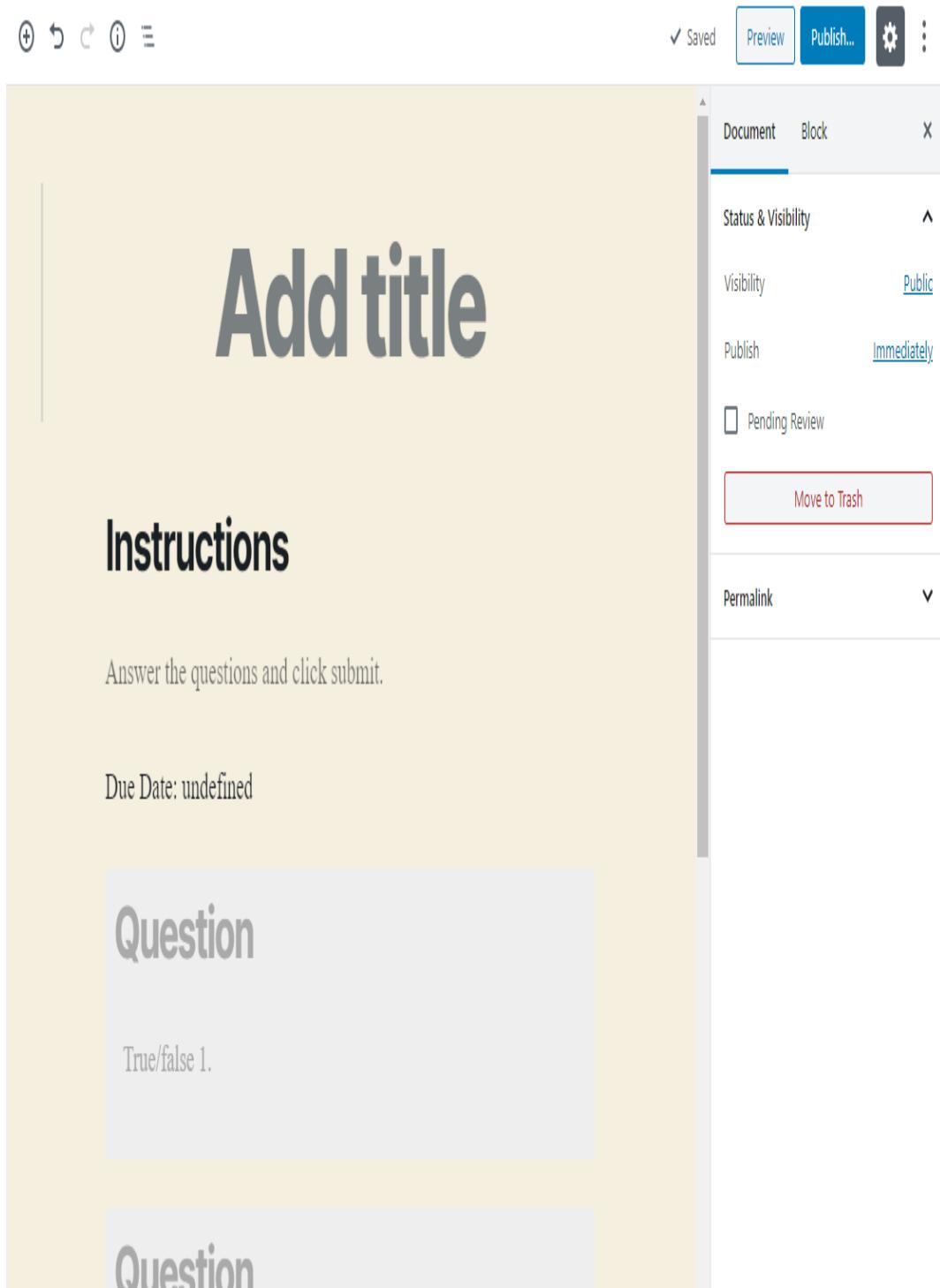


Figure 11-4. Default template for new homework posts

You can also add templates to core post types or other CPTs after they are registered, add templates in your block JavaScript, lock templates

so they cannot be rearranged or deleted, and next templates. Read about this and more in the [Block Templates section of the Block Editor Handbook](#).

## Saving Block Data to Post Meta

By default, attributes on blocks are stored inside of specially formatted HTML comments in the `post_content` body. The `instructions` block looks something like this before being converted for the frontend:

```
<!-- wp:homework/instructions -->
<p class="wp-block-homework-instructions-content">Email me your answers.</p>
<p class="wp-block-homework-instructions-due_date">Due Date: 2020-11-01</p>
<!-- /wp:homework/instructions -->
```

If our block is configured correctly, the content and due date are editable from the block editor and settings panel for those blocks. And things will then show up on the frontend of the website.

However, we might want to be able to sort homework assignments by due date or develop other reports around data set in our blocks. This is the kind of thing that would be possible if that data were stored in post meta.

To link block attributes to post meta, we need to register our post meta fields. We don't always register post meta. You can just call the `update_post_meta()` function directly and add any data you want. However, when manipulating post meta through the REST API, we need to register it:

```
register_post_meta( 'homework', '_homework_due_date', array(
    'show_in_rest' => true,
    'single' => true,
    'type' => 'string',
    'auth_callback' => function() {
        return current_user_can( 'edit_posts' );
    }
) );
```

Now we can simply set the source in the block attribute to link to our post meta:

```
wp.blocks.registerBlockType( 'homework/instructions', {
    // ...

    attributes: {
        content: {
            type: 'array',
            source: 'children',
            selector: 'p',
        },
        due_date: {
            type: 'string',
            meta: '_homework_due_date',
            source: 'meta',
            default: '',
        }
    },
    // ...
}
```

You can find more details on how to edit post meta through blocks in the “[Store Post Meta with a Block](#)” tutorial in the [Block Editor Handbook](#).

## Tips

Gutenberg development is complicated, fast-changing, and worthy of a book of its own. We have covered the basics here and a few techniques that should be useful to app developers working with Gutenberg. Here are some general tips that will help and some areas for further reading.

### **Enable WP\_SCRIPT\_DEBUG**

By default, WordPress will use minimized JavaScript and bundle all of its JavaScript into one file. This makes JavaScript errors in the console largely unusable. While doing any kind of JavaScript development, but especially when working with Gutenberg, you will want to enable `WP_SCRIPT_DEBUG`.

Adding `define( 'WP_SCRIPT_DEBUG', true );` to your *wp-config.php* file tells WordPress to use unminimized versions of scripts and load all JavaScript files individually. This way, errors in the debug bar console will point to useful filenames and line numbers that you can cross-reference in your codebase.

### **Use filemtime() for the Script Version**

When you enqueue JavaScript, you can set a version. Because browsers will aggressively cache JavaScript files, you can update the version of the script to break that cache. When working on your JavaScript files, you are going to be editing and updating them a lot. Setting the version to the last modified date of the underlying file is

one way to keep your browser from caching that JavaScript file while working on it.

Here is a version of the code to enqueue the question blocks that uses the `filemtime()` function for the script version:

```
wp_register_script(
    'homework-question',
    BWAWWP_URL . 'homework-cpt/blocks/question/blocks.js',
    array( 'wp-blocks',
        'wp-element',
        'wp-editor',
        'wp-components',
        'wp-dom-ready',
        'wp-edit-post',
    ),
    filemtime( BWAWWP_DIR . 'homework-
cpt/blocks/question/blocks.js' )
);
```

Before you deploy this code in the form of a plugin, you'll want to update the version of the script to be the version number of the plugin or a specific version number for the script.

## More Tips

Zac Gordon's "31 Tips for Gutenberg Development" includes some great tips and also direction into advanced topics in block development.

## Learn JavaScript, Node.js, and React More Deeply

Learning vanilla JavaScript, Node.js, and React more deeply will help you with your WordPress block development. At the very least, take some time to read up on, experiment with, and gain a general understanding of how each of these technologies works. You'll find that debugging issues with Gutenberg development is much easier if you can better tell which layer of your stack the issue is coming from:

### *JavaScript*

We cover JavaScript and its history with WordPress in [Chapter 9](#). The stronger your JavaScript skills, the easier you will find it to develop blocks. *JavaScript: The Definitive Guide*, by David Flanagan, and *JavaScript: The Good Parts*, by Douglas Crockford, are two great books for learning JavaScript. You can browse them to pick up the basics and use them for reference or dive into them to “learn JavaScript deeply.”

### *Node.js\**

Node.js is a JavaScript runtime that will run in your server environment. Several Node.js-based tools are used in the typical Gutenberg build setup, including webpack, Babel, and the Node Package Manager (npm). Webpack is used to compile all your JavaScript into a single bundle to be deployed to the browser. Babel is used to convert ESNext and JSX code into JavaScript that will run across the widest range of browsers. The npm is used to install JavaScript packages and manage the dependencies between them. Both webpack and Babel are typically installed as node modules. You may also want to use additional node libraries in the JavaScript code for your blocks. You can install and manage those via npm. The “[JavaScript Build Setup](#)” section of the [Block Editor Handbook](#) will walk you through installing node and npm, then installing the `@wordpress/scripts` package, which

includes the recommended default configurations for webpack and Babel.

## *React*

We give a full description of React in [Chapter 9](#). In summary, React is a JavaScript library for building interactive UIs that is used extensively in the block editor screens. A WordPress block can be viewed as a collection of React UI components designed to serve a common editor or content style use case. WordPress has its own library of React components specific for building blocks. These can be combined with other React components or custom developed components. Experimenting with React outside the context of building a Gutenberg block will help you to get familiar with React in general. When you run into issues building WordPress blocks, you will be better equipped to know if those issues are React issues or WordPress issues, which will help you to know where to go for help. If you'd like to dive deep into React, check out the book [\*Learning React\*](#), by Alex Banks and Eve Porcello. Just going through the [official tutorial](#) on the React website or one of the other examples there will be very useful as well.

---

<sup>1</sup> A full list of formatting shortcuts and other useful shortcut keys can be found [on the WordPress site](#).

# Chapter 12. WordPress Multisite Networks

---

With the release of WordPress version 3.0 came WordPress Multisite. WordPress Multisite was known as WordPress Multiuser (WPMU) prior to version 3.0 because it was a separate open source project. Since WordPress and WPMU shared most of the same code, it made sense to roll it all into one project. Multisite gives WordPress administrators the ability to create their own network with multiple sites. All sites on a Multisite network share the same database and same source files. When Multisite is set up, new tables are created in the database for each new website that is created on the network.

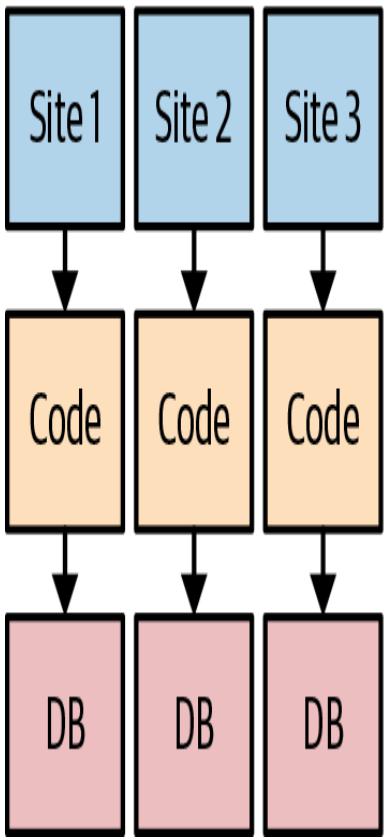
## Why Multisite?

If you are running more than one installation of WordPress, especially if they are mostly similar websites, you should consider using Multisite to save your valuable time and/or money. Imagine updating all of your WordPress installs at the same time instead of separately. Some benefits of Multisite includes the following (see Figure 12-1):

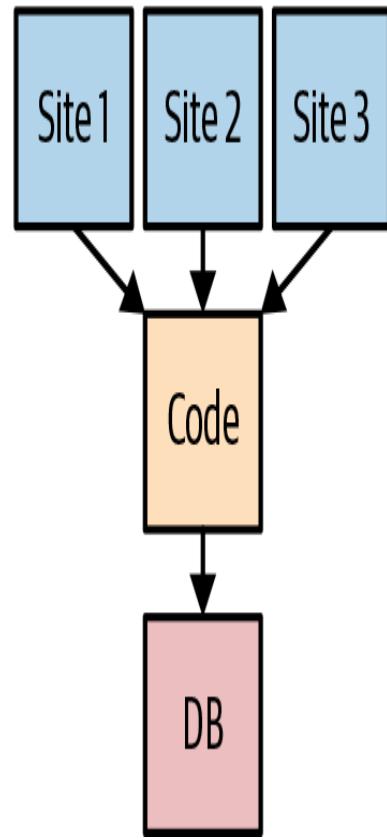
- Sharing a common set of plugins, themes, and custom code across many sites.
- Managing all of the users on your network in one location.

- Accessing all of your websites with one super administrator account.
- Updating core WordPress and installed plugins and themes at one time and in one place instead of via multiple websites.
- Easily deploying a new website with a few clicks instead of building a new installation from scratch.
- If you are using a child theme framework for all the sites on your network, you could make updates to all of your themes at the same time utilizing available hooks in the theme framework.
- Allowing users to set up and manage their own sites on your network. An administrator of one site might have a different role on another.

### Three Standalone Sites



### Multisite Network



*Figure 12-1. Multisite versus standalone sites*

Following are some good examples of when to use WordPress Multisite:

- You need to run a network of similar websites for which each site administrator manages their own site content. Let's say you need to build a network of school websites for a school district where each school has its own website with mostly the same features. Each site administrator needs to be able to make branding updates via theme settings and update all of the content. WordPress Multisite, yo!
- You need one website in multiple languages, run by different people, and you don't have faith in the WordPress

Multilingual Plugin (WPML) plugin. WordPress Multisite, yo!

- You want to set up hundreds of affiliate websites that you can easily manage and add new features. WordPress Multisite, yo!

## Why Not Multisite?

Many of the reasons to use Multisite are also reasons to avoid it. For some projects, managing all users on the same network will cause confusion or even legal issues. For some projects, you may need to run different versions of certain themes, plugins, or WordPress itself.

Third-party plugins can also be an issue when running a Multisite setup. While most plugins work well on WordPress Multisite, some plugins may not work as expected. For example, a plugin that modifies the new user screen in the administrator may not account for the fact that this screen is different when using Multisite. Or a plugin could store data networkwide when you need it to work on a per-blog basis. Again, most plugins work fine, but once in a while you run into a plugin that needs to be fixed, replaced, or altered to work with your specific Multisite setup.

In general, you shouldn't upgrade your WordPress installation to a Multisite unless you really need multiple websites. If you do need multiple sites, consider the benefits and downsides to hosting them in the same WordPress network.

## Multisite Alternatives

Here are some alternatives to consider before using Multisite on your WordPress project.

## **Multiple Authors or Categories on the Same WordPress Site**

If you are primarily interested in organizing your content into separate “sites” or “sections,” consider using multiple blog authors or categories on one WordPress site. If you have multiple authors on the same site, you can update your theme to style each author’s post archive to look like a unique site.

If you’d like to post behind-the-scenes-style content to a separate blog, you can create a post category for that content. You can use filters to keep that content from being included in the main index. You can link to the archive view for that one post category and style it separately to look like a “blog” for your larger site.

## **Custom Post Types**

Another way to organize content to give a “separate sites” feel without actually setting up separate sites is to use CPTs. Similar to using different post categories, you can filter your main index to exclude certain CPTs and treat the archive view of some CPTs as a separate site.

## **Totally Separate Sites**

If you are trying to share content across a few sites or get a few sites to work more seamlessly together, you might consider putting them

on the same WordPress Multisite network. Alternatively, you can keep them as separate sites and use the REST API or tools like the WordPress Single Sign-On (SSO) plugin we developed to share content and users across each site (these are covered in detail in Chapter 10).

## Use a WordPress Maintenance Service

If you are most excited about the ability to update WordPress themes and plugins from one spot, consider using a WordPress maintenance service. There are many popular ones to choose from. These apps, plugins, and services allow you to manage and update multiple WordPress sites from one dashboard.

- [Calypso](#), by Automattic
- [WordPress Website Management Dashboard](#)
- [WordPress Management for Professionals](#)
- [InfiniteWP](#)

The [WordPress Command Line Interface](#) can also be used to update multiple WordPress sites at once. For more details, see this [good writeup](#) by Phil Banks.

## Multitenancy

If you like the concept of a shared repository for WordPress, theme, and plugin files but don't want the shared database architecture of WordPress Multisite, you should look into multitenancy solutions. A simplistic view of multitenancy for WordPress is that it's like using a symbolic link for your core WordPress folder and sharing those files

across several WordPress installs on the same server or server cluster. In practice, it takes some work to get WordPress to respect this setup. A good place to start to learn more about multitenancy for WordPress are these talks by Cliff Seal on [WordPress.tv](#).

## Setting Up a Multisite Network

Once you've determined that a Multisite network is what you need, you'll need to set it up. Although setting up Multisite is not as easy as enabling it in a WordPress setting, it is fairly straightforward. The first thing you should do if you are not setting up a brand-new WordPress installation is to make a backup of your database and file directory.

Open your *wp-config.php* file in the root of your WordPress directory and add the following piece of code directly under the line that says

```
/* That's all, stop editing! Happy blogging.  
*/:
```

```
define( 'WP_ALLOW_MULTISITE', true );
```

Refresh your WordPress administrator dashboard and hover over the Tools menu item. On the submenu that opens, click Network Setup. On the Network Setup page you should see a few form fields asking you for the following information:

### *Subdomain or subdirectory*

How do you want to build the subsites on your Multisite network?  
If you want subdomains like sub.domain.com, choose

“subdomain.” If you want domain.com/sub, choose “subdirectory.” You could always use a domain mapping plugin to map any domain you want to, whether a subdomain or subdirectory.

*Network title*

The name of your Multisite network.

*Admin email address*

The email of the network administrator, most likely your email address.

Once you have filled out the required information, click the Install button.

You should now be seeing two text boxes. The first text box will contain code that you need to copy and paste into your *wp-config.php* file, under the line of code you just previously added, beneath the line that says /\* That's all, stop editing! Happy blogging. \*/:

```
define( 'MULTISITE', true );
define( 'SUBDOMAIN_INSTALL', false );
define( 'DOMAIN_CURRENT_SITE', 'whatever.com' );
define( 'PATH_CURRENT_SITE', '/' );
define( 'SITE_ID_CURRENT_SITE', 1 );
define( 'BLOG_ID_CURRENT_SITE', 1 );
```

## NOTE

When we talk about WordPress Multisite, we talk about one *network* with many *subsites* under that network. The term *blog* is deprecated in many ways, but still shows up in the Multisite-related database tables and functions. A network is identified by a `site_id`, and a subsite is identified by a `blog_id`. Using the word *subsite* when referring to sites in the network helps to avoid confusion, but you'll need to get used to figuring out what kind of "site" a piece of code or documentation is referring to.

In this example, we chose to use subdirectories, which is why `SUBDOMAIN_INSTALL` is set to `false`. If we chose subdomains, `SUBDOMAIN_INSTALL` would be set to `true`.

The second text box contains code that you need to copy and paste into your `.htaccess` file, which should also be in the root directory of your WordPress installation:

```
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]

# add a trailing slash to /wp-admin
RewriteRule ^([_0-9a-zA-Z-]+/)?wp-admin$ $1wp-admin/
[R=301,L]

RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]
RewriteRule ^([_0-9a-zA-Z-]+/)?(wp-
(content|admin|includes)\.*$2 [L]
RewriteRule ^([_0-9a-zA-Z-]+/)?(.*\.\php)$ $2 [L]
RewriteRule . index.php [L]
```

## CAUTION

You may have a customized `.htaccess` file (this could be due to your host or you doing custom things), so make sure you are just replacing the default WordPress rewrite rules. You will also notice, depending on whether you choose subdomains or subdirectories, that the rewrite rules will be different.

Once you have copied and pasted the code from each text box into the appropriate files and saved them on your web server, you can refresh your browser.

You should now be prompted to log in again. Do so with your administrator account username and password. Boom! You should now be running WordPress Multisite as a super administrator, and will have full access to all sites on your network (regular admins have only full access to sites for which they are administrators). Be careful who you hand super administrator rights to.

If you decided to use subdomains instead of subdirectories, there are a couple of extra steps you should take to save you time in the long run. You should set up a subdomain wildcard record on your domain registrar DNS settings page. Remember how, when first installing WordPress, you set the A (Host) record to the IP address of your hosting account? Well, in the same place you did that, you should be able to add a host name of “`*`” instead of “`@`” and point it to the same IP address as the “`@`” record. This acts as a catchall for any subdomains on your main domain. In short, `whatever.whatever.com` or `somethingelse.whatever.com` will both be mapped to the same IP address as the main domain.

Depending on your hosting account, you may also need to set up a wildcard subdomain entry to point to the same directory where your main domain is pointing. So register a new subdomain like \*. <whatever>.com and point it to the same folder of <whatever>.com.

Why are we doing all this? We set up a wildcard for subdomains so that when we create new sites on the Multisite network they will automatically work. If we didn't do this, we'd have to manually add each subdomain we create to the domain registrar and to the host. If we set up the subdomain wildcard once, it automatically works for any subdomain sites we create on our Multisite network.

#### NOTE

Depending on your host you may still need to do some things manually on the server side. If you are using *cPanel* and want to point a domain to your Multisite installation, add an *add on domain* and make sure you point it to the root of your Multisite installation.

## Managing a Multisite Network

In your WordPress administrator menu bar, go to My Sites → Network Administrator to administer your new Multisite network. You can also go to <whatever>.com/wp-admin/network/ to get to the network dashboard. The network administrator area looks similar to any other site administrator area on the network. To help keep track of where you are, look for the /network/ at the end of the address bar in your browser, because it can be easy to get confused.

## Dashboard

Your network dashboard is set up very similarly to the default dashboard you are used to seeing on a standard WordPress installation except the Right Now widget displays links to quickly add new network sites or users. It also has two text boxes to search for specific users or sites. Just like the regular WordPress dashboard, you can completely customize this network dashboard using plugins or custom code.

## Sites

Under Sites, you will manage all of the sites on your network. You can add any number of sites you like and even give your network users permission to create their own sites.

Adding a new site is pretty straightforward; at the top of the sites page, click the Add New button, or, on the Sites submenu, click Add New Link to get to the Add New Site page, where you'll see the following fields:

### *Site address*

Depending on how you set up your network, the address you enter will either be a subdomain or a subdirectory.

### *Site title*

The title or name of your new site.

### *Administrator email*

The email address of the administrator of your new site. This does not have to be your email address; it could be a client or a user for

whom you're setting up a new WordPress.

Click the Add Site button, and voilà, instant WordPress website. That sure saves a lot of time setting up a brand-new WordPress installation.

## Users

All the sites you create on your network will pull from the same users pool. Technically, all users are stored in the `wp_users` table, and they each have user metadata tying them to one or more sites on your network. From the network users page, you can manage all the users on your sites, make any user a super administrator with rights to manage the entire WordPress Multisite network, and see what sites each user is a member of.

To get to the Add New User page, at the top of the Users page, click the Add New button, or, on the Users submenu, click the Add New link. On the Add New User page, you'll see the following fields:

### *Username*

The username of the new user you are creating. Remember, with all lowercase and this should have no spaces or special characters.

### *Email*

The email address of the new user you are adding.

Once you click the Add User button, the user you added should receive an email with their username and password, which they can use to log in to the default top-level website with a default role of what you set a new user's role to be. Adding users this way may not

be ideal, as you will have to take another step to add them to specific subsites on your network. Depending on your situation, it might be easier for you to add new users to subsites directly from within that subsite, where you can add a new user the same way you would on a typical WordPress installation. If you try to add a user to a site and that username already exists on the network, you will receive a message indicating that the username already exists. At that point, you can add the user to the site by adding their username to the Add Existing User section and choosing a role and whether to send a confirmation email to that user.

## Themes

These are all the themes available in the */wp-content/themes/* directory. You can control all of the themes the sites on your network can use. You must network-enable a theme before a site on your network can use it. If you don't do so, it won't even show up as an option to activate on a site's Appearance → Themes page.

## Plugins

These are all of the plugins available in the */wp-content/plugins/directory*. You can network-activate plugins so that they will automatically run on all sites on your network, including new sites. Network-activated plugins will not show up on the individual site's plugins page. In fact, unless you specifically enable the plugins menu on the network settings page (see next section), individual site administrators won't even see the plugins page.

If you allow each site administrator to manage their own plugins, they will only be able to activate plugins that are already installed; they will not be able to install their own plugins. This is good because you want to know what plugins are available to all of the sites on your network. You don't want a site administrator to be able to install any plugin they want or a custom plugin that could potentially have a negative effect on other sites on your network.

To add a new plugin at the network level, you would add it the same way you would on a normal installation of WordPress.

### NOTE

Not all plugins should be network-activated. Consult the specific plugin's documentation when deciding whether to network-activate a plugin or to activate it separately on each site. In general, if it's only needed on a subset of sites or needs to maintain data or CPTs for each site separately, a plugin should be activated for each site separately.

## Settings

These settings are unlike the typical WordPress settings that you can update for a standard WordPress site; these are network-wide settings.

If you click the Settings link on the lefthand administrator navigation menu, you should see the following options:

### *Operational settings*

- Network name

- Network administrator email

### *Registration settings*

- Allow new registrations
- Registration notification
- Add new users
- Banned names
- Limited email registrations
- Banned email domains

### *New site settings*

- Welcome email
- Welcome user email
- First post
- First page
- First comment author
- First comment URL

### *Upload settings*

- Site upload space
- Upload file types
- Max upload file size

### *Menu settings*

- Enable administration menus

## **Updates**

Just as in a standard WordPress installation, you can update core WordPress or outdated plugins and/or themes all from this page. The beauty of a Multisite network is that you can update everything once, and across all the sites on your network. This is far more efficient than running updates on multiple WordPress installs. Update WordPress, plugins, and themes...done and done!

## **Multisite Database Structure**

All sites on a Multisite network share the same database. Enabling WordPress Multisite creates a few new tables in your existing database.

### **Networkwide Tables**

#### **WP\_SITE**

The `wp_site` table stores basic information about your Multisite network like the ID, domain, and path. This table will usually contain only one record. Table 12-1 shows the structure of the `wp_site` table.

*Table 12-1. Database schema for wp\_site table*

Column	Type	Collation	Null	Default	Extra
id	bigint(20)		No	None	AUTO_INCREMENT
domain	varchar(200)	utf8_general_ci	No		
path	varchar(100)	utf8_general_ci	No		

## WP\_SITEMETA

The `wp_sitemeta` table stores all of the network-wide options or settings. [Table 12-2](#) shows the structure of the `wp_sitemeta` table.

*Table 12-2. Database schema for wp\_sitemeta table*

Column	Type	Collation	Null	Default	Extra
meta_id	bigint(20)		No	None	AUTO_INCREMENT
site_id	bigint(20)		No	0	
meta_key	varchar(255)	utf8_general_ci	Yes	NULL	
meta_val	longtext	utf8_general_ci	Yes	NULL	

## WP\_BLOGS

The `wp_blogs` table stores information about each site created on the Multisite network. [Table 12-3](#) shows the structure of the `wp_blogs` table.

*Table 12-3. Database schema for wp\_blogs table*

Column	Type	Collation	N ul l	Default	Extra
blog_id	bigint(20)		N o	None	AUTO_INCREMENT
site_id	bigint(20)		N o	0	
domain	varchar(200)	utf8_general_ci	N o		
path	varchar(100)	utf8_general_ci	N o		
register_ed	datetime		N o	0000-00-00 00:00:00	
last_upd_ated	datetime		N o	0000-00-00 00:00:00	
public	tinyint(2)		N o	1	
archived	enum('0', '1')	utf8_general_ci	N o	0	
mature	tinyint(2)		N o	0	
spam	tinyint(2)		N o	0	
deleted	tinyint(2)		N o	0	
lang_id	int(11)		N o	0	

## WP\_BLOG VERSIONS

The wp\_blog\_versions table stores which database schema each site on the network is using. Table 12-4 shows the structure of the wp\_blog\_versions table.

Table 12-4. Database schema for wp\_blog\_versions table

Column	Type	Collation	Nu ll	Default	Extr a
blog_id	bigint(20)		No	0	
db_version	varchar(20)	utf8_general_ci	No		
last_updated	datetime		No	0000-00-00 00:00:00	

## WP\_SIGNUPS

The wp\_signups table also stores information about each user that registers on your network. Table 12-5 shows the structure of the wp\_signups table.

*Table 12-5. Database schema for the wp\_signups table*

Column	Type	Collation	Null	Default	Extra
domain	varchar(200)	utf8_general_ci	No		
path	varchar(100)	utf8_general_ci	No		
title	longtext	utf8_general_ci	No	None	
user_login	varchar(60)	utf8_general_ci	No		
user_email	varchar(100)	utf8_general_ci	No		
registered	datetime		No	0000-00-00 00:00:00	
activated	datetime		No	0000-00-00 00:00:00	
active	tinyint(1)		No	0	
activation_key	varchar(50)	utf8_general_ci	No		
meta	longtext	utf8_general_ci	Yes	NULL	

## WP\_REGISTRATION\_LOG

The wp\_registration\_log table stores information about each user that registers on your network, such as user ID, email address, IP

address, and blog ID. [Table 12-6](#) shows the structure of the `wp_registration_log` table.

*Table 12-6. Database schema for wp\_registration\_log table*

Column	Type	Collation	N u ll	Default	Extra
ID	bigint (20)		N o	None	AUTO_INCREMENT
email	varchar (255)	utf8_general_ci	N o		
IP	varchar (30)	utf8_general_ci	N o		
blog_id	bigint (20)		N o	0	
date_registered	datetime		N o	0000-00-00 00:00:00	

## Individual Site Tables

Every site added to your network is automatically given a `blog_id` when it is created. Each site also creates its own tables, adding its `blog_id` to each table name. Let's say we are creating the first additional site on our network besides our main site. It would be given a `blog_id` of 2, and the following tables would be created in the database:

- `wp_blog_id_options`
- `wp_blog_id_posts`

- wp\_{\$blog\_id}\_postmeta
- wp\_{\$blog\_id}\_comments
- wp\_{\$blog\_id}\_commentsmeta
- wp\_{\$blog\_id}\_links
- wp\_{\$blog\_id}\_term\_taxonomy
- wp\_{\$blog\_id}\_terms
- wp\_{\$blog\_id}\_term\_relationships

As you can see, these are the same tables included in a standard WordPress installation except that they have a `blog_id` in their names. For every new site you create on your Multisite network, these tables will be duplicated with that new site's `blog_id`.

#### NOTE

In WordPress version 3.5 the UI for links was removed from the administrator but the database table was kept for backward compatibility. If you need the links feature, you can use the [Link Manager plugin](#).

## Shared Site Tables

All users on your Multisite network share the same `wp_users` and `wp_usermeta` tables.

Users are associated with various sites on the network by a few user meta keys in the `wp_usermeta` table. If we added a new user to our second site on the network, these meta keys would be created:

- primary\_blog
- wp\_2\_capabilities
- wp\_2\_user\_level

A user can have only one primary\_blog but can be tied to multiple sites with the capabilities and user\_level meta keys. In the default WordPress installation and on the top-level site of a Multisite network, these meta keys are stored as wp\_capabilities and wp\_user\_level. When you add users to sites on the network, new meta key records are created for each user for the blog\_id you are adding them to with whatever role you added them as.

## Domain Mapping

By default, any subsites on your Multisite network will use your top-level domain to be either a subdomain or subdirectory unless you specifically set a subsite to use a separate domain.

As long as your DNS is set correctly and you're pointing the domain to your host and your host has a record of the domain pointing to your Multisite install, mapping a domain should be easy:

1. Go to your Network dashboard, and then, on the menu, click Sites.
2. Hover over any subsite on the network and click on the Edit link.
3. Change the Site Address (URL) field to the domain you want to use, making sure you use the full address:

*https://<whatever>.com*. If you have a SSL certificate for your newly mapped domain, make sure that it is installed and set up.

4. Click Save to update the page, go to your new domain in the browser, and hopefully put your feet up.

If you encounter any errors about cookies while attempting to log in to the backend of your mapped domain, you may need to update your *wp-config.php* file by adding the following line of code after the Multisite code you already added.

```
define('COOKIE_DOMAIN', $_SERVER['HTTP_HOST']);
```

Some managed WordPress hosting companies like WP Engine have an API available that you can use to programatically add domains to their DNS. This would save you the time of adding *add-on* or *parked domains* to your host's records manually. With WordPress Multisite integrated with an API like this, all you would need to do is add the new subsite or domain in the WordPress Network Administrator and the API would do the rest, setting up DNS records on the host. For more details, check out the [WP Engine API](#).

#### NOTE

Before WordPress 4.5, domain mapping required installing and configuring a third-party plugin to map domains. Now it's all baked into core WordPress.

## Random Useful Multisite Plugins

If a regular WordPress plugin is built correctly, it should work the way it was intended on one or more sites on your network. Developers can also build WordPress plugins specifically for Multisite. The following are a few of the more popular Multisite plugins and what they are built to accomplish.

## **Gravity Forms User Registration Add-On**

Add-On is a great plugin if you want to easily allow users to register to various sites on your Multisite network via a Gravity Forms shortcode or a widget you can place anywhere. It can also be configured to create a new site on your Multisite network when a user creates their account. This could be useful if you want to trigger the creation of a new site for a new user joining your Multisite network.

## **Member Network Sites Add-On for Paid Memberships Pro**

Add-On for Paid Memberships Pro allows you to charge users to create new sites on your network. We cover how to set this up in Chapter 15.

## **More Privacy Options**

Once installed, the WordPress Privacy plugin adds further levels of privacy to your Reading settings. These new levels are Network Users Only, Blog Members Only, and Admins Only, which makes your site visible to only whichever of these groups you choose.

## **Multisite Global Media**

The Multisite Global Media plugin shares a global Media Library across sites on the network. Basically it uses the Media Library of a specific site so any of the sites on a network can access the same shared resources.

## Multisite Plugin Manager

If you have a network of sites and you want to give access to site administrators to manage some of the plugins they are using, the Multisite Plugin Manager makes that very easy to manage. The plugin provides a network administrator UI for setting sitewide defaults for each site, you can also override the sitewide defaults for individual sites. There are three main settings for managing your plugins:

- Auto Activation
- User Control
- Mass Activation/Deactivation

## Multisite Global Search

Multisite Global Search allows you to search across the multiple sites on your network and receive results from all those sites. This plugin also comes with a built-in widget that can be used to display the search bar in the sidebar. Both the widget and the results page come with a customizable stylesheet. The plugin uses shortcodes, enabling you to insert the search in any templates you choose.

## Multisite Robots.txt Manager

[Multisite Robots.txt Manager](#) allows you to create custom *robots.txt* files for each website on the network and then quickly publish those files to the network or a website. This plugin will also instantly add sitemap URLs to all the *robots.txt* files. In addition, it will automatically detect 404 or old *robots.txt* files and allows for easy correction once identified.

## **NS Cloner: Site Copier**

There are a few plugins available to copy sites on your network but [NS Cloner: Site Copier](#) is a favorite. This plugin is very useful for anyone who needs to duplicate the settings and content from one site to another site on the network. If you are pumping out similar websites, you can create a “template” site all preconfigured exactly how you want it (theme settings, plugin settings, default content, etc.). You can put placeholders into your template site content so when you clone it to make a new site, an action is taken to automatically update these placeholders with whatever values you want. Why start with a blank site where you would have to reconfigure everything? Just clone whatever site you want and use it as a starting point. Save time and money!

## **WP Multi Network**

The [WP Multi Network](#) plugin allows one WordPress installation to have multiple networks, with each network having its own subsites. Build a network of networks! Remember how when Multisite is activated, it inserts just one row in the `wp_site` table? The WordPress Multi Network plugin basically allows you to add new

rows to that table and have new subsites set up under a different site\_id.

## Basic Multisite Functionality

When you activate WordPress Multisite, you can utilize Multisite-specific functionality that was sitting there dormant in WordPress core just waiting to be used.

### \$blog\_id

After reviewing the tables Multisite creates, we know that each site has a unique blog\_id. You can use this ID to point WordPress to the site you want to retrieve data from or push data to.

The global variable \$blog\_id will automatically be set to the site you are on unless changed with the switch\_to\_blog() function. This variable will be useful when writing custom Multisite functionality:

```
function wds_show_blog_id(){
    global $blog_id;
    echo 'current site id: ' . $blog_id;
}
add_action( 'init', 'wds_show_blog_id' );
```

If you are on the top-level site or original site on your network, you should see 1. If you are on the second site you created on your network, you should see 2.

### is\_multisite()

This function checks to see whether WordPress Multisite is enabled. If you are not running Multisite and try to use one of its functions, you may get an error. Always check that Multisite is enabled before executing Multisite-specific code:

```
function wds_run_multisite_functions() {
    if ( is_multisite() )
        echo 'Run whatever WordPress Multisite
functionality you want!';
}
add_action( 'init' , 'wds_run_multisite_functions' );
```

## get\_current\_blog\_id()

This function returns the `blog_id` that you are currently on. The function itself is literally two lines of code:

```
// core function get_current_blog_id
function get_current_blog_id() {
    global $blog_id;
    return absint($blog_id);
}
```

`get_current_blog_id()` is located in *wp-includes/load.php*.

## switch\_to\_blog( \$new\_blog )

This function switches the current blog to any blog you specify. This function is useful if you need to pull posts or other information from other sites on your network. You can switch back afterward using `restore_current_blog()`. Autoloaded options and plugins are not switched with this function. This function accepts one parameter,

`$new_blog`, which is a required integer of the ID of the site to which you want to switch.

If we want to switch the site we're currently on, we could run the following code in any plugin function or theme file:

```
echo 'current site id: ' . get_current_blog_id() . '<br>';
switch_to_blog(2);
echo 'new current site id: ' . get_current_blog_id();
```

The code should output something like:

```
current site id: 1
new current site id: 2
```

`switch_to_blog()` is located in *wp-includes/ms-blogs.php*.

## **restore\_current\_blog()**

Use this function to restore the current site, after calling the `switch_to_blog()` function. This function doesn't accept any parameters.

If we want to restore a switched site, we can run the following code:

```
echo 'current site id: ' . get_current_blog_id() . '<br>';
switch_to_blog(2);
echo 'new current site id: ' . get_current_blog_id() .
'<br>';
restore_current_blog();
echo 'original site id: ' . get_current_blog_id();
```

The code should output something like this:

```
current site id: 1  
new current site id: 2  
original site id: 1
```

`restore_current_blog()` is located in *wp-includes/ms-blogs.php*.

## **`get_blog_details( $fields = null, $get_all = true )`**

This function gets all the available details of a site and accepts two parameters:

*\$fields*

The ID or name of a specific blog, or an array of blog IDs or blog names. Defaults to the current blog ID.

*\$getall*

Default is set to `true` to return all available data in the object.

This function returns an object of the following variables:

*blog\_id*

The ID of the blog being queried.

*site\_id*

The ID of the site this blog ID is attached to.

*domain*

The domain used to access the blog.

*path*

The path used to access the site.

*registered*

Timestamp of when the blog was registered.

*last\_updated*

Timestamp of when the blog was last updated.

*public*

1 or 0 indicating whether the blog is public or not.

*archived*

1 or 0 indicating whether the blog is archived or not.

*mature*

1 or 0 indicating whether the blog has adult content or not.

*spam*

1 or 0 indicating whether the blog has been marked as spam or not.

*deleted*

1 or 0 indicating whether the blog has been deleted or not.

*lang\_id*

ID of the language the blog is written in.

*blogname*

The name of the blog.

*siteurl*

The URL of the site the blog belongs to.

*post\_content*

The number of posts in the blog.

To display the entire object returned by this function, we would run the following code:

```
$details = get_blog_details( 1 );
echo '<pre>';
print_r($details);
echo '</pre>';
echo 'Site URL:' . $details->siteurl;
echo 'Post Count:' . $details->post_count;
```

The code should return a similar object and string:

```
stdClass Object
(
    [blog_id] => 1
    [site_id] => 1
    [domain] => schoolpress.me
    [path] => /
    [registered] => 2013-03-01 00:23:26
    [last_updated] => 2013-04-01 14:18:59
    [public] => 1
    [archived] => 0
    [mature] => 0
    [spam] => 0
    [deleted] => 0
    [lang_id] => 0
    [blogname] => School Press
    [siteurl] => http://schoolpress.me
    [post_count] => 10
)
```

This site URL is <http://schoolpress.me>, and it has 10 posts.

`get_blog_details()` is located in `wp-includes/ms-blogs.php`.

**`update_blog_details( $blog_id, $details = array() )`**

This function updates the details for a blog and accepts two parameters:

`$blog_id`

A required integer of the ID of the blog you want to update.

`$details`

A required array of any of the fields from the blog's table as keys with any values you want to update.

To mark a particular site as deleted, we would run the following code:

```
update_blog_details( 2, array( 'deleted' => '0' ) );
```

`update_blog_details()` is located in *wp-includes/ms-blogs.php*.

## **get\_blog\_status( \$id, \$pref )**

This function is similar to the `get_blog_details()` function, except instead of returning an object of all of the fields in the `wp_blogs` table, it returns the value of one specific field:

`$id`

A required integer of the ID of the site you want to return a `wp_blogs` field from.

`$pref`

A required string of the field name from the `wp_blogs` table.

To show when the current site was registered, we could run the following code:

```
echo get_blog_status( get_current_blog_id(), 'registered' );
```

`get_blog_status()` is located in *wp-includes/ms-blogs.php*.

## **update\_blog\_status( \$blog\_id, \$pref, \$value )**

This function is similar to the `update_blog_details()` function, except instead of updating an array of fields in the `wp_blogs` table, it updates one specific field:

`$blog_id`

A required integer of the ID of the site you want to update a `wp_blogs` field for.

`$pref`

A required string of the field name from the `wp_blogs` table you want to update.

`$value`

A required string of the field value you want to update.

If we wanted to mark the current site as deleted, we could run the following code:

```
update_blog_status( get_current_blog_id(), 'deleted', '1' );
```

`update_blog_status()` is located in *wp-includes/ms-blogs.php*.

## **get\_blog\_option( \$id, \$option, \$default = false )**

This function saves you the hassle of using `switch_to_blog()` and then using the regular WordPress `get_option()` function or writing a custom SQL query if you want to grab an option from a specific site. This function returns an option value from any site on your network by passing in the following parameters:

*\$id*

A required integer of the ID of the site you want to get an option from. You can pass in `null` if you want to get an option from the current site.

*\$option*

A required string of the option name you want to get.

*\$default*

Optional string to return if the function does not find a matching option.

To get the `admin_email` of a particular site, run the following code:

```
echo 'The admin email for site id 2 is ' . get_blog_option(2, 'admin_email');
```

`get_blog_option()` is located in `wp-includes/ms-blogs.php`.

## **update\_blog\_option( \$id, \$option, \$value )**

This function updates any option for a particular site and accepts three parameters:

*\$id*

A required integer of the ID of the site you want to update an option on.

*\$option*

A required string of the option name you want to update.

*\$value*

A required string of the option value you want to update.

To update the `admin_email` of a particular site, we would run the following code:

```
update_blog_option( 2, 'admin_email', 'brian@alphaweb.com'  
);
```

`update_blog_option()` is located in `wp-includes/ms-blogs.php`.

## **delete\_blog\_option( \$id, \$option )**

This function deletes any option from a particular site and accepts two parameters:

*\$id*

A required integer of the ID of the site you want to delete an option on.

*\$option*

A required string of the option name you want to delete.

To delete a custom site option from a particular site, we run the following code:

```
delete_blog_option( 2, 'wds_custom_option' );
```

`delete_blog_option()` is located in *wp-includes/ms-blogs.php*.

### **get\_blog\_post( \$blog\_id, \$post\_id )**

This function gets a post from any site on the network and accepts two parameters:

`$blog_id`

A required integer of the blog ID of the site you want to get a post from.

`$post_id`

A required integer of the post ID of the post that you want to get.

To get the post title of the third post from the second site on our network, we could run the following code:

```
$post = get_blog_post( 2, 3 );
echo $post->post_title;
```

`get_blog_post()` is located in *wp-includes/ms-functions.php*.

### **add\_user\_to\_blog( \$blog\_id, \$user\_id, \$role )**

This function adds a user to any site on the network with a specified user role and accepts three parameters:

`$blog_id`

A required integer of the blog ID of the site you want to add the user to.

`$user_id`

A required integer of the user ID of the user that you want to add to the site.

`$role`

A required string of the role you want the user to have.

This function will return `true` if a user was added successfully; and if not, it will return a `WP_Error`.

If we wanted to add a specific user to the second site on our network with a role of Administrator, we could run the following code:

```
add_user_to_blog( 2, 5, 'administrator' );
```

`add_user_to_blog()` is located in *wp-includes/ms-functions.php*.

## wpmu\_delete\_user( \$user\_id )

This function deletes a user from the entire network, and deletes all posts authored by that user.

`$user_id`

A required integer of the user ID of the user that you want to delete.

This function returns `true` if the user was successfully deleted; if not, it returns `false`.

When Multisite is active, the regular `wp_delete_user()` function will remove a user from the current site instead of deleting the user entirely. So it is important to make sure you use the `wpmu_delete_user()` function and hook into both the `wp_delete_user` and `wpmu_delete_user` hooks in multisite situations. If you are coding something that needs to run in Multisite or in single-site settings, you can do this with the `is_multisite()` function first:

```
if ( is_multisite() ) {
    wpmu_delete_user( $user_id );
} else {
    wp_delete_user( $user_id );
}
```

`wpmu_delete_user()` is located in *wp-includes/ms-functions.php*.

### **`create_empty_blog( $domain, $path, $weblog_title, $site_id = 1 )`**

This function creates a new site on the network after making sure it doesn't already exist. It's also used by the network admin, to add new sites. It accepts four parameters:

`$domain`

A required string of the domain of the new blog.

`$path`

A required string of the path of the new blog.

`$weblog_title`

A required string of the title or name of the new blog.

`$site_id`

An optional integer of the site ID associated with the new blog.  
The default is 1.

If we want to add a new site to our network, we can run the following code:

```
create_empty_blog( 'someteacher.schoolpress.me', '/', 'Mr.  
Some Teacher' );
```

`create_empty_blog()` is located in *wp-includes/ms-functions.php*.

## Functions We Didn't Mention

We didn't cover every Multisite function, but we did cover most of the important ones. Though that depends on what you are trying to do. To find all available Multisite functions, look in the source code! You can find WordPress Multisite functions in the following files:

- *wp-admin/includes/ms.php*
- *wp-includes/ms-blogs.php*
- *wp-includes/ms-functions.php*

For more information, check out the “[Multisite functions](#)” section of the [WordPress Codex](#).

# Chapter 13. Localizing WordPress Apps

---

*Localization* (or *internationalization*) is the process of translating your app for use in different locales and languages. In this chapter we'll discuss the tools and methods available to you for localizing your apps, themes, and plugins.

## NOTE

You will sometimes see localization abbreviated as *I10n* and internationalization sometimes abbreviated as *i18n*.

## Do You Even Need to Localize Your App?

The market for web apps is increasingly global. Offering your app in other languages can be a strong advantage to help you gain market share against competition within your own locale/language and will also help to stave off competition in other locales/languages.

If you plan to release any of your code under an open source license, localizing it first is a good way to increase the number of developers who can get involved in your project. If your plugin or theme is localized, developers speaking other languages will be more likely to

contribute to your project directly instead of forking it to get it working in their language.

If you plan to distribute a commercial plugin or theme, localizing your code increases your number of potential customers.

If your target market is the United States only and you don't have any immediate plans to expand into other regions or languages, you may not want to spend the time preparing your code to be localized. Also, remember that each language or regional version of your app will likely require its own hosting, support, customer service, and maintenance. For many businesses, this will be too high a cost to take on in the early days of an application. On the other hand, you will find that the basics of preparing your code for localization (wrapping string output in a `_()`, `_e()`, or `_x()` function) are simple to do and often have other uses beyond localization.

Finally, it's important to note that sometimes localization means more than just translating your code. If your code interfaces with other services, you will need to make sure that those services work in different regions or be prepared to develop alternatives. For example, an important component of the Paid Memberships Pro plugin is integration with payment gateways. Before localizing Paid Memberships Pro, Jason made sure that the plugin integrated well with international payment gateways. Otherwise, people would have been able to use it in their language, but it wouldn't have worked with a viable payment gateway for their region.

# How Localization Is Done in WordPress

WordPress uses the *gettext translation system* developed for the GNU translation project. The gettext system inside WordPress includes the following components:

- A way to define a locale/language
- A way to define text domains
- A way to translate strings in your code
- *.pot* files containing all of the words and phrases to be translated
- *.po* files for each language containing the translations
- *.mo* files for each language containing a compiled version of the *.po* translations

Each of these components must be in place for your translations to work. The following sections explain each step in detail. At the end, you should have all of the tools needed to create a localized plugin and translated locale files.

## Defining Your Locale in WordPress

To define your locale in WordPress, simply set the `WPLANG` constant in your `wp-config.php` files:

```
<?php  
// use the Spanish/Spain locale and language files.  
define('WPLANG', 'es_ES');  
?>
```

## NOTE

The term “locale” is used instead of “language” because you can have multiple translations for the same language. For example, British English is different from US English. And Mexican Spanish is different from the Spanish spoken in Spain.

# Text Domains

The gettext specification uses *text domains* to organize the translation tables. In WordPress, this means that each plugin and theme should have its own unique text domain.

Technically, you can use anything for your text domain as long as it is unique and consistent and uses the proper syntax (all lowercase, using dashes but no underscores). In practice, the text domain for a plugin or theme should match the slug because most other plugins and tools will expect that. For example, plugins in the WordPress.org repository require that the text domain matches the plugin’s slug so GlotPress and other tools on the WordPress.org site function properly.

Here are some text domains being used in live projects:

- All code in WordPress core uses the text domain `default`.
- Paid Memberships Pro plugin uses the text domain `paid-memberships-pro`.
- Memberlite uses the text domain `memberlite`.

## Setting the Text Domain

For each of your site's localized plugins or themes, WordPress needs to know how to locate your localization files. This is done via the `load_plugin_textdomain()`, `load_textdomain()`, and `load_theme_textdomain()` functions. All three functions are similar, but take different parameters and make sense in different situations.

### NOTE

The gettext specification and WordPress functions use the concatenated term `textdomain` when referring to text domains. It is common, however, to spell out both words separately when writing about them.

Whichever function you use, it should be called as early as possible in your app because any strings used or echoed through translation functions before the text domain is loaded will not be translated.

Here are a few ways we could load our text domain in `includes/localization.php`:

```
load_plugin_textdomain( $domain,  
$abs_rel_path, $plugin_rel_path )
```

This function takes three parameters. The first is the `$domain` of your plugin or app (`schoolpress` in our case). You then use either the second or third parameter to point to the languages folder from which the `.mo` file should be loaded. The `$abs_rel_path` is deprecated, but still here for reverse-compatibility reasons. Just pass `FALSE` for this and use the `$plugin_rel_path` parameter:

```

<?php
function schoolpress_load_textdomain() {
    //load text domain from
    /plugins/schoolpress/languages/
    load_plugin_textdomain(
        'schoolpress',
        FALSE,
        dirname( plugin_basename(__FILE__) ) .
    '/languages/'
    );
}
add_action( 'init', 'schoolpress_load_textdomain', 1 );
?>

```

This code loads the correct language file from our languages folder based on the `WPLANG` setting in `wp-config.php`. To get the path to the current file, we use

`plugin_basename(__FILE__)`, and then `dirname(...)` to get to the path to the root plugin folder, since we are in the `includes` subfolder of our `schoolpress` plugin folder.

### NOTE

You may also see a “Text Domain” set in the PHP comment header of a plugin or theme. This is used by WordPress to translate the plugin meta information even if the plugin itself is disabled. Since WordPress 4.6, this header setting is optional and defaults to the plugin or theme’s slug if not set.

`load_theme_textdomain( $domain, $path )`

If you have language files for your theme in particular, you can load them through the `load_theme_textdomain()` function like so:

```

<?php
function schoolpress_load_textdomain() {

```

```

        load_theme_textdomain(
            'schoolpress', get_template_directory() .
        '/languages/'
    );
}

add_action( 'init', 'schoolpress_load_textdomain', 1 );
?>

```

`load_textdomain( $domain, $path )`

This function can also be used to load the text domain, but you'll need to get the locale setting yourself.

Calling `load_textdomain()` directly is not recommended for plugins or themes, but could be useful for projects in which you want to use a single domain across many different plugins.

Calling `load_textdomain()` directly also adds some flexibility if you want to allow others to easily replace or extend your language files. You can use code like the following to load any `.mo` file found in the global WordPress languages directory (usually `wp-content/languages/`) first, and then load the `.mo` file from your plugin's local languages directory second. This allows developers to override your translations by adding their own `.mo` files to the global languages directory:

```

<?php
function schoolpress_load_textdomain() {
    // get the locale
    $locale = apply_filters( 'plugin_locale',
                            get_locale(), 'schoolpress'
    );
    $mofile = 'schoolpress-' . $locale . '.mo';

    /*
     Paths to local (plugin) and global (WP) language
     files.
         Note: dirname(__FILE__) here changes if this
     code
     is placed outside the base plugin file.

```

```

        */
        $mofile_local = dirname( __FILE__
) . '/languages/' . $mofile;
        $mofile_global = WP_LANG_DIR . '/schoolpress/'
. $mofile;

        // load global first
load_textdomain( 'schoolpress', $mofile_global
);

        // load local second
load_textdomain( 'schoolpress', $mofile_local
);
}

add_action( 'init', 'schoolpress_load_textdomain', 1 );
?>
```

This version gets the locale via the `get_locale()` function, applies the `plugin_locale` filter, and then looks for a `.mo` file in both the global languages folder (typically `/wp-content/languages/`) and the languages folder of our plugin.

## Prepping Your Strings with Translation Functions

The first step in localizing your code is to make sure that every displayed string is wrapped in one of the translation functions provided by WordPress. They all work pretty much the same way: some default text is passed into the function along with a domain and/or other information to let translators know what context to use when translating the text.

Let's go over the most useful functions in detail.

`__( $text, $domain = "default" )`

This function expects two parameters: the `$text` to be translated and the `$domain` for your plugin or theme. It returns the translated text based on the domain and the language set in `wp-config.php`.

### NOTE

The `__()` function is really an alias for the `translate()` function used in the background by WordPress. There's no reason you couldn't directly call `translate()`, but `__()` is shorter and you'll be using this function *a lot*.

Here is an example of how you would wrap some strings using the `__()` function:

```
<?php
// setting a variable to a string without localization
$title = 'Assignments';

// setting a variable to a string with localization
$title = __( 'Assignments', 'schoolpress' );
?>
```

### `_e( $text, $domain = “default” )`

This function expects two parameters: the `$text` to be translated and the `$domain` for your plugin or theme. It echoes the translated text based on the domain and the language set in `wp-config.php`.

This function is identical to the `__()` function except that it echoes the output to the screen instead of returning it. Here is an example of how you would wrap some strings using the `_e()` function:

```
<?php
// echoing a var without localization
?>
<h2><?php echo $title; ?></h2>
<?php
// echoing a var with localization
?>
<h2><?php _e( $title, 'schoolpress' ); ?></h2>
```

In practice, you will use the `__()` function when setting a variable, and the `_e()` function when echoing a variable.

## `_x( $text, $context, $domain = "default" )`

This function expects three parameters: the `$text` to be translated, a `$context` to use during translation, and the `$domain` for your plugin or theme. It returns the translated text based on the context, the domain, and the language set in `wp-config.php`.

The `_x()` function acts the same as the `__()` but gives you an extra `$context` parameter to help the translators figure out how to translate your text. This is required if your code uses the same word or phrase in multiple locations, which might require different translations.

For example, the word *title* in English can refer both to the title of a book and also to a person's title, like Mr. or Mrs. In other languages, different words might be used in each context. You can differentiate between each context using the `_x()` function.

In the following (slightly convoluted) example, we are setting a couple of variables to use on a class creation screen in SchoolPress:

```
<?php
$class_title_field_label = _x( 'Title', 'class title',
'schoolpress' );
$class_professor_title_field_label = _x( 'Title', 'name
prefix', 'schoolpress' );
?>
<h3>Class Description</h3>
<label><?php echo $class_title_field_label; ?></label>
<input type="text" name="title" value="" />

<h3>Professor</h3>
<label><?php echo $class_professor_title_field_label; ?>
</label>
<input type="text" name="professor_title" value="" />
```

## NOTE

The `_x()` and `_ex()` functions are sometimes referred to as *\_ex\_plain functions* because you use the `$context` parameter to further explain how the text should be translated.

## **`_ex( $title, $context, $domain = “default” )`**

The `_ex()` function works the same as the `_x()` function but echoes the translated text instead of returning it.

## **Escaping and Translating at the Same Time**

In Chapter 7, we talked about the importance of escaping strings that are displayed within HTML attributes or in other sensitive areas. When also translating these strings, instead of calling two functions, WordPress offers a few functions to combine two functions into one. These functions work exactly as you would expect them to by first translating and then escaping the text:

- esc\_attr\_\_()
- esc\_attr\_e()
- esc\_attr\_x()
- esc\_html\_\_()
- esc\_html\_e()
- esc\_html\_x()

## Creating and Loading Translation Files

Once your code is marked up to use the translation functions, you'll need to generate a *.pot* file for translators to use to translate your app. The *.pot* file will include a section like the following for each string that shows up in your code:

```
#: schoolpress.php:108
#: schoolpress.php:188
#: pages/courses.php:10
msgid "School"
msgstr ""
```

The preceding section says that on lines 108 and 188 of *schoolpress.php* and line 10 of *pages/courses.php*, the word *School* is used.

To create a Spanish-language translation of your plugin, you would then copy the *schoolpress.pot* file to *schoolpress-es\_ES.po* and fill in the *msgstr* for each phrase. It would look like:

```
#: schoolpress.php:108
#: schoolpress.php:188
#: pages/courses.php:10
```

```
msgid "School"  
msgstr "Escuela"
```

Those *.po* files must then be compiled into the *.mo* format, which is optimized for processing the translations.

For large plugins and apps, it is impractical to locate the line numbers for each string by hand and keep that up to date every time you update the plugin. In the next section, we'll walk you through using the `xgettext` command-line tool for Linux to generate your *.pot* file and the `msgfmt` command-line tool to compile *.po* files into *.mo* files. Alternatively, the free program `Poedit` has a nice GUI to scan code and generate *.pot*, *.po*, and *.mo* files and is available for Windows, macOS, and Linux.

## Our File Structure for Localization

Before getting into the specifics of how to generate these files, let's go over how we typically store these files in our plugins. For our SchoolPress app, we store the localization files in a folder called *languages* inside the main app plugin. We add all our localization code, including the call to `load_plugin_textdomain()`, in a file in the *includes* directory called *localization.php*.

So our file structure looks something like this:

1. *../plugins/schoolpress/schoolpress.php* (includes *localization.php*)
2. *../plugins/schoolpress/includes/localization.php* (loads text domain and other localization functions)

3. *../plugins/schoolpress/languages/schoolpress.pot* (list of strings to translate)
4. *../plugins/schoolpress/languages/schoolpress.po* (default/English translations)
5. *../plugins/schoolpress/languages/schoolpress.mo* (compiled default/English translations)
6. *../plugins/schoolpress/languages/schoolpress-es\_ES.po* (Spanish/Spain translations)
7. *../plugins/schoolpress/languages/schoolpress-es\_ES.mo* (compiled Spanish/Spain translations)

When you're building a larger app with multiple custom plugins and a custom theme, localization is easier to manage if you localize each individual plugin and theme separately instead of trying to build one translation file to work across everything. If your plugins will only be used for this one project, they can probably be built as `includes` or module `.php` files in your main app plugin. If you might use the plugins on another project, they need to be localized separately so the localization files can be ported along with the plugin.

## Generating a `.pot` File

We'll use the `xgettext` tool, which is installed on most Linux systems,<sup>1</sup> to generate a `.pot` file for our plugin.

To generate a `.pot` file for our SchoolPress app, we would open up the command line and `cd` to the main app plugin directory at `wp-content/plugins/schoolpress`. Then execute the following command:

```
xgettext -o languages/schoolpress.pot \
--default-domain=schoolpress \
--language=PHP \
--keyword=_ \
--keyword=__ \
--keyword=_e \
--keyword=_ex \
--keyword=_x \
--keyword=_n \
--sort-by-file \
--copyright-holder="SchoolPress" \
--package-name=schoolpress \
--package-version=1.0 \
--msgid-bugs-address="info@schoolpress.me" \
--directory=. \
$(find . -name "*.php")
```

Let's break this down:

*--o languages/schoolpress.pot*

Defines where the output file will go.

*--default-domain=schoolpress*

Defines the text domain as schoolpress.

*--language=PHP*

Tells xgettext that we are using PHP.

*--keyword=...*

Sets xgettext up to retrieve any string used within these functions.

Be sure to include a similar parameter for any of the other translation functions (like `esc_attr__`) you might be using.

*--sort-by-file*

Helps organize the output by file when possible.

*--copyright-holder="SchoolPress"*

Sets the copyright holder stated in the header of the *.pot* file. This should be whatever person or organization owns the copyright to the application, plugin, or theme being built.

## NOTE

From the [GNU.org](#) website:

*Translators are expected to transfer or disclaim the copyright for their translations, so that package maintainers can distribute them without legal risk. If [the copyright holder value] is empty, the output files are marked as being in the public domain; in this case, the translators are expected to disclaim their copyright, again so that package maintainers can distribute them without legal risk.*

`--package-name=schoolpress`

Sets the package name stated in the header of the *.pot* file. This is typically the same as the domain.

`--package-version=1.0`

Sets the package version stated in the header of the *.pot* file. This should be updated with every release version of your app, plugin, or theme.

`--msgid-bugs-address="info@schoolpress.me"`

Sets the email stated in the header of the *.pot* file to use to report any bugs in the *.pot* file.

`--directory=.`

Tells xgettext to start scanning from the current directory.

`$ (find . -name "*.php")`

This appears at the end, and is a Linux command to find all *.php* files under the current directory.

## **Creating a .po File**

Again, the Poedit tool has a nice graphical interface for generating *.po* files from *.pot* files and providing a translation for each string. Hacking it yourself is fairly straightforward though: simply copy the *.pot* file to a *.po* file (e.g., `es_ES.po`) in your languages directory and then edit the *.po* file and enter your translations on each `msgstr` line of the file.

## **Creating a .mo File**

Once your *.po* files are updated for your locale, they need to be compiled into *.mo* files. The `msgfmt` program for Linux can be used to generate the *.mo* files using the command `msgfmt es_ES.po --output-file es_ES.mo`.

## **GlotPress**

GlotPress is a tool to allow translators to collaborate on translations online. Instead of managing individual *.po* files for each locale, GlotPress provides a website UI for editing any string for any locale. Translations are stored in a database, and when a defined percentage of strings is translated for the plugin, GlotPress automatically generates the *.po* and *.mo* files. You don't even need to download the generated language packs or bundle them with your plugin. WordPress will find and download them automatically based on your chosen locale.

## Using GlotPress for Your WordPress.org Plugins and Themes

If you are hosting your plugin or theme in the WordPress.org repository, using GlotPress is as easy as properly wrapping your strings for translation and making sure that your text domain matches your plugin or theme's slug. If you did this properly, a new translation project is generated for you at one of the following locations:

- *<https://translate.wordpress.org/projects/wp-plugins<your-plugins-slug>/>*
- *<https://translate.wordpress.org/projects/wp-themes<your-themes-slug>/>*

That's it. If you've already bundled existing *.po* files for your plugin in a */languages/* folder, they should be imported into the GlotPress project as locales with those strings already translated.

Once your plugin or theme is integrated with GlotPress, you no longer need to bundle language files with your project. WordPress looks for language packs based on your locale and automatically downloads them for use. However, note that only locales that are 95% or more translated will be distributed automatically. So make sure you stay on top of the translations to make them complete. Alternatively, you can download *.po* and *.mo* files for incomplete translations and bundle them with your plugin per the instructions earlier in this chapter.

## Creating Your Own GlotPress Server

If you host your plugin or theme on your own server, you can still take advantage of the GlotPress technology by setting up your own translation server. The GlotPress team has created a WordPress plugin that you can install and activate on any WordPress site. [The GlotPress Plugin](#) will set up an endpoint at `/glotpress/` where you can create new translation projects for your own self-hosted plugins and themes.

Once your GlotPress server is set up, you can create a new project for your non-WordPress.org plugin. WordPress won't download those translations automatically for your users, but you can export the `.po` and `.mo` files to include in your plugin files.

### NOTE

It seems like it would be possible to use the [`translations\_api`](#) hook to update WordPress to look at your GlotPress server to download available translations, but we don't see indications that anyone is doing that. It could be an interesting exercise for a motivated reader of this book. Bundling the translation files with your distributed plugin seems like a more straightforward way to do things.

Depending on the use case of your web application, translating your app may be essential to its success. When building a custom theme or plugin, it's good practice to write all of your code with localization in mind!

---

<sup>1</sup> If not, locate and install the `gettext` package for your Linux distribution.

# Chapter 14. WordPress Optimization and Scaling

---

This chapter is all about squeezing the most performance possible out of WordPress through optimal server configuration, caching, and clever programming.

WordPress often gets knocked for not scaling as well as other PHP frameworks or other programming languages. The idea that WordPress doesn't work at scale mostly comes from the fact that WordPress has traditionally been used to run small blogs on shared hosting accounts. Decisions are made by the WordPress core team (including supporting deprecated functionality and older versions of PHP and MySQL) to make sure that WordPress will boot up easily on as many hosting setups as possible, including underpowered shared hosting accounts.

So there are a lot of really slow WordPress sites out in the wild that help to give the impression that WordPress itself is slow. However, *WordPress is pretty darn fast on the right setup and can be scaled using the same techniques any PHP/MySQL-based app can use.* We will cover many of those techniques in this chapter, introducing you to a number of tools and concepts that can be applied to your own WordPress apps.

# Terms

In this chapter, and throughout this book, we'll throw around terms like “optimization” and “scaling.” It's important to understand exactly what we mean by these terms:

## *Optimization*

This generally refers to getting your app and scripts to run as fast as possible. In some cases, we will be optimizing for memory use or something other than speed. But for the most part, when we say “optimize,” we are talking about making things fast.

## *Scaling*

This term means building an app that can handle *more and more stuff*. More page views. More visits. More users. More posts. More files. More subsites. More computations.

Scaling can also mean building an app to handle *bigger stuff*. Bigger pages. Bigger posts. Bigger files.

The truth is that sometimes an app or specific parts of an app will run fine under light use or when database tables are smaller, for example. But once the number of users and objects being worked on gets larger in number or size, the performance of the app falls off or locks up completely.

*Scalability* is a subjective measure of how well your code and application handles more and bigger stuff. Generally, you want to build your app to handle the amount of growth you expect and then some more just in case. On the other hand, you want to always weigh the pros and cons of any platform or coding decision made for the sake of scalability. These decisions usually come at a cost, both in

money and also in technical debt or added complexity to your codebase. Also, some techniques that make handling many, big transactions as fast as possible actually slow things down when you're working with fewer, smaller transactions. So it's always important to make sure that you are building your app toward your real-world expectations and aren't programming for scalability for the sake of it.

Scaling and optimization are closely related because applications that are fast scale better. There is more to scaling than having fast components, but fast components will make scaling easier. And having a slow application can make scaling harder. For this reason, it always makes sense to optimize your application from the inside out. In “[The Truth About WordPress Performance](#)”, a great whitepaper by Copyblogger Media and W3 Edge, the authors refer to optimizing the “origin” versus optimizing the “edge”:

### *Origin*

Refers to your WordPress application, which is the source of all of the data coming out of your app. Optimizing the origin involves making your WordPress app and the server it runs on faster.

### *Edge*

Refers to services outside of your WordPress application, which are further from the origin but potentially closer to your users. These services include content delivery networks (CDNs) as well as things like browser caching. Optimizing the edge involves using these services in a smart way to make the end user experience better.

## Origin Versus Edge

Again, we advocate optimizing from the inside out, or from the origin to the edge. Improvements in the core WordPress performance will always trickle through the edge to the end user. On the other hand, performance increases based on outside services, while improving the user experience, will sometimes hide bigger issues in the origin that need to be addressed.

A typical example to illustrate this point is when a proxy server such as Varnish (covered in more detail later in this chapter) is used to speed up load times on a site that is loading slowly . Varnish will make a copy of your fully rendered WordPress pages. If a visitor then requests a page that is available in the Varnish cache, that copy will be served to the visitor rather than there being a new one generated through WordPress.

Serving flat files is much faster than running dynamic PHP code, and so Varnish can greatly speed up your website. A page that takes 10 seconds to load on a slow WordPress setup might load in 1 second using Varnish when a copy is fetched. However, 10-second load times are unacceptable, and they are still going to happen. The first time each page is loaded, it will take 10 seconds. Page loads in your admin dashboard are going to take 10 seconds. If a page copy is cleaned out of the Varnish cache for any reason, either because it has been updated or because Varnish needed to make room, it is going to take 10 seconds to load a fresh copy of that page.

Varnish and tools like it are great at what they do and can be a valuable part of your application platform. At the same time, you want to make sure that these edge services aren't hiding issues in your origin.

### NOTE

Be careful: much of the online advice about speeding up a website is meant for a static website. We're talking about building web apps here, with interactive components. At the same time, a good caching system relieves pressure on your web servers, freeing up CPU and memory that could improve noncached page performance.

## Testing

For this chapter, part of our definition of performance will be tied to how fast certain pages load in the web browser. We will be using a few different tools to test page loads for a single user and also for many simultaneous concurrent users.

For all of the tests in this chapter, we used a fresh install of WordPress, running the Twenty Thirteen theme and no other plugins. When not otherwise specified, the server was running a minimal setup with only Apache, MySQL, and PHP installed. The site was hosted on a dedicated server running CENTOS 6 with the following specs:

- Intel® Xeon® E3-1220 processor
- 4 Cores x 3.1 GHz

- 12 GB DDR3 ECC RAM
- 2 TB SATA hard drives in software RAID

## What to Test

Before getting into *how to test*, let's spend a little bit of time thinking about *what to test*. The testing tools described next primarily work by pointing your browser or another tool at a specific URL or a group of URLs for testing. But how do you choose which URLs to test?

The easy answer is to test everything, but that's not very helpful. Just as important as knowing which pages to test is knowing why those pages should be tested and what you are looking for. So here are a few things to think about when testing your app's pages for performance:

### *Test a “static” page to use as a benchmark*

By static here, we don't mean a static *.html* file. The page should be one generated by WordPress, but choose one, like your “about” page or contact form, that has few moving parts. The results for page load on your more static pages will represent a sort of best-case scenario for how fast you can get pages to load on your app. If static pages are loading slowly, fix that first before moving on to more complicated pages.

### *Test your pages with all outside page caches and accelerators turned off*

You first want to make sure that your core WordPress app is running well before testing your entire platform including CDNs, reverse proxies, and any other accelerators you are using to speed up the end user experience. If you send 100 concurrent

connections a page with a full-page cache setup, the first page load might take 10 seconds, and the following 99 may take 1 second. Your average load time will be 1.09 seconds! However, as we discussed earlier, that first 10-second load time is really unacceptable and hints at larger problems with your setup.

#### *Test your pages with all outside page caches and accelerators turned on*

Turning off the outside accelerators will help you locate issues with your core app. However, you want to run tests with the services on as well. This will help you locate issues with those services. Sometimes they will *slow down* your app.

#### *Test prototypical pages*

Whichever *kind* of page your users are most likely to be interacting with is the kind you will want to test. If your app revolves around a custom post type, make sure that the CPT pages perform well. If your app revolves around a search of some kind, test the search form and search results pages.

#### *Test atypical pages*

While you should spend the most time focusing on the common uses of your app, it is a good idea to test the atypical or longtail uses of your app as well, especially if you have some reason to expect a performance issue there.

#### *Test URLs in groups*

Some of the following tools (like Siege and Blitz.io) allow you to specify a list of URLs. By including a list of all of the different types of pages your users will interact with, you get a better idea of what kind of traffic your site can handle. If you expect (or know from analytics) that 80% of your site traffic is on static pages and 20% is on your search pages, you can build a list of URLs with eight static pages and two search results pages, which

will simulate that same 80/20 split during testing. If the test shows your site can handle 1,000 visitors per minute this way, it's a pretty good indication that your site will be able to handle 1,000 visitors in a real-world scenario.

#### *Test URLs by themselves*

Testing URLs in groups will make the topline results more realistic, but it will make tracking down certain performance issues harder. If your static pages load in under 1 second, but your search results pages load in 10 seconds, doing the 80/20 split test described would result in an average load time of 2.8 seconds. However, the 10-second load time on the search results page may be unacceptable. If you test a single search result page or a group of similar search results pages, you'll be better able to diagnose and fix performance issues with the search functionality on your site.

#### *Test your app from locations outside your web server*

The command-line tools described next can be run from the same server serving your website. It's a good idea to run the tools from a different server outside that network so you can get a more realistic idea of what your page loads are when including network traffic to and from the server.

#### *Test your app from inside your web server*

It also makes sense to run performance tests from within your web server. This way, you remove any effect the outside network traffic will have on the numbers and can better diagnose performance issues that are happening within your server.

Each preceding example has a good counterexample, which is another way of saying you really do have to test every page of your site under multiple conditions if you want the best chances of finding performance issues. The important point is to have an idea of what

you are trying to test and to try as much as possible to reduce outside influences on the one piece you are focusing on.

## Chrome Debug Bar

The Google Chrome Debug Bar is a popular tool with web developers that can be used to analyze and debug HTML, JavaScript, and CSS on websites. The Network tab also allows you to view all requests to a website, their responses, and the time each request took.

Similar tabs exist in Firefox and in Microsoft Edge's Developers Tools. To test your site's page load time using the Chrome Debug Bar:

1. Open Chrome.
2. Click the Chrome menu and go to Tools → Developer Tools.
3. Click the Network tab of the debug bar that shows in the bottom pane.
4. Navigate to the page you want to test.
5. You will get a report of all of the requests made to the server.
6. Scroll to the bottom to see the total number of requests and final page load time.

Figure 14-1 shows an example of the Chrome Debug Bar running on a website. The final report will look something like the following:

```
19 requests | 35.7KB transferred | 1.42s (load: 1.16s,  
DOMContentLoaded: 1.10s)
```

This line is telling us the number of requests, the total amount of data transferred to and from the server, the final load time, and also the amount of time it took to load the DOM.

A `DOMContentLoaded` action is fired once all of the HTML of a given site has been loaded, but before any images, JavaScript, or CSS may have finished loading. For this reason the “`DOMContentLoaded`” time will be smaller than the total load time reported by the debug bar.

The Chrome Debug Bar is a crude way to test load times. You have to do multiple loads manually and keep track of the load times to get a good average. However, the debug bar does give you useful information about individual file and script load times, which can be used to find bottlenecks in your site images or scripts.

Log In

# Building Web Apps with WordPress

## WORDPRESS AS AN APPLICATION FRAMEWORK

The screenshot shows the Network tab in the Chrome DevTools. The table lists various resources loaded by the page:

Name	Status	Type	Initiator	Size	Time	Waterfall
fc98dc7b6690c7a987a779e4f06e...	200	jpeg	/blog/:141	(memor...	0 ms	
schoolpress_screen.png	200	png	/blog/:328	(memor...	0 ms	
daf29fcc2c89e2c061f2b7c035c6f...	200	png	/blog/:360	(memor...	0 ms	
51gOTZ2xI0L-228x300.jpg	200	jpeg	/blog/:371	(memor...	0 ms	
Macworld-iWorld-2014.png	200	png	(index)	(memor...	0 ms	
BWAWWP.jpg	200	jpeg	(index)	(memor...	0 ms	
collect?v=1&_v=j77&a=15272827...	200	gif	analytics.js:16	244 B	49 ms	
2764.svg	200	svg+xml	wp-emoji-release...	(memor...	0 ms	
g.gif?v=ext&j=1%3A7.5.1&blog=6...	200	gif	e-201927.js:1	97 B	73 ms	

At the bottom, the stats are: 9 / 49 requests | 341 B / 74.0 KB transferred | 262 KB / 1.0 MB resources | Finish: 1.05 s | DOMContentLoaded: 958 ms | Load: 1.12 s

*Figure 14-1. A shot of the Network tab of the Chrome Debug Bar*

When testing page load times with the Chrome Debug Bar, the first time you visit a web page will typically be much slower than subsequent loads. This is because CSS, JavaScript, and images will be cached by the browser. Additional server-side caching may also affect load times. Keep this in mind when testing load times. Unless you are trying to test page loads with caching enabled, you may want to disable caching in your browser and on the server.

You can also use the Audits tab of the Chrome Debug Bar (or the [PageSpeed Tools by Google](#)) to get recommendations for how to make your website faster. [Figure 14-2](#) shows an example of the report generated by the Audits tab. This chapter will cover the main tools and methods for carrying out the kinds of recommendations made by the PageSpeed audit.

Google also now offers modules for Apache and Nginx web servers to automatically detect and perform certain optimizations. Even if you would rather manually implement some of their recommendations, [the docs page for the PageSpeed module](#) is a great collection of website performance best practices.

Log In

Building Web Apps with WordPress

# Building Web Apps with WordPress

## WORDPRESS AS AN APPLICATION FRAMEWORK

https://bwawwp.com

Elements Console Sources Network Performance Memory Application Audits » 2 4 :

+ (new audit) ▾



### Audits

Identify and fix common problems that affect your site's performance, accessibility, and user experience.

[Learn more](#)

Device

Mobile

Desktop

Audits

Performance

*Figure 14-2. The Audit tab of the Chrome Debug Bar*

## The WordPress Site Health Tool

WordPress 5.2 shipped with a new tool to test and report on various security and performance issues. It's a good idea to run this tool on any WordPress site you manage to make sure that all of your various server and WordPress software is up to date and that any required server modules are enabled. Figure 14-3 shows the WordPress Site Health Tool.

The screenshot shows the WordPress Site Health tool interface. On the left, a dark sidebar lists various site management options: Dashboard, Jetpack, Posts, Media, Pages, Comments (with 273 notifications), Feedback, Memberships, Appearance, Plugins, Users, Tools (which is selected), Available Tools, Import, Export, Site Health (which is also selected), and Delete Site. The main content area is titled "Site Health Status". It displays a "Status" section with a progress bar at 85% and a yellow circular icon. Below this, it says "The site health check shows critical information about your WordPress configuration and items that require your attention." It lists "2 Critical issues": "Your PHP version requires an update" (Security) and "Background updates are not working as expected" (Security). Under "Available Tools", it shows "1 Recommended improvement": "One or more recommended modules are missing" (Performance).

Site Health 85%

Status Info

Site Health Status

The site health check shows critical information about your WordPress configuration and items that require your attention.

2 Critical issues

Your PHP version requires an update

Background updates are not working as expected

1 Recommended improvement

One or more recommended modules are missing

Figure 14-3. The WordPress Site Health Tool

## Apache Bench

Using your web browser, you can get an idea of load times for a single user under whatever load your server happens to be under at the time of testing. To learn how well your server will respond under constant heavy load, you need to use a benchmarking tool like Apache Bench.

Despite the name, Apache Bench can be used to test other HTTP servers besides Apache. What it basically does is spawn the specified number of dummy connections against a website and records the average load times along with other information.

### INSTALLING APACHE BENCH

Apache Bench is available for all Linux distributions. On CENTOS and RedHat servers, you can install it via the `httpd-tool` package. If you have the `yum` package manager installed, you can use this command:

```
yum install httpd-tool
```

On Ubuntu servers, Apache Bench will be part of the `apache2-util` package. If you have `apt-get` installed, you can use this command:

```
apt-get install apache2-util
```

Apache Bench is also available for Windows and should have been installed alongside your Apache installation. Information on how to

install and run Apache Bench on Windows can be found in the [Apache docs](#).

## RUNNING APACHE BENCH

The full list of parameters and options can be found in the main file or on the [Apache website](#). The command to run Apache Bench is `ab`, and a typical command will look like this:

```
ab -n 1000 -c 100 http://yourdomain.com/index.php
```

The two main parameters for the `ab` command are `n` and `c`. `n` is the number of requests, and `c` is the number of concurrent requests to perform at one time. In the last example, 1,000 total requests will be made in batches of 100 simultaneous requests at a time.

### NOTE

If you leave off the trailing slash on your domain or don't specify a `.php` file to load, Apache Bench may fail with an error.

The output will look something like this (the report shows the results for 100 concurrent requests against the home page of a default WordPress install running on our test server).

```
#ab -n 1000 -c 100 http://yourdomain.com/index.php
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
http://www.zeustech.net/
Licensed to The Apache Software Foundation,
http://www.apache.org/

Benchmarking yourdomain.com (be patient)
```

```

Server Software: Apache/2.2.15
Server Hostname: yourdomain.com
Server Port: 80

Document Path: /
Document Length: 251 bytes

Concurrency Level: 100
Time taken for tests: 8.167 seconds
Complete requests: 1000
Failed requests: 993
    (Connect: 0, Receive: 0, Length: 993, Exceptions: 0)
Write errors: 0
Non-2xx responses: 7
Total transferred: 9738397 bytes
HTML transferred: 9516683 bytes
Requests per second: 122.44 [#/sec] (mean)
Time per request: 816.740 [ms] (mean)
Time per request: 8.167 [ms] (mean, across all
concurrent requests)
Transfer rate: 1164.40 [Kbytes/sec] received

Connection Times (ms)
              min   mean[+/-sd] median   max
Connect:        0     0.4      0       2
Processing:     3    799 127.4     826    1164
Waiting:        2    714 113.7     729    1091
Total:          3    799 127.4     826    1164

Percentage of the requests served within a certain time (ms)
 50%    826
 66%    854
 75%    867
 80%    876
 90%    904
 95%    936
 98%    968
 99%    987
100%   1164 (longest request)

```

The main stat to track here is the first “Time per request.” In the test, the mean is shown as 816.740 milliseconds (ms), or about 0.817 seconds. What this number means<sup>1</sup> is that when there are 100 people

hitting the site at the same time, it takes about 0.817 seconds for the server to originate the HTML for the home page.

There is a second “Time per request” stat under the first labeled “mean, across all concurrent requests.” This is simply the mean divided by the number of concurrent connections. In the example it shows 8.167 ms, or about 1/100 second. Note that the second “Time per request” number is *not* the load time for a single request. However, across multiple tests, this ratio (average request time / number of concurrent requests) does give you insight into how well your server handles larger numbers of concurrent users. If the mean across concurrent requests stays the same as you increase `-c`, that means that your server is scaling well. If it goes up drastically, your server is not scaling well.

Another important stat in this report is “requests per second.” This number sometimes maps more directly to your load estimates. You can get real-life “requests per day” or “requests per hour” numbers from your site stats and convert these to “requests per second” and compare that to your report numbers. Then you can tweak the `n` and `c` inputs to match your desired conditions.

## TESTING WITH APACHE BENCH

Following are a few tips that will help you when testing a website with Apache Bench:

1. Run Apache Bench from somewhere other than the server you are testing, since Apache Bench itself will be using resources required for your web server to run. Running your

benchmarks from outside locations will also give you a more realistic idea of page generation times, including network transfer times.

On the other hand, running Apache Bench from the same server the site is hosted on removes the network latency from the equation and gives you a window into the performance of your stack irrespective of the greater internet.

2. Start with a small number of simultaneous connections and build up to larger numbers. If you test 100,000 simultaneous connections right out of the gate, you can fry your web server or your testing server, or both. Start with 100 connections, then 200, then 500, then 1,000, adding more as you go. Large errors or bottlenecks in your server and app performance can appear with as few as 100 connections. Once you pass those tests, try throwing more connections at the app.
3. Run multiple tests. Many factors will affect the results of your benchmarks. No two tests will be exactly the same, so run a few tests on different pages of your site, at different times, under different conditions, and from different servers and geographical locations. This will give you more realistic results.

## **GRAPHING APACHE BENCH RESULTS WITH GNUPLOT**

The `-g` parameter of Apache Bench can be used to specify an output file of the result data in the gnuplot format. This data file can be fed into a gnuplot script to generate a graph image.

## NOTE

You can also use the `-e` parameter to specify an output file in CSV (Excel) format.

You can run the following commands to set up some space for your testing and save the gnuplot data:

```
# mkdir benchmarks
# mkdir benchmarks/data
# mkdir benchmarks/graphs
# ab -n 5000 -c 200 -g benchmarks/data/testing.tsv
"http://yourdomain.com/"
```

The summary report for this benchmark includes:

```
Requests per second:      95.00 [#/sec]  (mean)
Time per request:        2105.187 [ms]  (mean)
```

Then you'll need to install gnuplot.<sup>2</sup> Once it's installed, you can create a couple of gnuplot scripts to generate your graphs. Here are some scripts modified from the examples in [a blog post by Brad Landers](#). Put these in your `/benchmark/` folder.

This first graph will draw a line chart showing the distribution of load times. This chart is good at showing how many of your requests loaded under certain times. You can save this script as `plot1.gp`:

```
# Let's output to a png file
set terminal png size 1024,768
# This sets the aspect ratio of the graph
set size 1, 1
# The file we'll write to
set output "graphs/sequence.png"
# The graph title
```

```

set title "Benchmark testing"
# Where to place the legend/key
set key left top
# Draw gridlines oriented on the y-axis
set grid y
# Label the x-axis
set xlabel 'requests'
# Label the y-axis
set ylabel "response time (ms)"
# Tell gnuplot to use tabs as the delimiter instead of
# spaces (default)
set datafile separator '\t'
# Plot the data
plot "data/testing.tsv" every ::2 using 5 title 'response
time' with lines
exit

```

This second graph will draw a scatterplot showing the request times, as well as the distribution of load times, throughout the tests. You can save this script as *plot2.gp*:

```

# Let's output to a png file
set terminal png size 1024,768
# This sets the aspect ratio of the graph
set size 1, 1
# The file we'll write to
set output "graphs/timeseries.png"
# The graph title
set title "Benchmark testing"
# Where to place the legend/key
set key left top
# Draw gridlines oriented on the y axis
set grid y
# Specify that the x-series data is time data
set xdata time
# Specify the *input* format of the time data
set timefmt "%s"
# Specify the *output* format for the x-axis tick labels
set format x "%S"
# Label the x-axis
set xlabel 'seconds'
# Label the y-axis
set ylabel "response time (ms)"
# Tell gnuplot to use tabs as the delimiter instead of
# spaces (default)

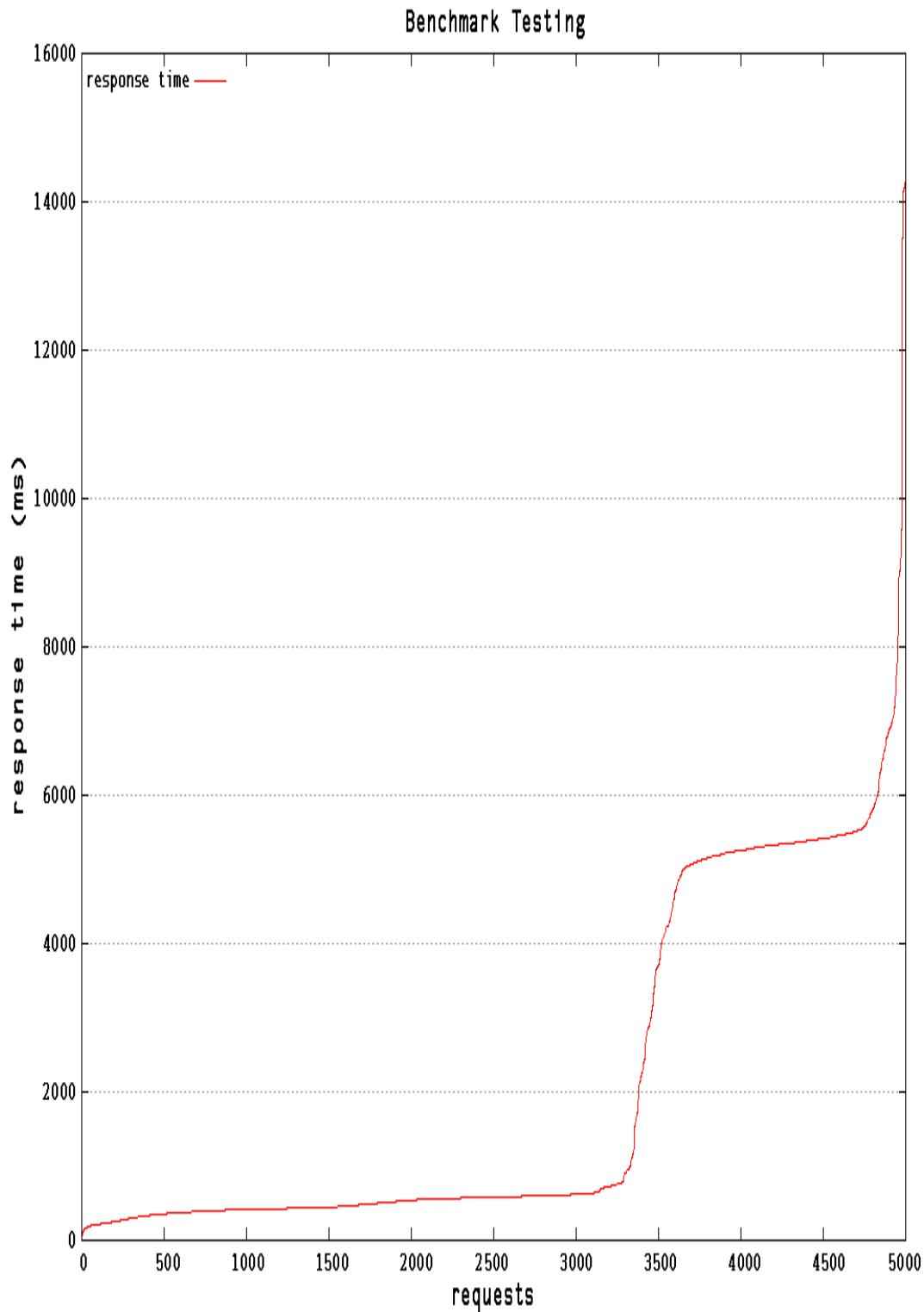
```

```
set datafile separator '\t'  
# Plot the data  
plot "data/testing.tsv" every ::2 using 2:5 title 'response  
time' with points  
exit
```

To turn your benchmark data into graphs, then, run these commands:

```
# cd benchmark  
# gnuplot plot1.gp  
# gnuplot plot2.gp
```

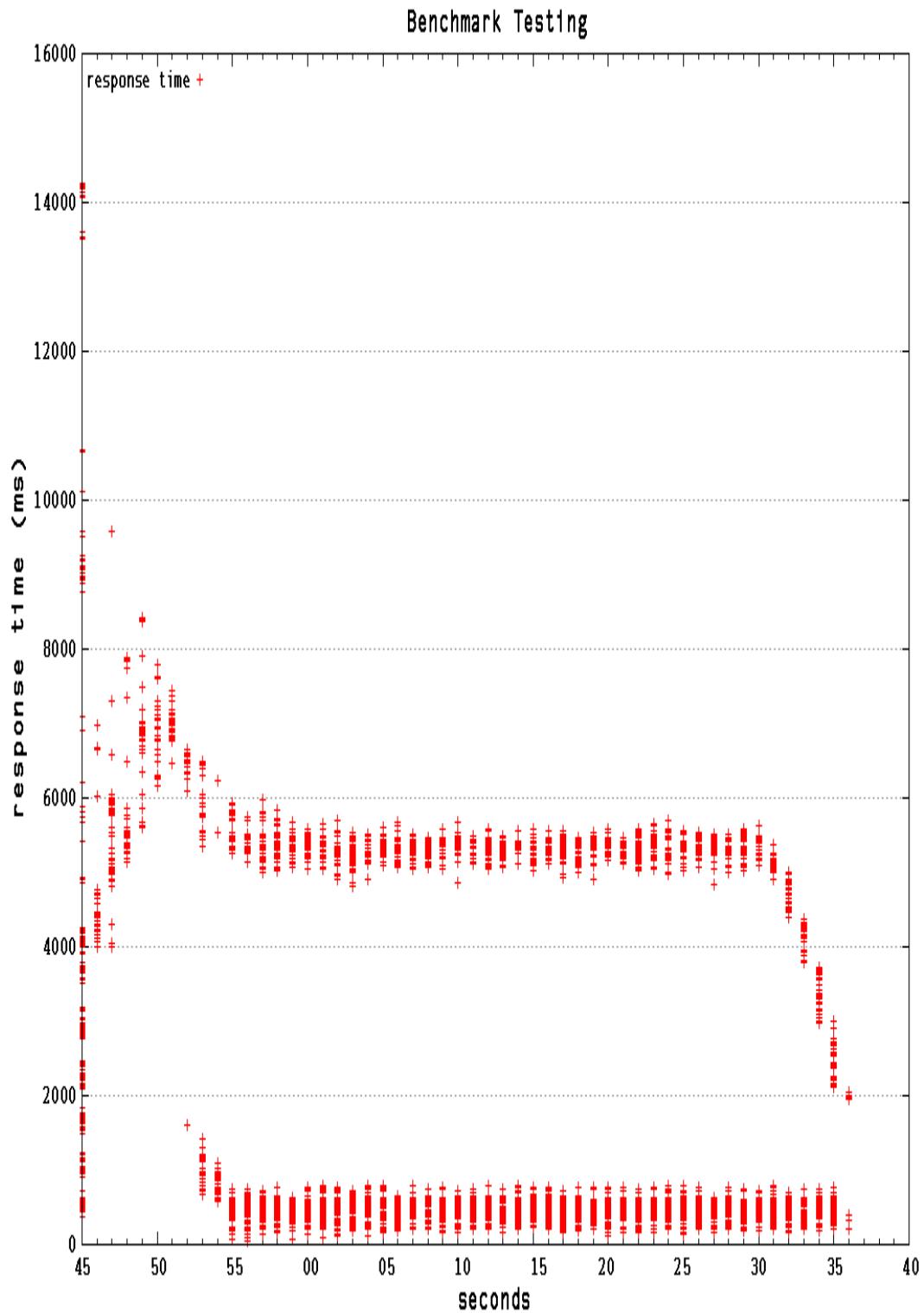
The resulting charts should look like Figures 14-4 and 14-5.



*Figure 14-4. The output of the plot1.gp gnuplot script*

Seeing the data in graphical form can help a lot. For example, while the summary showed a mean load time of 2,105 ms, the graphs here show us that just over half of our requests were processed in under 1 second, and the remaining requests took over 4.5 seconds.

You might think that a 2-second load time is acceptable, but a 4-second load time is not. Based on the summary report, you'd think you were in the clear, when really something like 30%+ of your users would be experiencing load times over 4 seconds.



*Figure 14-5. The output of the plot2.gp gnuplot script*

**Siege**

Siege is a tool that, like Apache Benchmark, will hit your site with multiple simultaneous connections and then record response times. The report generated by Siege does a good job of showing just the most interesting information.

Siege will need to be installed from the source: you can get the latest source files at the [Joe Dog software site](#).

A sample Siege command will look like this:

```
siege -b -c100 -d20 -t2M http://yourdomain.com
```

The `-b` parameter tells Siege to run a benchmark. The `-c100` parameter says to use 100 concurrent users. The `-d20` parameter sets the average sleep time between page loads for each user to 20 seconds. And the `-t2M` parameter says to run the benchmark for two minutes. You can also use `-t30S` to set a time in seconds or `-t1H` to set a time in hours.

The output will look like this:

```
** Preparing 100 concurrent users for battle.  
The server is now under siege...  
Lifting the server siege...      done.  
  
Transactions:          1160 hits  
Availability:         100.00 %  
Elapsed time:          119.29 secs  
Data transferred:     9.53 MB  
Response time:        0.11 secs  
Transaction rate:    9.72 trans/sec  
Throughput:           0.08 MB/sec  
Concurrency:          1.05  
Successful transactions: 1160  
Failed transactions:   0
```

Longest transaction:	0.26
Shortest transaction:	0.09

The server was hit 1,160 times by 100 users, with an average response time of 0.11 seconds. The server was up 100% of the time, and the longest response time was just 0.26 seconds.

## W3 Total Cache

There are a few plugins for WordPress that will help you set up various tools to increase the performance of a WordPress site. One plugin in particular, W3 Total Cache, offers just about every performance-increasing method out there.

Frederick Townes, founder of Mashable and the lead developer of W3 Total Cache, shares our belief that WordPress optimization should be done as close to the core WordPress app (the origin) as possible:

*Mileage varies, but one thing we know for certain is that user experience sits right next to content in terms of importance—they go hand in hand. In order for a site or app to actually reach its potential, it's critical that the stack, app and browser are all working in harmony. For WordPress, I try to make that easier than it was in the past with W3 Total Cache.*

For many sites with low traffic or little dynamic content, setting up the most common settings in W3 Total Cache is all you will need to scale your app. For other sites, you may want to implement some of the methods bundled with W3 Total Cache individually so you can customize them to your specific app. In general, W3 Total Cache

does a great job of making sure that all of the bundled techniques play nice together. For this reason, it's a good idea to work *with* W3 Total Cache to customize things rather than use a solution outside of the plugin that could conflict with it. We'll go over a typical configuration for W3 Total Cache, and also describe briefly how some of the techniques work in general.

W3 Total Cache is available for download from the WordPress.org plugin repository. Once the plugin is installed, a Performance menu item will be added to the admin dashboard. You'll usually have to update permissions on various folders and files on your WordPress install to allow W3 Total Cache to work. The plugin will give you very specific messages to get things set up. Once this is done, you're ready to start enabling the various tools bundled with the plugin.

### NOTE

The W3 Total Cache plugin is available in its entirety for free through the WordPress plugin repository. You will need to purchase a plan through a CDN to take advantage of W3 Total Cache's CDN features. And finally, the makers of W3 Total Cache offer various support and configuration services through their website.

To enable the tools we want to use, go to the General Settings page of the W3 Total Cache Performance Menu. Find the Enable checkbox for the Page Cache, Minify, Database Cache, Object Cache, and Browser Cache sections; check the box; and then click “Save All Settings” (see [Figure 14-6](#)).

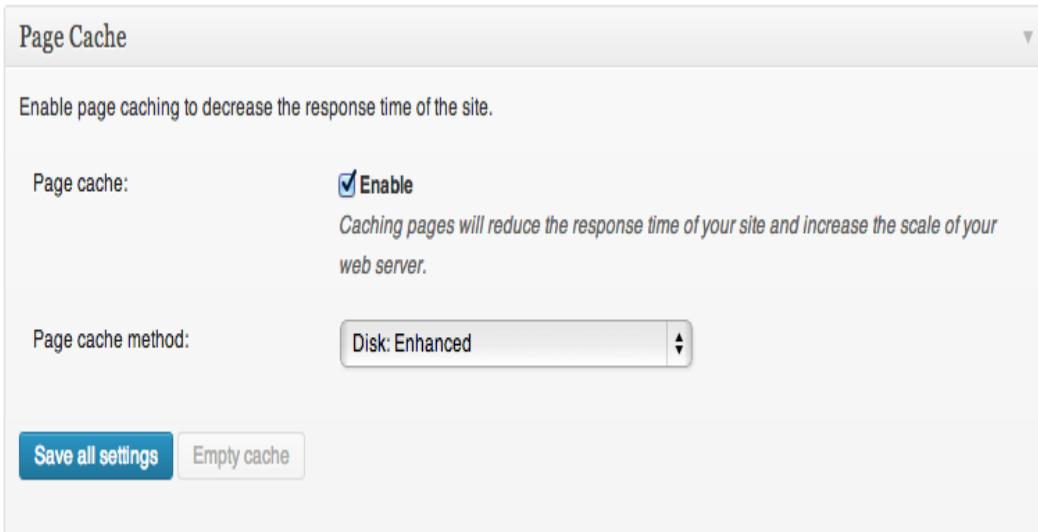


Figure 14-6. Check *Enable* in the box for each performance technique you want to use

You can typically get by using the default and recommended settings for all of the W3 Total Cache tools. Your exact settings will depend on the specifics of your app, your hosting setup, and how users use your app. There are a lot of settings, and we won't go over all of them, but we'll cover a few important ones in the following sections.

## Page Cache Settings

A page cache is exactly what it sounds like: the caching of entire web pages after they've been generated. When a new user visits your site, if a cache of the page is available, that static HTML file is served instead of the page being loaded through PHP and WordPress. If there is no cache or the cache has expired, the page is loaded as it normally would through WordPress.

For a web server like Nginx or Apache, serving a static HTML file is much faster than serving a dynamic PHP file. Serving static files avoids all of the database calls and calculations that are required in

your dynamic PHP scripts, but it also plays to the strengths of your web stack, which is architected from the OS level up to the web server level primarily to push files around quickly.

Every visit that is served a static HTML file instead of generating a dynamic page in PHP is going to save you some RAM and CPU time. With more resources available, even noncached or noncacheable pages are going to load faster. So page caching can greatly speed up page loads on your site, and is one of the primary focuses of web hosts and others trying to serve WordPress sites quickly.

On the Page Cache page under the Performance menu, you'll usually want to enable the following options in the General box: Cache front page, Cache feeds, Cache SSL (https) requests, Cache 404 (not found) pages, and “Don’t cache pages for logged in users.” See Figure 14-7 for an example.

The screenshot shows the 'General' settings page of the W3 Total Cache plugin. On the left is a sidebar with icons and links: Plugins, Users, Tools, Settings (selected), and Performance. Under Performance, there are links for Dashboard, General Settings (selected), Page Cache, Minify, Database Cache, Object Cache, Browser Cache, User Agent Groups, Referrer Groups, CDN, Monitoring, Extensions, FAQ, Support, Install, and About. At the bottom of the sidebar is a 'Collapse menu' button. The main content area has a title 'General'. It contains several configuration options with checkboxes:

- Cache front page  
*For many blogs this is your most visited page, it is recommended that you cache it.*
- Cache feeds: site, categories, tags, comments  
*Even if using a feed proxy service (like [FeedBurner](#)), enabling this option is still recommended.*
- Cache SSL (https) requests  
*Cache [SSL](#) requests (uniquely) for improved performance.*
- Cache URLs with query string variables  
*Search result (and similar) pages will be cached if enabled.*
- Cache 404 (not found) pages  
*Reduce server load by caching 404 pages. If the disk enhanced method of disk caching is used, 404 pages will be returned with a 200 response code. Use at your own risk.*
- Cache requests only for www.payrascal.com site address  
*Cache only requests with the same [URL](#) as the site's [site address](#).*
- Don't cache pages for logged in users  
*Unauthenticated users may view a cached version of the last authenticated user's view of a given page. Disabling this option is not recommended.*
- Don't cache pages for following user roles  
*Select user roles that should not receive cached pages:*  
 Administrator    Editor    Author    Contributor    Subscriber

**Save all settings**

Figure 14-7. W3 Total Cache page cache general settings

The “Don’t cache pages for logged in users” checkbox is an important option to check because logged-in users will often have access to private account information, and you don’t want that stuff getting into the cache. At the very least, you might accidentally show a cached “Howdy, Jason” in the upper right of your website for users who aren’t Jason. In the worst-case scenario, you might share Jason’s personal email address or account numbers.

For these reasons, full-page caching is typically going to cause problems for logged-in members. Other types of caching can still help speed up page load speeds for logged-in members, and we'll cover a few methods later in this chapter.

Another important option to get familiar with is the ability to exclude certain pages, paths, and URLs from the page cache. Inside of the Advanced box is a text area labeled "Never cache the following pages." This text area is shown in Figure 14-8. Pages, paths, and URLs added to this setting are going to be ignored by the page cache, and so will be generated fresh on every page load. Place one URL string per line, and regular expressions are allowed.

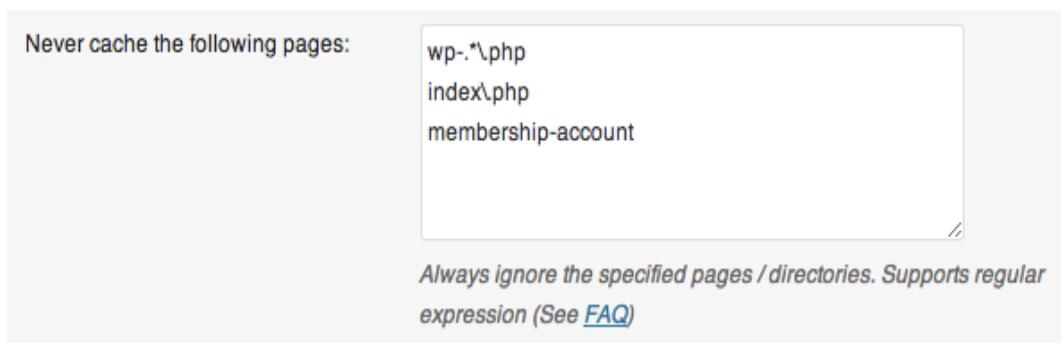


Figure 14-8. "Never cache the following pages" section in W3 Total Cache Page settings

Some common pages that you will want to exclude from the page cache include checkout pages, login pages, non-JavaScript-based contact forms, API URLs, and any other pages that should be generated dynamically on each load.

## Minify

Minifying is the process of both combining and removing excess whitespace and unnecessary characters from source files, typically

JavaScript and CSS files, to reduce the file size of those files when served to the web browser. Smaller file sizes means faster load times.

You are probably familiar with files like *jquery.min.js*, which is a minimized version of the jQuery library. W3 Total Cache will automatically minify all of your CSS and JS files for you. You can also enable the minification of HTML files (most notably from the page cache), which can save a bit on page loads as well.

In general, minification is a good idea on production sites. On development sites, you will want to leave minification off so you can better debug CSS and scripting issues.

## Database Caching

W3 Total Cache offers database caching. This will store the results of SELECT queries inside of a cache file (or in a memory backend).

Repeated calls to the same database query will pull results from the cache instead of querying the database, which may be on an entirely different service that is part of the slowdown.

If your database server is running on solid state drives (SSDs) or has some kind of caching enabled at the MySQL layer, database caching with W3 Total Cache may not improve performance and can negatively impact it, relatively speaking. So be sure to run benchmarks before and after enabling database caching to see if it helps your site and analyze your slow query log to identify queries that can be manually tuned. Remember, caching scales servers; it

doesn't magically resolve fundamentally slow-performing queries or code.

If you find that specific queries are taking a long time, you can cache them individually using WP transients or other fragment caching techniques, which are covered later in this chapter.

## Object Cache

Object caching is similar to database caching, but the objects' PHP representations are stored in the cache instead of the raw MySQL results. Like database caching, object caching can sometimes slow your site down instead of speeding it up; your mileage may vary. Using a persistent object cache (covered later in this chapter) will make it more likely that the object cache will speed up your site. Be sure to benchmark your site before and after configuring object caching.

Object caching is also known to cause issues with some WordPress plugins or activities in WordPress. Object caching is a powerful tool for speeding up your site, but you may have to spend time tweaking the lower levels of the scripts you use to make them work with the plugins and application code for your specific app.

## CDNs

A content delivery network, or CDN, is a service that can serve static files for you—typically images, JavaScript files, and CSS files—on one or many colocated servers that are optimized for serving static files. So instead of being loaded off the same server that is generating

the PHP pages for your site, your images will be loaded from whichever CDN server is closest to you. Even if you use your own server as a CDN, you can decrease load times because the browser will be able to load the static files and PHP page at the same time, as a separate browser connection will be used for both.

W3 Total Cache can help you integrate with many of the most popular CDNs. The plugin will handle uploading all of your media files, static script files, and page cache files to the CDN; automatically redirect URLs on your site to the CDN; and, most importantly, purge modified files for those CDNs that support it.

## GZIP Compression

GZIP compression is another neat trick that will often speed up your site. In effect, you trade processing time (when the files are zipped up) for download time (since the files will be smaller). The browser will unzip the files on the receiving end. The time saved by downloading smaller files usually makes up for the time spent zipping and unzipping them. Of course, when using W3 Total Cache, the compression happens once when the cache is built.

But again, like everything else, run a benchmark before and after enabling GZIP compression to make sure that your site is benefiting from the feature.

## Hosting

Upgrading your hosting is one of the best things you can do to improve performance for your WordPress app. More CPU and RAM will speed up PHP, MySQL, and your underlying web server. This may sound obvious, but many people can get caught up in the excitement of optimizing code or using caching techniques to speed up part of their web app while ignoring a simple hosting upgrade that will improve performance across the board.

Of course, we advocate using all of the techniques in this chapter if applicable and within budget. However, one of the earliest decisions you are likely to make, possibly before you even start coding, is where you are going to host your finished web app.

You can find our specific recommendations for hosting WordPress apps on this book's [website](#). In this section, we'll cover the different types of hosting to consider.

## WordPress-Specific Hosts

As WordPress has become more popular for building websites, hosting companies have cropped up that are configured specifically for running WordPress sites. The earliest of these were Page.ly, Zippykid, WP Engine, and SiteGround.

The WordPress-specific hosts offer managed environments with server-side caching and a support staff that is more knowledgeable about WordPress than a typical hosting company.

The control panels for these hosts are similar to shared hosting plans, with limited flexibility in adjusting the underlying configuration. On

the plus side, these hosts typically handle a lot of your caching setup, do a great job managing spam and denial-of-service (DoS) attacks, and can quickly scale your app as load increases. On the downside, the limited configurability can be an issue with certain apps and plans can get pricey for larger sites.

## **Rolling Your Own Server**

The alternative to WordPress-specific managed hosting is to roll your own server, either on dedicated hardware or in a cloud environment.

On the dedicated side, Rackspace is a popular choice, and 1and1 provides powerful hardware at wholesale prices. On the cloud side, Amazon EC2 is very popular, and DigitalOcean is a cost-effective alternative.

No matter which route you go, you will have to set up your own web server, install PHP and MySQL yourself, and manage all of the DNS and other server maintenance yourself. Depending on your needs and situation, this could be a good thing or a bad thing. If you need a more specific configuration for your app, you have to roll your own server. On the other hand, you'll have to spend time or money on server administration that might be better spent elsewhere.

It's important to know where your limits are in terms of server administration. For example, Jason is very experienced setting up web servers like Apache and configuring and maintaining PHP and MySQL. On the other hand, he has little experience managing a firewall against DoS attacks or load-balancing across multiple

servers. You'll want to choose a hosting company and option that works to your strengths and makes up for your weaknesses.

Rolling your own server and getting 10 times the raw performance for 1/10 the cost of a shared hosting plan can feel good. But when you find yourself up at 3 a.m., wasting time struggling to keep your server alive against automated hacking attempts from foreign countries, the monthly managed options fee may not seem so steep.

### **WARNING**

Best practices for setting up and running web servers and the various caching tools that speed them up are changing all the time. Also, instructions will depend on your particular server, which version of Linux it is running, which other tools you are using, and the specifics of the app itself. The proper way to use the information in the rest of this chapter is to go over the instructions provided here and in the linked to articles to get an idea of how the technique being covered works. If you decide to implement the technique on your own server, do some research (Google) to find a tutorial or instructions that are up to date and more specific to your situation.

Next, we'll go quickly over a few common setups for Linux-based servers running WordPress. Details on how to set up each individual configuration are constantly evolving. We will try to always have links to the most recent instructions and reviews online at the book's website.

## **APACHE SERVER SETUP**

As the most popular web server software in use today, it is usually fairly painless to install Apache on any flavor of Linux server.

Once set up, there are a few things you can do to optimize the performance of Apache for your WordPress app:

- Disable unnecessary modules loaded by default.
- Set up Apache to use prefork or worker multiprocessing, depending on your need. A good overview of each option can be found in the blog post "[Understanding Apache 2 MPM \(worker vs prefork\)](#)". Prefork (the default) is usually best for running WordPress.
- If using the Apache Prefork Multi-Processing Module (default), configure the `StartServers`, `MinSpareServers`, `MaxSpareServers`, `ServerLimit`, `MaxClients`, and `MaxRequestsPerChild` values.
- If using the worker Multi-Processing Module, configure the `StartServers`, `MaxClients`, `MinSpareThreads`, `MaxSpareThreads`, `ThreadsPerChild`, and `MaxRequestsPerChild` values.

There are a couple of settings in particular to pay attention to when optimizing Apache for your hardware and the app it's running. These settings typically have counterparts in other web servers as well. The concepts behind them should be applicable to any web server running WordPress.

The following settings and instructions assume you are using the more common prefork module for Apache:

*MaxClients*

When Apache is processing a request to serve a file or PHP script, the server creates a child process to handle that request. The MaxClients setting in your Apache configuration tells Apache the maximum number of child processes to create.

After reaching the MaxClients number, Apache will queue up other incoming requests. So if MaxClients is set too low, your visitors will experience long load times as they wait for Apache to even start to process their requests.

If MaxClients is set too high, Apache will use up all of your RAM and start to use swap memory, which is stored on the hard drive and much, much slower than physical RAM. When this happens, your visitors will experience long load times since their requests will be handled using the slower swap memory.

Besides simply being slower, swap memory also requires more CPU as the memory is swapped from hard disks to RAM and back again, which can lead to lower performance overall. When your server backs up like this, it's called thrashing and can quickly spiral out of control and eventually will lock up your server.

So it's important to pick a good value for your MaxClients setting. To determine an appropriate value for MaxClients for your Apache server, take the amount of server memory you want to dedicate to Apache (typically as much as possible after you subtract the amount that MySQL and any other services on your server use) and then divide that by the average memory footprint of your Apache processes.

There is no exact way to figure out how much memory your services are using or how much memory each Apache process takes. It's best to start conservatively and to tweak the values as you watch in real time.

Using the command `top -M` we can see the total memory on our server, how much is free, and how much active processes are currently using. On our test server, which is under no load, you can see that we have 11.7 GB of memory and 10.25 GB of that free. If we want to do a 50/50 split between Apache and MySQL (another assumption you should test out and adjust to your specific app), we can dedicate about 4.5 GB to Apache, dedicate 4.5 GB to MySQL, and leave the rest (up to 2.7 GB in this case) for padding and other services running on the server.

Figure 14-9 shows an example of the output from running the `top` command. To figure out how much memory Apache needs for each process, use `top -M` again, when the server is under normal loads. Look for processes running the `httpd` command. If we see our app using about 20 MB of memory for each process, we would divide 4.5 GB (~4600 MB) by 20 MB and get 230, meaning our server should be able to support 230 `MaxClients` in 4.5 GB of memory.

When setting the `MaxClients` value, set the `ServerLimit` value to the same number. `ServerLimit` is a kind of `MaxClients` that can only be changed when Apache is restarted. The `MaxClients` setting can be changed by other scripts while Apache is running, although this isn't commonly done. So theoretically `ServerLimit` could be set higher than `MaxClients` and some process could increase or decrease the `MaxClients` value while Apache is running.

```

top - 19:16:27 up 88 days, 4:00, 2 users, load average: 2.97, 0.68, 0.22
Tasks: 149 total, 9 running, 140 sleeping, 0 stopped, 0 zombie
Cpu(s): 96.2%us, 3.5%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 11.731G total, 1652.238M used, 10.117G free, 179.586M buffers
Swap: 4095.984M total, 0.000k used, 4095.984M free, 999.453M cached

PID USER      PR  NI    VIRT   RES   SHR S %CPU %MEM     TIME+ COMMAND
29098 apache    20   0 285m 28m 17m R 26.0  0.2  0:04.08 httpd
28891 apache    20   0 285m 27m 17m R 25.3  0.2  0:04.44 httpd
29304 apache    20   0 285m 28m 17m R 25.3  0.2  0:03.77 httpd
15956 apache    20   0 289m 42m 29m S 25.0  0.4  0:05.69 httpd
15957 apache    20   0 297m 50m 35m S 25.0  0.4  0:06.05 httpd
15959 apache    20   0 288m 42m 31m R 25.0  0.4  0:06.16 httpd
29303 apache    20   0 286m 28m 17m S 25.0  0.2  0:03.76 httpd
15962 apache    20   0 287m 39m 29m R 24.3  0.3  0:05.82 httpd
28036 apache    20   0 317m 67m 34m S 24.3  0.6  0:04.91 httpd
29305 apache    20   0 286m 28m 17m S 24.3  0.2  0:03.75 httpd
29301 apache    20   0 285m 28m 17m S 24.0  0.2  0:03.64 httpd
15963 apache    20   0 289m 37m 26m R 23.6  0.3  0:05.56 httpd
15965 apache    20   0 307m 54m 23m S 23.6  0.5  0:05.46 httpd
29097 apache    20   0 286m 28m 17m R 23.6  0.2  0:04.12 httpd
15954 apache    20   0 299m 57m 35m S 23.0  0.5  0:05.78 httpd
15960 apache    20   0 298m 47m 32m R 22.6  0.4  0:05.54 httpd
28799 root      20   0 772m 7624 940 S  5.3  0.1  0:00.90 siege

```

*Figure 14-9. Using the top command to figure out how much memory is available*

### *MaxRequestsPerChild*

Each child process or client spun up by Apache will handle multiple requests one after another. If

`MaxRequestsPerChild` is set to 0, these child processes are never shut down, which is good since it lowers the overhead of spinning up a new child process, but can be bad if there is a memory leak in your app. Setting `MaxRequestsPerChild` to a very high number like 1,000 or 2,000 is a nice compromise so that new processes aren't shut down and restarted too often, but if a memory leak does occur it will be cleaned up when the child process is eventually shut down.

### *KeepAlive*

By default, the `KeepAlive` setting of Apache is turned off, meaning that after serving a file to a client browser, the connection is closed. A separate connection is opened and closed for each file request from that browser. Since a single page may have several files associated with it (in the form of images, JavaScript, and CSS), this can lead to a lot of unnecessary opening and closing of connections.

With `KeepAlive` turned on, Apache will keep the first connection from a web browser open and serve all subsequent requests from the same browser session through that connection. After sitting idle with no requests from the same browser session, Apache will close the connection. Using a single connection instead of many can lead to great performance gains for some sites, especially if there are a lot of images or separate CSS and JavaScript on each page (you should probably be minimizing your CSS and JavaScript into one file for each anyway).

On the other hand, turning `KeepAlive` on requires more RAM since each connection will hold on to the memory for each request as it keeps a connection open.

It's useful to experiment with turning `KeepAlive` on. If you do, you should change the `KeepAliveTimeout` value from the default 15 seconds to something smaller like 2 or 3 seconds—or something closer to the real load times of a single page visit on your site. This will free up the memory faster.

Also, if you turn `KeepAlive` on, you should probably adjust the `MaxClients` and `MaxRequestsPerChild` settings. Since each child process will be using more memory as it keeps the connection open, you may need to decrease your `MaxClients` value to avoid running out of memory. And since each connection counts as one request with respect to `MaxRequestsPerChild`, you may want to decrease your

`MaxRequestsPerChild` value since there will be fewer requests overall per visit.

Some other good articles on optimizing Apache include:

- “[Apache Performance Tuning](#)”
- “[Apache MPM Prefork](#)”
- “[Apache MPM Worker](#)”
- “[Optimize Apache for WordPress](#)” by Drew Strojny
- “[Apache Optimization: KeepAlive On or Off?](#)” by Abdussamad

## NGINX SERVER SETUP

A popular alternative to Apache that is gaining a lot of momentum right now is Nginx. The main advantage of Nginx is that it is an *asynchronous* web server, whereas Apache is a *process-based* web server. What this means in practice is that when many simultaneous clients hit an Apache-based server, a new thread is created for each connection. With Nginx, all connections are handled by a single thread or a small group of threads. As each thread requires a block of memory, Nginx is more memory efficient and so can process a higher number of simultaneous requests than Apache.

Some good articles about installing and configuring Nginx include:

- “[Nginx](#),” an article from the WordPress Codex
- “[How to Install WordPress with nginx on Ubuntu 12.04](#)” by Etel Sverdlov

## NGINX IN FRONT OF APACHE

The trade-off in using Nginx over Apache is that Nginx has fewer module extensions than Apache. Some modules like `mod_rewrite` for “pretty permalinks” will have to be ported over to the Nginx way of doing things. Other modules may not have Nginx equivalents.

For this reason, it is becoming popular to set up a dual web server configuration where Nginx serves cached web pages and static content and Apache serves dynamically generated content. One article explaining how to configure this setup is [“How to Configure Nginx as a Front End Proxy for Apache”](#) by Etel Sverdlov.

The main advantage of this setup is that static files will be served from Nginx, which is configured to serve static files quickly; this will ease the memory burden of Apache. If you are already using a CDN for your static files, then using Nginx for static files would be redundant. Also, because you are still serving PHP files through Apache, you won’t gain the memory benefits of Nginx on dynamically generated pages. For these reasons, it is probably better to use Nginx for both static files and PHP or Apache with a CDN for static files.

## MYSQL OPTIMIZATION

To get the best performance out of WordPress, you will want to make sure that you’ve configured MySQL properly for your hardware and site use and that you’ve optimized the database queries in your app.

*Optimizing MySQL configuration*

The MySQL configuration file is typically found at `/etc/my.cnf` or `/etc/mysql/my.cnf` and can be tweaked to improve performance on your site. There are several interrelated settings. The best way to figure out a good configuration for your hardware and site is to use the MySQLTuner Perl script.

After downloading the MySQLTuner script, you will also need to have Perl installed on your server. Then run `perl mysqltuner.pl` and follow the recommendations given. The output will look like the following:

```
----- General Statistics -----
-----
[--] Skipped version check for MySQLTuner script
[OK] Currently running supported MySQL version 5.5.32
[OK] Operating on 64-bit architecture

----- Storage Engine Statistics -----
-----
[--] Status: +Archive -BDB -Federated +InnoDB -ISAM -
NDBCluster
[--] Data in MyISAM tables: 35M (Tables: 395)
[--] Data in InnoDB tables: 16M (Tables: 316)
[--] Data in PERFORMANCE_SCHEMA tables: 0B (Tables: 17)
[!!] Total fragmented tables: 327

----- Security Recommendations -----
-----
[OK] All database users have passwords assigned

----- Performance Metrics -----
-----
[--] Up for: 26d 22h 6m 21s (8M q [3.755 qps], 393K
conn, TX: 15B, RX:
1B)
[--] Reads / Writes: 95% / 5%
[--] Total buffers: 168.0M global + 2.8M per thread (151
max threads)
[OK] Maximum possible memory usage: 583.2M (7% of
installed RAM)
[OK] Slow queries: 0% (0/8M)
[OK] Highest usage of available connections: 21%
(33/151)
[OK] Key buffer size / total MyISAM indexes: 8.0M/21.1M
```

```
[OK] Key buffer hit rate: 100.0% (84M cached / 40K reads)
[!!] Query cache is disabled
[OK] Sorts requiring temporary tables: 0% (3 temp sorts / 1M sorts)
[!!] Joins performed without indexes: 23544
[!!] Temporary tables created on disk: 26% (359K on disk / 1M total)
[!!] Thread cache is disabled
[OK] Table cache hit rate: 34% (400 open / 1K opened)
[OK] Open file limit used: 68% (697/1K)
[OK] Table locks acquired immediately: 99% (8M immediate / 8M locks)
[OK] InnoDB data size / buffer pool: 16.1M/128.0M

----- Recommendations -----
-----
General recommendations:
    Run OPTIMIZE TABLE to defragment tables for better performance
        Enable the slow query log to troubleshoot bad queries
            Adjust your join queries to always utilize indexes
                When making adjustments, make
                    tmp_table_size/max_heap_table_size
                        equal
                    Reduce your SELECT DISTINCT queries without LIMIT clauses
                        Set thread_cache_size to 4 as a starting value
Variables to adjust:
    query_cache_size (>= 8M)
        join_buffer_size (> 128.0K, or always use indexes with joins)
            tmp_table_size (> 16M)
            max_heap_table_size (> 16M)
            thread_cache_size (start at 4)
```

One thing to note is that MySQLTuner will give better recommendations if it has at least one day's worth of log data to process. For this reason, it should be run 24 hours after MySQL has been restarted. You'll want to follow the recommendations given, wait 24 hours, and run the script again, then rinse and repeat over a few days to narrow in on optimal settings for your MySQL setup.

## *Optimizing database queries*

A large source of load-time-draining process cycles is unoptimized, unnecessary, or otherwise slow MySQL queries. Finding and optimizing these slow SQL queries will speed up your site. Caching database queries, either at the database level or through the use of transients for specific queries, will help with slow queries, but you definitely want the original SQL as optimized as possible.

The first step in optimizing your database queries is to find out which queries are slow or otherwise undeeded. A great tool to do this is the [Query Monitor plugin by John Blackbourn](#).

Query Monitor, shown in Figure 14-10, adds a bar to the bottom of your website that shows you a page's load time in milliseconds, the number of SQL queries made and how long they took, PHP errors and warnings, and other useful information.

# Building Web Apps with WordPress

## WORDPRESS AS AN APPLICATION FRAMEWORK

Query Monitor

Overview

	Page Generation Time	Peak Memory Usage	Database Query Time
PHP Errors			
Queries	2.5588 8.5% of 30s limit	20,066 kB 15.3% of 131,072 kB limit	0.0971
└ Duplicate Queries (17)			
└ Queries by Caller	Database Queries	Object Cache	
└ Queries by Component	SELECT: 125 SHOW: 2 UPDATE: 2 (: 20 DELETE: 10 INSERT: 10 Total: 169	91.0% hit rate (4,530 hits, 450 misses) <span style="color: red;">! External object cache not in use</span>	Opcode cache in use: Zend OPcache
Request			
└ Hooks in Use			
Template			

<https://bwawwp.com>

*Figure 14-10. The Query Monitor added to the bottom of all pages on your site while active*

If you click on the Queries link in the Query Monitor bar, you will also see all of the SQL queries made and the individual query times.

### NOTE

Viewing the final generated SQL query is especially useful for queries that might be constructed across several PHP functions or with a lot of branching logic. For example, the final query to load posts on the blog home page of WordPress is generated using many variables stored in the `$wp_query` object depending on if a search is being made, what page of the archive you are on, and other factors.

With Query Monitor turned on, you can browse around your site looking for queries that are slow or outright unnecessary.

Another way to find slow SQL queries is to enable slow query logging in your MySQL configuration file. This will help find slow queries that come up in real use. You don't want to rely on the slow query log, but it can catch some real-world use cases that won't come up in testing.

To enable slow query logging in MySQL, find your `my.cnf` or `my.ini` file and add the following lines:

```
slow-query-log = 1;  
slow-query-log-file = /path/to/a/log/file;
```

After updating your MySQL configuration file, you will need to restart MySQL.

When trying to optimize your database queries, always be on the lookout for:

- Cases where the same SQL query is being run more than once per page load. Store the result in a global variable or somewhere else to access it later in the page load.
- Cases where one SQL query can be used instead of many queries. For example, plugins can load all their options at once instead of using a separate query or a `getOption()` call for each option.
- Cases where a SQL query is being run, but the result is not being used. Some queries may only need to be run in the dashboard or only on the frontend or only on a specific page. Change the WordPress hook being used or add PHP logic around these calls so they are only executed when needed.

If you find a necessary query that is taking a particularly long time, how you go about optimizing it will be very specific to the query itself. Here are some things to try:

- Adjust queries to use only indexed columns in WHERE, ON, ORDER BY, and GROUP BY clauses.
- Add WHERE clauses to your JOINS so you are joining smaller subtables.
- Use a different table to store your data—for example, using taxonomies versus post meta (covered in Chapter 5).
- Add indexes to columns that are to be used in WHERE, ON, ORDER BY, and GROUP BY clauses.

## ADVANCED-CACHE.PHP AND OBJECT-CACHE.PHP

The keystones<sup>3</sup> that enable all of these caching techniques, including the ones used by the W3 Total Cache plugin, are the *advanced-cache.php* and/or *object-cache.php* files, which can be added to the */wp-content/* directory.

To tell WordPress to check for the *advanced-cache.php* and *object-cache.php* files, add the line `define ('WP_CACHE', true);` to your *wp-config.php* file.

The *advanced-cache.php* file is loaded by *wp-settings.php* before the majority of the WordPress source files are loaded. Because of this, you can execute certain code (e.g., to look for a cache file on the server) and then stop PHP execution with an `exit;` command before the rest of WordPress loads.

If a *object-cache.php* file is present, it will be used to define the WP Cache API functions instead of the built-in functions found in *wp-includes/cache.php*. By default, WordPress will cache all options in an array during each page load. Transients are stored in the database. If you write your own *object-cache.php* file, you can tell WordPress to store options and transients in a RAM-based memory that is persisted between page loads.

Plugins like W3 Total Cache are mostly a frontend for generating an *advanced-cache.php* file based on the settings you choose. You can also choose to roll your own *advanced-cache.php* or *object-cache.php* file or use one configured already for a specific caching tool or technique. Most of the caching techniques that follow involve

using a specific *advanced-cache.php* or *object-cache.php* file to interact with another service for caching.

If you add a header comment to the top of your *.php* files dropped into the *wp-content* directory with the same structure as a plugin (plugin name, description, etc.), then that information will show up on the Drop-ins tab of the plugins page of the WordPress dashboard.

## **ALTERNATIVE PHP CACHE (APC)**

Alternative PHP Cache is an extension for PHP that acts as an opcode cache and can be used to store key-value pairs for object caching.

### *Opcode caching*

When a PHP script is executed, it is compiled to opcodes that are ready to be executed by the server. With an opcode cache, part of the compiling is cached until the underlying PHP scripts are updated.

### *Key-value cache*

APC also adds the `apc_store()` and `apc_fetch()` functions, which can be used to store and retrieve bits of information from memory. A value stored in memory can typically be loaded faster than a value stored on a hard disk or in a database, especially if that value requires some computation. Plugins like W3 Total Cache or APC Object Cache Backend can be used to store the WordPress object cache inside of RAM using APC.

These are the rough steps to set up APC:

1. Install APC on your server, configure PHP to use it, and restart your web server.
2. Configure WordPress to use APC using W3 Total Cache, APC Object Cache Backend, or another plugin or custom *object-cache.php* script.

These links have good information on using APC in general and with WordPress:

- “Alternative PHP Cache” at [PHP.net](#)
- “How to Install Alternative PHP Cache (APC) on a Cloud Server Running Ubuntu 12.04” by [Danny Sipos](#)

#### NOTE

PHP versions 5.5 and higher come compiled with OPCache, an alternative to APC for opcode caching. However, OPCache does not have the same store and fetch functionality that APC has for object caching. For this reason, either disable OPCache and use APC or run an updated version of APC called APCu alongside OPCache. APCu offers the store and fetch functionality but leaves the opcode caching to OPCache.

## MEMCACHED

Memcached is a system that allows you to store key-value pairs in RAM that can be used as a backend for an object cache in WordPress. Memcached is similar to APC, minus the opcode caching.

You can store your full-page caches inside of Memcached instead of files on the server for faster load times, although the performance

gain will be slower for modern servers with faster solid-state drives. Memcached can be run on both Apache- and Nginx-based servers.

One of the advantages of Memcached over other object caching techniques (including Redis and APC) is that a Memcached cache can be distributed over multiple servers. So if you have multiple servers hosting your app, they can all use the one Memcached instance to store a common cache instead of having their own (often redundant) cache stores on each server. Interesting note: the enterprise version of W3 Total Cache allows you to use APC across multiple servers seamlessly.

These are the rough steps to set up Memcached:

1. Install the Memcached service on your server, give it some memory, and run it.
2. Use W3 Total Cache or the [Memcached Object Cache plugin](#) to update the WordPress object cache to use Memcached.

The following links have information on using Memcached in general and with WordPress:

- “[Memcached](#)” at PHP.net
- [The Memcached website](#)
- “[WordPress + Memcached](#)” by Scott Taylor

## REDIS

Redis is another system for storing key-value pairs in memory on your Apache- or Nginx-based web server. Like Memcached, it can be

used as a backend for your WordPress object cache or page cache.

Unlike Memcached, Redis can store data in lists, sets, and sorted sets in addition to simple key-value hashes. These data structures are always useful for your apps, and the maturity of Memcached, which was created a few years before Redis, is appreciated by some developers.

These are the rough steps to set up Redis:

1. Install Redis on your server, give it some memory, and run it.
2. Use a replacement for the WordPress *index.php* that searches the Redis cache and serves pages from there if found. A popular version is wp-redis-cache.
3. Run a plugin or other script to clear the Redis cache on, for example, post updates.

These links have good information on using Redis in general and with WordPress:

- [The Redis website](#)
- [WP-Redis-Cache](#)
- [“WordPress with Redis as a Frontend Cache” by Jim Westergren](#)

## VARNISH

Varnish is a reverse proxy that can sit in front of your Apache or Nginx setup and serve cached versions of complete web pages to your visitors. Because your web server and PHP are never even

loaded for cached pages, Varnish will outperform Memcached and Redis for full-page caching. On the other hand, Varnish is not meant to do object caching and so will only work for static pages on your site.

These are the rough steps for setting up Varnish with WordPress:

1. Install Varnish on your server.
2. Configure Varnish to ignore the dashboard at `/wp-admin/` and other sections of your site that shouldn't be cached.
3. Use a plugin to purge the Varnish cache when posts are updated and other updates are done to WordPress. Some popular plugins to do this are [WP-Varnish](#) and [Varnish HTTP Purge](#).

The following links have good information on using Varnish in general and with WordPress:

- [The Varnish website](#)
- [“How to Install and Customize Varnish for WordPress” by Austin Gunter](#)
- [Varnish 3.0 Configuration Templates](#)

## BATCACHE

Batcache uses APC or Memcached as a backend for full-page caching in WordPress. The end result should be similar to using W3 Total Cache or another plugin integrated with APC or Memcached for full-page caching.

One thing unique to Batcache is that the caching is only enabled if a page has been loaded two times within 120 seconds. A cache is then generated and used for the next 300 seconds. These values can be tweaked to fit your purposes, but the basic idea here is that Batcache is meant primarily as a defense against traffic spikes like those that happen when a website is “slashdotted,” “techcrunched,” “reddited,” or linked to by any of the other websites large enough to warrant its own verb. Another benefit to caching only pages under heavy load is that a lower amount of RAM is required to store the cache. If you tweak the default settings, you can set up Batcache to work as an always-on full-page caching system.

These are the rough steps for setting up Batcache with WordPress:

1. Set up Memcached or APC to be used as Batcache’s in-memory key-value store.
2. Download the Batcache plugin from the WordPress repository.
3. Move the *advanced-cache.php* file to the *wp-content* folder of your WordPress install.

Batcache has an interesting pedigree since it was developed specifically for WordPress. It was first used on WordPress VIP sites and WordPress.com. Batcache was originally called Supercache, but the popular caching plugin WP Super Cache was released around the same time and the Supercache/Batcache authors changed the name from one famous DC Comics caped crusader to another. The following links have good information on using Batcache with WordPress:

- [The Batcache plugin](#)
- [“WordPress Caching using APC and Batcache” by Jonathan D. Johnson](#)
- [Original Batcache announcement and overview by Andy Skelton](#)

## Selective Caching

The caching methods described so far have either been full-page caches or otherwise “dumb” caches storing every WordPress object in cache. Rules could then be added to tell the cache to avoid certain URLs or conditions, but basically you were caching all of the things.

Sometimes you will want to do things the other way around. You’ll want to cache specific pages and objects. This is typically done by storing information within a WordPress *transient*. If you have a persistent object enabled like APC, that stored object will load that much faster.

### NOTE

What we’re calling *selective caching* here is commonly referred to as *fragment caching*. No matter the term, the concept is the same: caching parts of a rendered web page instead of the full page.

For example, you might have a full-page cache enabled through W3 Total Cache or Varnish, but you’ll need to exclude logged-in users from seeing the cache because member-specific information could get

cached. Mary could end up seeing “Welcome, Bob” in the upper right of the page. Still, some portion of each page might be the same for each user or certain kinds of users. We can selectively cache that information if it takes excessive database calls or computation to compile.

Good candidates for selective caching include reports, complicated post queries, and other bits of content that require a lot of time or memory to compute.

## The Transient API

Transients are the preferred way for WordPress apps to set and get values out of the object cache. If no persistent caching system is installed, the transients are stored inside of the `wp_options` table of the WordPress database. If an object caching system like APC, APCu, Memcached, or Redis is installed, then that system is used to store the transients.

At any time, the server could be rebooted or the object cache memory cleared, wiping out your stored transients. For this reason, when storing transients, always assume that the storage is temporary and unreliable. If the information to be stored needs to be saved, you can still redundantly store it inside a transient for performance reasons, but make sure you also store it in another way—most likely by saving an option through `update_option()`.

In the SchoolPress app, we need to get an average homework score across all assignments a single student has submitted. The query for

this would involve running the average function against the meta\_value column of the wp\_usermeta table. Running an average for one student with, say, 10 to 100 assignments wouldn't be too intense; however, if you had a page showing the average score for 20 to 80 students within one class, that series of computations might take a while to run. To speed this up, we can cache the results of the full class report within a transient, as illustrated in [Example 14-1](#).

This is a perfect use case for using transients because having access to the computed results inside of a transient will speed up repeated loads of the report, but it's OK if the transient suddenly disappears because we can always compute the averages from scratch.

#### *Example 14-1. SPClass*

---

```
class SPClass()
{
    /* ... constructor and other methods ... */

    function getStudents()
    {
        /* gets all users within the BuddyPress group
for this class */

        return $this->students; //array of student
objects
    }

    function getAssignmentAverages()
    {
        //check for transient
        $this->assignment_averages =
            get_transient('class_assignment_averages_'.
$this->ID);

        //no transient found? compute the averages
        if(empty($this->assignment_averages))
    }
}
```

```

    {

        $this->assignment_averages = array();
        $this->getStudents();

        foreach ($this->students as $student)
        {
            $this-
>assignment_averages[$student->ID] =
                $student->getAssignmentAverages();
        }

        //save in transient

        set_transient('class_assignment_averages_'.
                      $this->ID, $this->assignment_averages);
    }

    //return the averages
    return $this->assignment_averages;
}

//clear assignment averages transients when an assignment is
graded
public function
clear_assignment_averages_transient($assignment_id)
{
    //class id is stored as postmeta on the assignment post
    $assignment = new Assignment($assignment_id);
    $class_id = $assignment->class_id;

    //clear any assignment averages transient for this
    class
    delete_transient('class_assignment_averages_'.
    $class_id);
}
add_action('sp_update_assignment_score', array('SPClass',
    'clear_assignment_averages_transient'));

```

The example includes a lot of snipped code and makes some assumptions about the SPClass and Student classes. However,

you should get the idea of how this report uses transients to store the computed averages, retrieve them, and clear them out on updates.

In the preceding example, we store the array stored in `$this->assignment_averages` in the transient. Alternatively, we could have stored the generated HTML, but storing the array saves us most of the complex database calls and is more flexible.

The function to store a value inside a transient is

`set_transient( $transient, $value, $expiration )`, and attributes are as follows:

`$transient`

Unique name for the transient, 45 characters or fewer.

`$value`

The value to store. Objects and arrays are automatically serialized and unserialized for you.

`$expiration`

An optional parameter to set an expiration for the transient in seconds. Expired transients are deleted by a WordPress garbage collection script. By default, this value is 0 and doesn't expire until deleted.

Notice that we use a descriptive key

(`class_assignment_averages_`) followed by the ID of the class group. This way, all classes will have their own transient for storing assignment averages.

To retrieve the transient, we simply call `get_transient( $transient )`, passing one parameter with the unique name of the transient. If the transient is available and not past expiration, the value is returned. Otherwise, the call returns `false`.

To delete the transient before expiration, we call `delete_transient( $transient )` passing one parameter with the unique name of the transient. Notice that in the example, we hook into the `sp_update_assignment_score` that is fired when any assignment gets scored. We pass an array as the callback for the hook since the method is part of the `SPClass` class. The `sp_update_assignment_score` hook passes the `$assignment_id` as a parameter. The callback method uses this ID to find the assignment and the associated class ID, then deletes the corresponding `class_assignment_averages_{ID}` transient.

If you were storing transients related to posts or users, you may want to clear them out on `save_post` or `profile_update`, respectively.

#### NOTE

The transient functions are simple wrappers for the functions defined in the default `wp-includes/cache.php` file, or your drop-in `object-cache.php` file: `wp_cache_set()`, `wp_cache_get()`, and `wp_cache_delete()`. If you want, you could call these functions directly. Information on these functions can be found in the [WordPress Codex](#).

## Multisite Transients

In network installs, the transients set with `set_transient()` are specific to the current network site. So our `class_assignment_averages_1` set on one network site won't be available on another network site. (This makes sense in the assignment scores example.)

If you'd like to set a transient network-wide, WordPress offers variants of the transient functions:

- `set_site_transient( $transient, $value, $expiration )`
- `get_site_transient( $transient )`
- `delete_site_transient( $transient )`

These functions work the same as the basic transient functions; however, the `_site_transients` are stored in `wp_site_options` instead of the individual network site `wp_options` tables.

Because the transients set with `set_site_transient()` prefix the string `_site` to the transient name, you only have 40 characters to work with for the name versus the usual 45.

Finally, note that a different set of hooks fires before and after a network-wide transient versus a single network site transient. If you've written code that hooks into `pre_set_transient`, `set_transient`, or `setted_transient`, you may need that

code to also hook into `pre_set_site_transient`, `pre_set_transient`, and `pre_setted_site_transient`.

## Using JavaScript to Increase Performance

A useful tactic for speeding up page loads is to load certain parts of a web page through JavaScript instead of generating the same output through dynamic PHP.

### NOTE

If your app has been built from the ground up as a JavaScript application using the WP REST API and a framework like REACT, this suggestion will be redundant since you're already pulling your web pages and app screens together through many Ajax requests.

This technique can make pages appear to load faster since the frame of the web page can be loaded quickly, while the time-intensive portion of the site can be loaded over time as a “loading...” icon flashes on the screen or a progress bar fills up. Your users will get immediate feedback that the page has loaded along with an indication to sit tight for a few seconds while the page renders.

Using JavaScript can also literally speed up your page loads. If you load all of the dynamic content of a page through JavaScript, you can then use a page cache to serve the rest of the page without hitting PHP.

For example, on many blogs, the only piece of dynamic content is the comments. Using the built-in WordPress comments and a full-page cache means that recent comments won't show up on the site until the cache clears. However, if you use a JavaScript-based commenting system like those provided through the JetPack plugin or a service like Disqus or Facebook, then your comments section is simply a bit of static JavaScript code that loads the dynamic comments from another server.

Example 14-2 shows a bare-bones example of how you can go about loading specific content through JavaScript on an otherwise static page.

---

*Example 14-2. JS Display Name plugin*

---

```
<?php
/*
Plugin Name: JS Display Name
Plugin URI: http://bwawwp.com/js-display-name/
Description: A way to load the display name of a logged-in
user through JS
Version: .1
Author: Jason Coleman
Author URI: http://bwawwp.com
*/
/*
use this function to place the JavaScript in your
theme
if(function_exists("jsdn_show_display_name"))
{
    jsdn_show_display_name();
}
*/
function jsdn_show_display_name($prefix = "Welcome, ")
{
?>
<p>
<script src="<?php echo admin_url(

```

```

        "/admin-ajax.php?
action=jsdn_show_display_name&prefix=" .
        urlencode($prefix)
) ; ?>"></script>
</p>
<?php
}

/*
This function detects the JavaScript call and returns the
user's display name
*/
function jsdn_wp_ajax()
{
    global $current_user;
    if(!empty($current_user->display_name))
    {
        $prefix =
sanitize_text_field($_REQUEST['prefix']);
        $text = $prefix . $current_user->display_name;

        header('Content-Type: text/javascript');
    ?>
        document.write(<?php echo json_encode($text);?>);
    <?php
    }

    exit;
}
add_action('wp_ajax_jsdn_show_display_name', 'jsdn_wp_ajax');
add_action('wp_ajax_nopriv_jsdn_show_display_name',
'jsdn_wp_ajax');

```

## Custom Tables

Another tool you'll definitely need when building WordPress apps in general, but specifically when trying to optimize performance, is to

build a custom database table or view to make certain lookups and queries faster.

With SchoolPress, we may need to do lots of queries on the assignment objects. We'll want to sort by score, class, teacher, student, assignment date, and submission date. Maybe we need to sort by some combination of those. If those values are stored in the `wp_postmeta` table, queries on that data will be slow because 1) the `wp_postmeta` table will be large with other nonassignment posts and post meta, and 2) the `meta_value` column is not indexed.

Indexing the `meta_value` column would be overkill because we would be indexing a lot of post meta that we don't need indexed. Inserts into the `wp_postmeta` table would take forever and use up a lot of memory. Switching some of the post meta over to taxonomies wouldn't make much sense, would be pretty hard to manage, and wouldn't necessarily give us the speed increase that we need.

The following code is a bit contrived, but at some point you will come across a case where it is better for you to store your data in a custom table rather than some combination of posts, post meta, and taxonomies.

Our assignments table might look like the following:

```
CREATE TABLE `wp_sp_assignments` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) unsigned,
  `class_id` bigint(20) unsigned,
  `student_id` bigint(20) unsigned,
```

```

`score` int(10),
`assignment_date` DATETIME,
`due_date` DATETIME,
`submission_date` DATETIME
PRIMARY KEY (`id`),
UNIQUE KEY `post_id` (`post_id`),
KEY `class_id` (`class_id`),
KEY `student_id` (`student_id`),
KEY `score` (`score`),
KEY `assignment_date` (`assignment_date`),
KEY `due_date` (`due_date`),
KEY `submission_date` (`submission_date`)
);

```

This is a rather extreme example; every column has an index. It's probably overkill here, but it does allow us to make extremely fast queries against this table joined with the wp\_posts table or wp\_users table.

If you had a table like this, you would need to hook into the save\_post hook to update the corresponding row in wp\_sp\_assignments like this:

```

function sp_update_assignments_table($post_id)
{
    //get the post
    $post = get_post($post_id);

    //we only care about assignments
    if($post->post_type != "assignment")
        return false;

    //get data ready for insert or replace
    $assignment_data = array(
        "post_id" => $post_id,
        "student_id" => $post->post_author,
        "teacher_id" => $post->teacher_id,
    );
}

```

```

    "score" => $post->score,
    "assignment_date" => $post->assignment_date,
    "due_date" => $post->due_date
    "submission_date" => $post->submission_date
);

//look for an existing assignment
$assignment_id = $wpdb->get_var("SELECT id
                                    FROM wp_sp_assignments
                                    WHERE post_id = '" .
$post_id . "'"
                                    LIMIT 1");

//if no assignment id, this is a new assignment
if(empty($assignment_id))
{
    $assignment_id = $wpdb->insert("wp_sp_assignments",
$assignment_data);
}
else
{
    $assignment_data['id'] = $assignment_id;
    $wpdb->replace("wp_sp_assignments", $assignment_data);
}

return $assignment_id;
}
add_action('save_post', 'sp_update_assignments_table');

```

## Bypassing WordPress

Finally, one last technique to help you scale your WordPress app: you don't have to use WordPress for every part of your app.

We've already gone over a few variants of this advice earlier in this chapter. When you use Varnish, you are bypassing WordPress and loading static HTML files instead. When you use *advanced-*

*cache.php*, you are bypassing part of WordPress. When you use JavaScript to load comments from Facebook, you are bypassing WordPress. When you store some of your data in a custom database table, you are bypassing the WordPress framework.

We use WordPress to build our apps because of the benefits of its security, its functionality, and the large community of plugins and solutions. Having your code written with the WordPress platform makes it easy for you to hook into the WordPress CMS and user management. It makes it easy to add hooks into your own code.

However, sometimes the performance downside will outweigh all of the benefits. You don't have to scrap WordPress altogether, but you can bypass WordPress for specific functions.

For example, our last script to get the display name of a user through JavaScript could be written as a simple PHP script that runs a simple SELECT query to get the `display_name` column of the user specified in the WordPress user cookie. Doing so could save a few milliseconds off of each page load in your website. If you multiply this across several dynamic bits across your app, the savings can add up.

In most cases, cutting out WordPress like this should be a last resort. The speed savings are there but come at the cost of complicating your code. Running scripts like this through the REST API or *admin-ajax.php* like we did cuts out a lot of the overhead of WordPress, but still allows your code to interact with other plugins and use built-in WordPress classes and APIs if it needs to.

If you are loading a script to export CSV, which is going to take 10 seconds to run anyway, it's not as important to cut off that extra 0.5 seconds.

You should always program things as straightforwardly as possible, in this case through traditional WordPress methods, and only optimize at this level when you find a bottleneck worth optimizing. At that point, explore all the options presented in this chapter to determine which works best for the specific feature you are optimizing, considering your needs and the team and tools you have at your disposal.

---

1 Pun intended.

2 Run `yum install gnuplot` on CENTOS/Redhat systems. More information can be found at [the gnuplot home page](#).

3 We suppose there can only be one keystone. You'll have to forgive us this time.

# Chapter 15. Ecommerce

---

At some point, you may want to charge for access to your app or otherwise accept payments on your site. In this chapter, we go over the best ecommerce and membership plugins available and give you a few pointers for choosing among them. We'll also walk through the steps of setting up a typical paywall in the software-as-a-service (SaaS) model.

## Choosing a Plugin

There is one ecommerce plugin for WordPress that is nearly synonymous with ecommerce: WooCommerce. Since the first edition of this book published, back in 2014, WooCommerce has come to dominate not only the WordPress ecommerce platform, but ecommerce in general. We'll give a brief introduction to WooCommerce here and go over a few of the hooks and filters app developers would be interested in.

Despite the fact that WooCommerce is a well made and maintained product, there are cases in which other ecommerce plugins may be more appropriate. We also cover Paid Memberships Pro (a membership-focused ecommerce plugin) and Easy Digital Downloads (a virtual goods-focused ecommerce plugin).

The plugins we cover in this chapter have these features in common:

- Integration with multiple payment gateways
- Secure checkout forms
- Saved order information
- Products (or membership levels) with pricing

We'll point out the unique features of each type of ecommerce plugin in the following sections.

## WooCommerce

It is hard to calculate ecommerce platform market share,<sup>1</sup> but WooCommerce probably accounts for at least 20% of *all* ecommerce websites.<sup>2</sup> WordPress.org says the plugin has been downloaded over 47 million times on more than 4 million active sites. Any way you calculate it, a huge number of sites are using WooCommerce. That's why Automattic purchased WooCommerce back in 2015, and why we'll focus on WooCommerce first.

WooCommerce, and other “shopping cart plugins,” include these features:

- A products custom post type
- The ability to browse products
- The ability to search through products
- The ability to purchase multiple products at once
- Support for shipping address and shipping price calculations
- Support for custom tax rules

Documentation on how to set up WooCommerce and much more can be found on [the WooCommerce site](#).

## THE WOOCOMMERCE PLUGIN AND EXTENSIONS

WooCommerce is available for free in the WordPress.org repository. The core plugin has everything needed to define products and prices and accept payment through PayPal. The Stripe Payment Gateway extension was recently made available for free and is the most popular way to accept credit cards directly on your site for WooCommerce, but extensions are available for just about any payment gateway you would want to use.

WooCommerce has dozens of free and paid extensions to enhance the core plugin, integrate with third-party marketing services, introduce new product types, integrate with shipping services, or improve the store management. Two of its most popular extensions are [WooCommerce Subscriptions](#) (which allows you to collect recurring payments) and [WooCommerce Memberships](#) (which also allows you to collect recurring payments as well as restrict content based on membership levels). The Paid Memberships Pro plugin covered shortly is streamlined for that exact use case, but if you are using WooCommerce already or need another feature handled well by WooCommerce, those extensions are a good way to go.

## CUSTOMIZING WOOCOMMERCE THROUGH HOOKS

WooCommerce (like any good WordPress plugin) has an incredible number of action and filter hooks you can use to customize how the

plugin is working on your site. Check out their [full action and filter hooks reference](#).

In the following, we will go through some examples to give you an idea of what's possible using a few typical WooCommerce hooks.

### *Setting a sitewide sale*

There are plugins you can use to set a sitewide sale for your WooCommerce store, but since you're an awesome developer, you might like to do it yourself through custom code. The following code applies a 10% sale to any product that doesn't already have a sale price:

```
// Set sale price to 10% of regular price if not already
on sale
function my_get_sale_price($sale_price, $product) {
    if(empty($sale_price)) {
        $sale_price = $product-
>get_regular_price() * .9;
        $product->set_price($sale_price);
    }

    return $sale_price;
}
add_filter('woocommerce_product_get_sale_price',
'my_get_sale_price',
10, 2);
add_filter('woocommerce_product_variation_get_sale_price
',
'my_get_sale_price', 10, 2);
```

WooCommerce products have both a “regular” price and a “sale” price. The final calculated price is just called “price.” The hook you can use to set the sale price is `woocommerce_product_get_sale_price`; however, you

will have to use the

`woocommerce_product_get_variation_sale_price` hook as well to handle products with variations (e.g., Small, Medium, Large T-Shirts).

Note also the line where we call `$product->set_price($sale_price)`. The calculated price isn't updated automatically after the sale price is returned from this callback, so we have to do it manually. The `$product` parameter is passed into the callback by reference; updating the product object here updates it outside of the filter as well.

### *Autocompleting orders*

By default, when a new order is created at checkout, the order is put into “pending” status. A site administrator then needs to process the order. For typically shipped goods, *process* means putting the order items into a box and shipping it out. Once that is done, the order is placed into “completed” status.

For virtual goods, you might want to automatically move the items into “completed” status. Besides saving you a click, many WooCommerce plugins fire when an order goes into completed status. The sooner an order goes into completed status, the sooner those plugins can send their emails, hit their APIs, or whatever else they are doing. The following code will loop through the cart items after checkout and automatically mark the order as completed if all the items in the cart are virtual:

```
function autocomplete_virtual_orders($order_id) {
    //get the existing order
    $order = new WC_Order($order_id);

    //assume we will autocomplete
    $autocomplete = true;

    //get line items
    if (count( $order->get_items() ) > 0) {
```

```

foreach ($order->get_items() as $item) {
    if($item['type'] == 'line_item') {
        $_product = $order->get_product_from_item(
$item );
        if(!$_product->is_virtual()) {
            //found a non-virtual product in the
cart
            $autocomplete = false;
            break;
        }
    }
}

//change status if needed
if(!empty($autocomplete)) {
    $order->update_status('completed',
'Autocompleted.');
}
}

add_filter('woocommerce_thankyou',
'autocomplete_virtual_orders');

```

## Paid Memberships Pro

Paid Memberships Pro and other membership plugins focus on accepting payment for membership access to a WordPress site or app. Features of membership plugins include:

- Recurring pricing for subscriptions
- Tools for locking down content based on membership level

Documentation on how to set up Paid Memberships Pro and much more can be found on [the website](#).

## WHY WE LIKE PAID MEMBERSHIPS PRO

Besides being developed by coauthor Jason Coleman, Paid Memberships Pro is the only WordPress membership plugin that is 100% General Public License (GPL) and available for free in the WordPress repository. Other plugins require either paid modules or upgraded versions to gain access to all of the plugin's features.

All of the Paid Memberships Pro code is managed in public repositories on GitHub and open to developer input. As with WooCommerce, there are hooks and filters available to change the default behavior of the plugin.

Almost all membership sites have a slightly different way of calculating upgrades or special offers, or exactly how and when to lock down content. Instead of offering an extra-long settings page, Paid Memberships Pro carefully designed its hooks and filters to make it easy to set up nearly any pricing or content restriction model with just a few lines of code.

Another key difference between Paid Memberships Pro and some other membership plugins is that it uses its own table to define membership levels and their relationships to users and orders. Some membership plugins use the built-in WordPress user roles so that each membership level is also a user role. User roles are very important in some membership sites (see [Chapter 6](#)), but, in general, it's better to separate the concept of a membership level and a user role, allowing you (for example) to have members who are admins *and* members who are subscribers. If you do need to assign roles based on membership level, this is easily done with Paid Memberships Pro, and we show an example of this later in this chapter.

We share a few examples using Paid Memberships Pro hooks later. We'll cover one more unique ecommerce plugin before diving into some of the general concepts of ecommerce.

## Easy Digital Downloads

All the ecommerce plugins mentioned so far can be used for digital products and downloads as well as physical goods. However, if you plan on selling only digital goods, you should consider [Easy Digital Downloads](#), which was developed specifically for this use case.

Features of Easy Digital Downloads include the following:

- File downloads restricted to authorized customers only
- The ability to purchase multiple downloads at once

Documentation on how to set up Easy Digital Downloads and much more can be found on their [website](#).

### WHY WE LIKE EASY DIGITAL DOWNLOADS

Easy Digital Downloads includes extensions that could be useful to app developers, such as the Software Licensing and Product Support add-ons. The core plugin and all of the extensions are well coded and well supported.

Like WooCommerce, the core Easy Digital Downloads plugin is available for free in the WordPress repository, with extensions available for purchase at the plugin's website. Extensions can be purchased individually, but most users will want to buy one of the bundled passes that gives them access to most or all of the extensions at a reduced total price.

## EASY DIGITAL DOWNLOADS CODE EXAMPLES

Following are a couple of code examples showing how to use the Easy Digital Downloads (EDD) functions and hooks in your larger WordPress app. You can use the `edd_has_purchased()` function to check whether a user has purchased a specific EDD download. If you already use EDD to sell downloads from your site, you can use this function to restrict access to other WordPress pages or app features:

```
//Restrict access to a page if a user hasn't purchased a
specific download yet
function my_template_redirect_check_edd()
{
    global $current_user;

    //Set to slug of page to protect.
    $protected_page_slug = 'customers-only';

    //Set to ID of download to check for.
    $required_download_id = 184;

    //Only protecting one specific page.
```

```

    if(!is_page($protected_page_slug))
        return;

    //Redirect if no user or missing purchase.
    if(!is_user_logged_in() ||
        !edd_has_user_purchased($current_user->ID,
$required_download_id)) {

    wp_redirect(get_permalink($required_download_id));
        exit;
    }
}

add_action('template_redirect',
'my_template_redirect_check_edd');

```

In this example, we used the `template_redirect` hook in WordPress. If users haven't purchased the download we are checking for, they are redirected to the frontend page for that download. This is a good way to block access to an entire page (maybe a page that contains content or a shortcode for a contact form, or content you want to keep noncustomers from accessing).

You could do a similar check as a wrapper around any of your PHP code. If you do this often, it would help to abstract the `edd_has_user_purchased()` check into another function. The following examples includes a helper function to check whether a user has purchased a specific download and then uses that function to add a support link to the primary menu:

```

//Helper function to check if a user is a plugin customer
function is_plugin_customer($user_id = null)
{
    $plugin_download_id = 184;           //update this

    //default to current user

```

```

    if(empty($user_id))
    {
        global $current_user;
        $user_id = $current_user->ID;
    }

    return edd_has_user_purchased($user_id,
$plugin_download_id);
}

//Add a support link to primary menu for users who purchased
function add_support_link_to_menu($items, $args)
{
    if($args->theme_location == 'primary' &&
is_plugin_customer())
    {
        $items .= '<li class="menu-item menu-item-
type-post_type
menu-item-object-page
menu-item-support">';
        $items .= '<a href="/support/">Support</a>';
        $items .= '</li>';
    }

    return $items;
}
add_filter('wp_nav_menu_items', 'add_support_link_to_menu',
10, 2);

```

The `is_plugin_customer()` function applies a handy trick with the `$user_id` parameter. You can pass a specific `$user_id` to check, or optionally leave the parameter blank and the function will default to the current user's ID. The plugin's download ID (the post ID when editing that download in the dashboard) is hardcoded into the function and used in the `edd_has_user_purchased()` call that is returned.

The last bit of code uses the `wp_nav_menu_items` filter to add an additional link to the support page if the user is a plugin customer.

Let's cover some general ecommerce concepts, before diving into more examples on how to set up a SaaS with WordPress.

## Payment Gateways

A payment gateway is a service that processes and sometimes stores customer credit card information, and makes sure the money winds up in your bank account.<sup>3</sup> Popular payment gateways in the United States include Stripe, PayPal, Authorize.net, and Braintree Payments. There are dozens of gateways, many specializing in particular parts of the world or in particular markets.

These are the important things to look out for when choosing a gateway:

- Does the gateway support the country and currency in which you do business?
- Does the gateway integrate with the plugin you are using for ecommerce?
- Does the gateway work with the type of business you are in? Some gateways will not work with adult sites, gambling sites, or other “high-risk merchants.”
- Does the gateway offer the features you need, like recurring billing or stored credit cards?
- How does the gateway handle Payment Card Industry (PCI) compliance?<sup>4</sup>

- Will the gateway work compatibly with your merchant account (discussed in the next section)?
- Finally, what are the fees? One percent of \$10 million is a lot of money, and it is worth it for you to fight for lower fees. However, in general, the fees are fairly standard across gateways, and you should start by looking for a gateway that will work with your business setup. As your business grows in revenue and volume, it becomes easier to negotiate lowering your fees to the standard minimums in your industry.

## Merchant Accounts

Merchant accounts are often confused with payment gateways, but this type of account is actually a separate thing that you will need to process payments on your website. Part of the confusion comes from the fact that some gateways use their own merchant accounts.

In any case, both a payment gateway and merchant account are required to make money online, and both kinds of providers will help you secure the other service. That is, you can shop for a payment gateway and have it help you find a merchant account, or you can shop for a merchant account and have it help you find a payment gateway. We find that younger companies typically get better fees when they start with a payment gateway and get a merchant account with their help, rather than going to their bank to open a merchant account.

Here is how the credit card information and money flows from a customer on your website into your checking account: WordPress →

Ecommerce Plugin → Payment Gateway → Merchant Account → Your Checking Account.

One way in which you can think of the difference between payment gateways and merchant accounts is that the payment gateway is largely technology related, and the merchant account is largely related to the business. The payment gateway provides the technology for you to validate and charge a credit card, and can also set up recurring payments. In addition, some payment gateways can store customer information for later billing.

The merchant account is a kind of bank account that stores incoming money until it can later be moved to your bank account. Why doesn't it just put the money directly into your bank account? The delay involved is kind of like waiting for a check to clear, or waiting for a credit card to validate. If for some reason the credit card company needs to request the money back (because of an error or a customer request), it can then pull the money out of your merchant account during this delay period.

These are the important things to look out for when choosing a merchant account:

- Will my gateway work with this merchant account?
- Will this merchant account underwrite my type of business? Some merchant accounts will not work with adult or gambling sites, or other types of "high-risk merchants."
- Will this merchant account underwrite my size of business? Some merchant accounts won't approve new businesses that

sell high-priced (thousands of dollars) goods.

- Finally, what are the fees? Sometimes these fees are bundled into the payment gateway fees, and sometimes they are separate.

The best route to accepting credit cards online is usually to choose a plugin first, then choose a payment gateway that works with that plugin, and then work with the gateway to find a merchant account.

## **Setting Up SaaS with Paid Memberships Pro**

The model of charging for access to a web app is called SaaS (pronounced “sass”). In this section, we will go through setting up Paid Memberships Pro on our SchoolPress app, with a checkout option for schools to pay \$1,000/year.

### **The SaaS Model**

The SaaS model basically means that instead of purchasing your software in a shrinkwrapped box and installing it on your computer, you pay—typically a monthly or annual fee—for access to a cloud-enabled web app. Examples of companies using the SaaS model are GitHub, Dropbox, Evernote, and Google Apps, and now even Microsoft Office can be purchased under a SaaS plan.

SaaS is popular because it generates relatively predictable recurring revenue. But SaaS is also an important way to make money from open source software like WordPress. Because WordPress is GPL, if

you were to sell and distribute the code for your software, your customers would be entitled under the GPL to redistribute that code —potentially for free. So using the SaaS service model allows your customers to use your software without having to distribute your source code to them.

The following instructions will help you if you want to charge a one-time, monthly, or annual fee for access to your app.

## **Step 0: Establishing How You Want to Charge for Your App**

Is it a lifetime fee, or a monthly subscription? Is it an annual subscription? Does the subscription automatically bill every year, or does the customer have to renew?

Answer these questions as best as you can before you start integrating Paid Memberships Pro or coding up customizations. Jason has a good series on how to [price your web apps and premium content sites](#).

For our SchoolPress app, we will be charging each school account a \$1,000 annual fee. When a school signs up, we will create a WordPress network site for it (e.g., *myschool.schoolpress.com*) and give it administrator access to that site so it can start adding teachers and other content.

We will set the membership level to automatically bill the schools each year.

## **Step 1: Installing and Activating Paid Memberships Pro**

Paid Memberships Pro is available in the WordPress plugin repository, which makes installing and activating the plugin a breeze (Figure 15-1).

1. From your WordPress dashboard, go to Plugins → Add New.
2. Find Paid Memberships Pro, and then click the Install link.
3. Optionally, enter your FTP information here (some hosting setups won't require this).
4. When the plugin installs successfully, click the Activate link.

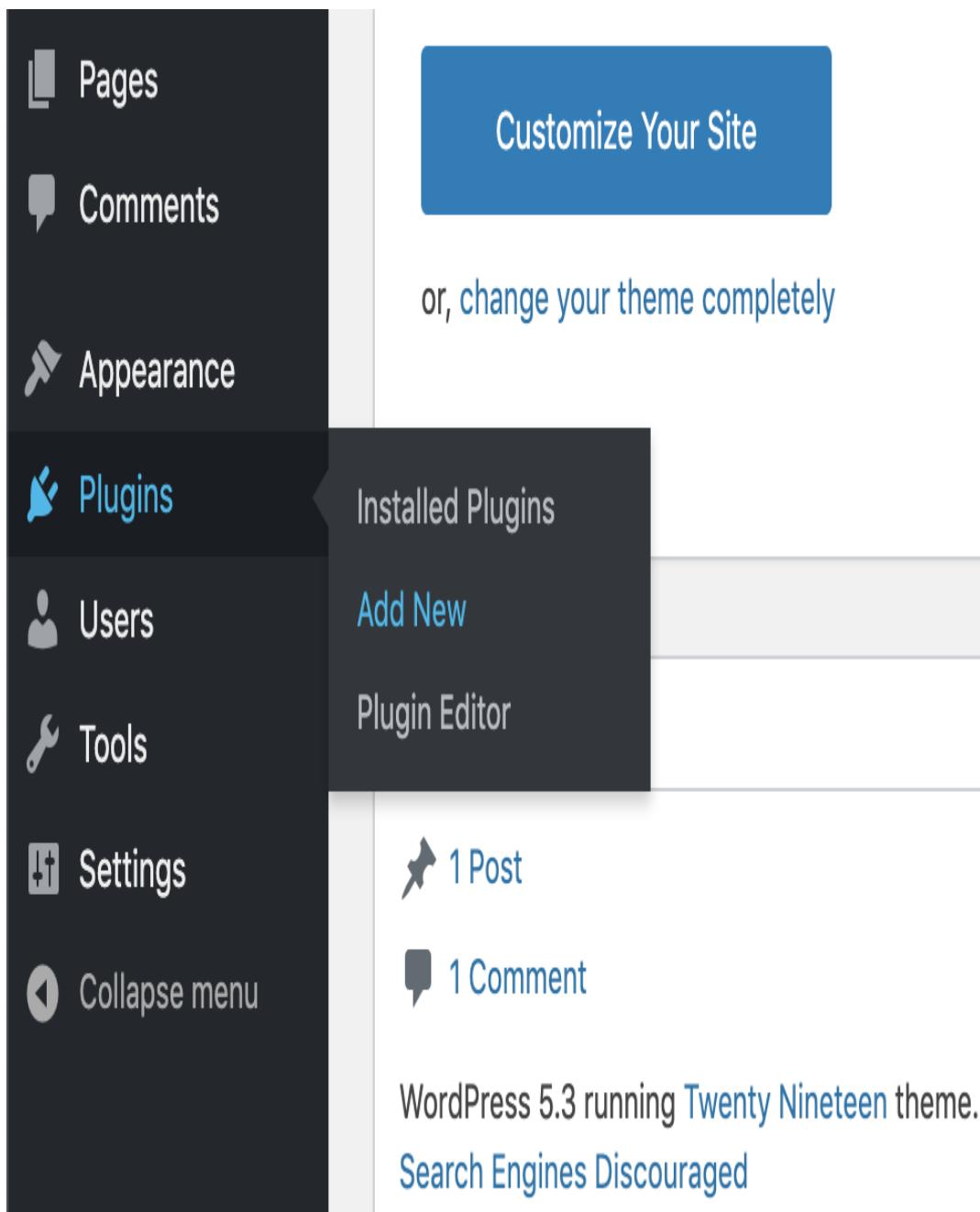


Figure 15-1. Adding a new plugin

## Step 2: Setting Up the Level

Now, you'll need to set up your membership levels:

1. From your WordPress dashboard, go to the newly created Memberships page.

2. Click the “Add new level” link or button.
3. Enter the membership information in the boxes, as shown in Figure 15-2. For our level, that will be:
  - a. Name: <School Name>
  - b. Description: School administrators should sign up here to create and gain access to your SchoolPress site.
  - c. Confirmation Message: (can leave it blank)
  - d. Initial Payment: 1000
  - e. Recurring Subscription: Checked
    - i. Billing Amount: 1000
    - ii. Per: 1
    - iii. Days/Weeks/Years: Years
    - iv. Billing Cycle Limit: 0
    - v. Custom Trial: Unchecked
  - f. Disable New Signups: Unchecked
  - g. Membership Expiration: Unchecked
  - h. Categories: All Unchecked
4. Click Save Level.

## Add New Membership Level

ID:

Name:

Description:

Add Media

Visual Text

b i link b-quote del ins img ul ol li code more close tags

Confirmation Message:

Add Media

Visual Text

b i link b-quote del ins img ul ol li code more close tags

### Billing Details

Initial Payment:

The initial amount collected at registration.

Recurring Subscription:

Check if this level has a recurring subscription payment.

---

### Other Settings

Disable New Signups:

Check to hide this level from the membership levels page and disable registration.

Membership Expiration:

Check this to set when membership access expires.

---

### Content Settings

Categories:

Uncategorized

**Save Level**

**Cancel**

Figure 15-2. The Paid Memberships Pro Add New Membership Level screen

## Step 3: Setting Up Pages

Paid Memberships Pro needs several pages to facilitate checkout and other membership-related functions. When you click the Pages tab of the Paid Memberships Pro settings, you will see a form like the one shown in Figure 15-3.

The screenshot shows the 'Pages' tab selected in the Paid Memberships Pro settings menu. The page title is 'Pages'. Below the title, there is a note: 'Assign the WordPress pages for each required Paid Memberships Pro page or [click here to let us generate them for you.](#)' followed by a hand cursor icon. There are three dropdown menus for selecting pages:

- Account Page:** -- Choose One --  
Include the shortcode [pmpro\_account].
- Billing Information Page:** -- Choose One --  
Include the shortcode [pmpro\_billing].
- Cancel Page:** -- Choose One --  
Include the shortcode [pmpro\_cancel].

Figure 15-3. Generate pages for Paid Memberships Pro

If you already have pages dedicated to describing your levels or for a user account, you can choose those pages through the drop-downs on the Pages tab of the Paid Memberships Pro settings. In most cases, though, you'll just want to click the "click here to let us generate them for you" link. Pages will be created for the Account, Billing Information, Cancel, Checkout, Confirmation, Invoice, and Levels pages. [Figure 15-4](#) shows what this page looks like once the script has generated these pages for you.

## Pages

The following pages have been created for you: 50, 51, 52, 53, 54, 55, 56.

Manage the WordPress pages assigned to each required Paid Memberships Pro page.

### Account Page:

Membership Account

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_account].

### Billing Information Page:

Membership Billing

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_billing].

### Cancel Page:

Membership Cancel

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_cancel].

### Checkout Page:

Membership Checkout

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_checkout].

### Confirmation Page:

Membership Confirmation

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_confirmation].

### Invoice Page:

Membership Invoice

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_invoice].

### Levels Page:

Membership Levels

[edit page](#)

[view page](#)

Include the shortcode [pmpro\_levels].

Figure 15-4. Default pages generated by Paid Memberships Pro

## **Step 4: Choosing Payment Settings**

Figure 15-5 shows the Payment Gateway & SSL tab of the Paid Memberships Pro settings. Here you will choose your gateway and then fill out the corresponding user and/or API values. Depending on which gateway option you choose, this page will also allow you to change the currency used, which credit cards are available, whether to use SSL or not (remember you should always install SSL unless it is a test site), and whether to use the Nuclear Option for SSL or not.

The screenshot shows the 'Payment Gateway & SSL Settings' page of the Paid Memberships Pro plugin. At the top, there's a logo, the text 'PaidMembershipsPro v1.7.6', and links for 'Plugin Support' and 'User Forum'. Below the header is a navigation bar with tabs: 'Membership Levels', 'Pages', 'Payment Gateway & SSL' (which is active), 'Email', 'Advanced', and 'Add Ons'. The main content area has three sections: 'Payment Gateway:' set to 'Testing Only', 'Gateway Environment:' set to 'Sandbox/Testing', and 'Currency:' set to 'US Dollars (\$)' with a note about currency support. Under 'Accepted Credit Card', there are checkboxes for 'Visa' (checked), 'Mastercard' (checked), and 'American Express' (unchecked).

Payment Gateway: Testing Only

Gateway Environment: Sandbox/Testing

Currency: US Dollars (\$) Not all currencies will be supported by every gateway. Please check ✓

Accepted Credit Card:  Visa  Mastercard  American Express

Figure 15-5. Paid Memberships Pro payment settings

On the payment settings page, you can also paste in your SSL Seal Code, and enter a tax state and percentage. The tax calculation can also be done programmatically through the `pmpro_tax` filter (described shortly).

The payment settings page also shows you the URL to share with your gateway to enable behind-the-scenes communication from the gateway to your site. This function has various names, depending on the gateway: PayPal calls it an “IPN handler”; Authorize.net calls it a “silent post URL”; and Stripe and Braintree call it a “webhook.”

## Step 5: Choosing Email Settings

By default, WordPress will send emails from your site from “WordPress” at *wordpress@<yoursite>.com*. This doesn’t look nice and is often not a real email address. The Email tab of the Paid Memberships Pro settings, shown in Figure 15-6, allows you to override these values and also check or uncheck which membership-related admin emails you would like to receive.

The screenshot shows the 'Email' tab selected within the Paid Memberships Pro plugin's interface. The top navigation bar includes the logo, 'Paid Memberships Pro v1.7.6', 'Plugin Support', and 'User Forum'. Below the navigation is a breadcrumb trail: 'Paid Memberships Pro - Membership Plugin for WordPress'. The main content area is titled 'Email Settings'. It contains instructions about modifying email headers and footers via theme files, followed by a link to learn more about Paid Memberships Pro emails. There are three input fields: 'From Email' set to 'wordpress@schoolpress.me', 'From Name' set to 'WordPress', and 'Send the site admin emails:' with a checked checkbox next to the text 'when a member checks out.'

Paid Memberships Pro - Membership Plugin for WordPress

v1.7.6 Plugin Support User Forum

Membership Levels Pages Payment Gateway & SSL Email Advanced Add Ons

## Email Settings

By default, system generated emails are sent from `wordpress@yourdomain.com`. You can update this from address using the fields below.

To modify the appearance of system generated emails, add the files `email_header.html` and `email_footer.html` to your theme's directory. This will modify both the WordPress default messages as well as messages generated by Paid Memberships Pro. [Click here to learn more about Paid Memberships Pro emails.](#)

**From Email:**

**From Name:**

**Send the site admin emails:**  when a member checks out.

Figure 15-6. Paid Memberships Pro email settings

## NOTE

You might want to send nicer-looking emails regardless of whether you're using Paid Memberships Pro. See the [email-related code](#) to adapt for your own website.

## Step 6: Choosing Advanced Settings

Figure 15-7 shows the Advanced Settings tab, which has a few built-in options for running Paid Memberships Pro. Of particular interest may be the option to choose a Terms of Service page to show users on sign-up. They will see a scrollable text box with the TOS page content shown within it and will have to check a box to agree to the Terms of Service.



**PaidMembershipsPro** v1.7.6 Plugin Support User Forum

Membership Levels Pages Payment Gateway & SSL Email Advanced Add Ons

## Advanced Settings

**Message for Logged-in Non-members:** This content is for !!levels!! members only.  
<a href="http://schoolpress.me/wp-login.php?action=register">Register</a>  
This message replaces the post content for non-members. Available variables: !!levels!!, !!referrer!!

**Message for Logged-out Users:** This content is for !!levels!! members only.  
<br /><a href="http://schoolpress.me/wp-login.php">Log In</a> <a href="http://schoolpress.me/wp-login.php?action=register">Register</a>  
This message replaces the post content for logged-out visitors.

**Message for RSS Feed:** This content is for !!levels!! members only. Visit the site and log in/register to read.  
This message replaces the post content in RSS feeds.

Figure 15-7. Paid Memberships Pro advanced settings

## NOTE

As of May 2018, Paid Memberships Pro also integrates with core functionality in WordPress to support the EU's General Data Protection Regulation. If your public app will have EU users (or really, if you care about privacy in general), you should consider how the GDPR relates to your web app. Jason's [post on the Paid Memberships Pro blog](#) gives a good overview of how he thought about the GDPR and updated Paid Memberships Pro to support it.

## Step 7: Locking Down Pages

Besides generating a checkout page and integrating with your payment gateway, the main functionality added by Paid Memberships Pro is the ability to lock down certain pages or portions of pages based on a user's membership level. There are a few different ways to do this.

### LOCK DOWN A SPECIFIC PAGE

Paid Memberships Pro adds a Require Membership checkbox (see [Figure 15-8](#)) to the sidebar of the edit post and edit page pages in the WordPress dashboard. To lock down a page for a certain membership level, check the box next to that level.

If more than one level is selected, members of *either* level can view that page. If no levels are checked, everyone (including nonusers) can view the page.

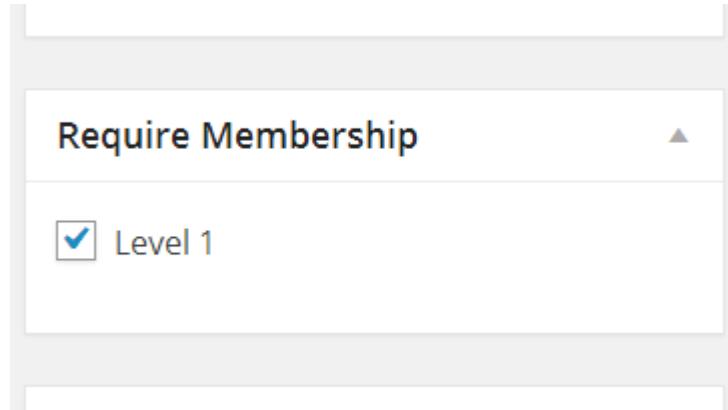


Figure 15-8. Select the levels required to view this page

## LOCK DOWN A PAGE BY URL

Sometimes it may be easier to restrict access to a page or group of pages by checking the page's URL. For example, to keep nonmembers out of certain BuddyPress groups, you could add the following code to a custom plugin:

```
//lock down our members group
function my_buddy_press_members_group()
{
    $uri = $_SERVER['REQUEST_URI'];
    if(strtolower(substr($uri, 0, 16)) ==
"/groups/members/")
    {
        //make sure they are a member
        if(!pmpro_hasMembershipLevel())
        {
            wp_redirect(pmpro_url("levels"));
            exit;
        }
    }
}
add_action("init", "my_buddy_press_members_group");
```

## NOTE

Since we published the first edition of this book, Paid Memberships Pro has launched an add-on to integrate with BuddyPress. To lock down BuddyPress, use the [PMPRO BuddyPress Add-on](#). We kept the example here since it does a good job of showing how to restrict any generic URL based on a user's membership level.

The workhorse here is the `pmpo_hasMembershipLevel()` function. This function can take two parameters. The first is the ID or name of a membership level to check for. The second parameter is the user ID of the user you want to check. If no parameters are set, the function will check whether the current user has *any* membership level.

You can also do negative checks by passing, for example, `-1` as the level ID. `pmpo_hasMembershipLevel(-1)` will return `true` if the current user *doesn't* have level 1. If you pass a `0` specifically, the function will check that the user has *no* level at all. So `pmpo_hasMembershipLevel(0)` will return `true` if the current user does not have a membership level. (You could also do `!pmpo_hasMembershipLevel()`.)

You can pass multiple level IDs and names in an array. For example, to check for members with level 1 or 2, use this code:

```
if(pmpo_hasMembershipLevel(array(1,2)))
{
    //do something for level 1 and 2 members here
}
```

## LOCK DOWN A PORTION OF A PAGE BY SHORTCODE

Another way to restrict access to content is to use shortcodes in your post body content. The following is an example of some page content that will show different messages to different membership levels:

```
Welcome to SchoolPress!
```

```
[membership level="1"]Thanks for your continuing membership.  
[/membership]
```

```
[membership level="-1"]Sign up your school now! [/membership]
```

The [membership] shortcode is fairly simple. It takes one parameter level that, similar to the parameter for the `pmpro_hasMembershipLevel()` function, can take a level ID, name, or a zero or negative level ID. Any content within the shortcode will be shown based on the stated level. Multiple level IDs can be passed separated by commas.

## LOCK DOWN A PORTION OF A PAGE BY PHP CODE USING THE PMPRO\_HASMEMBERSHIPLEVEL() FUNCTION

When locking down the BuddyPress members group, we used the `pmpro_hasMembershipLevel()` function. You can also use this function within your page templates or other code to restrict access to content or portions of code. For example, you might find code like this in your header:

```
<?php if(is_user_logged_in()) { ?>  
<div class="user-welcome">  
    Welcome  
    <?php if(function_exists("pmpro_hasMembershipLevel")) { ?>
```

```

&& pmpro_hasMembershipLevel()) { ?>
    <a href="php echo pmpro_url("account"); ?&gt;"&gt;
&lt;?php echo $current_user-&gt;display_name;?&gt;
&lt;/a&gt;
&lt;?php } else { ?&gt;
    &lt;a href="<?php echo home_url("/wp-
admin/profile.php"); ?&gt;"&gt;
        &lt;?php echo $current_user-&gt;display_name;?&gt;
    &lt;/a&gt;
&lt;?php } ?&gt;
&lt;/div&gt; &lt;!-- end userWelcome -->
<?php } ?>

```

The preceding code will show members a link to the Paid Memberships Pro account page. Users without a membership level are shown a link to their WordPress profile page.

## **Step 8: Customizing Paid Memberships Pro**

Next we describe a few common customizations for Paid Memberships Pro. The general process for customizing a plugin like this is to:

1. Figure out what you want to change.
2. Find out where the default behavior for your change is coded.
3. Locate or add a hook to support the customization you want.
4. Write an action or filter to use the hook.

## **RESTRICTING NONMEMBERS TO THE HOME PAGE**

By default, Paid Memberships Pro doesn't lock down any part of your site unless you specifically tell it to. For some sites, you will

want very limited public access (just the sales, about, and contact pages). You can do this by redirecting nonmembers away from any nonapproved page. Use the following code:

```
function my_template_redirect()
{
    $okay_pages = array(
        pmpro_getOption('billing_page_id'),
        pmpro_getOption('account_page_id'),
        pmpro_getOption('levels_page_id'),
        pmpro_getOption('checkout_page_id'),
        pmpro_getOption('confirmation_page_id')
    );

    //if the user doesn't have a membership, send them home
    if(!is_user_logged_in()
        && !is_home()
        && !is_page($okay_pages)
        && !strpos($_SERVER['REQUEST_URI'],
    "login"))
    {
        wp_redirect(home_url('wp-login.php?
    redirect_to='.
        urlencode($_SERVER['REQUEST_URI']))));
    }
    elseif(is_page()
        && !is_home()
        && !is_page($okay_pages)
        && !pmpro_hasMembershipLevel())
    {
        wp_redirect(home_url());
    }
}

add_action('template_redirect', 'my_template_redirect');
```

In the preceding code, we set up an array of post IDs for pages that nonmembers should be able to see. We used the `pmpro_getOption()` function to get the IDs of the pages generated by PMPro and also allow home page access by using the WordPress `is_home()` function. We also allow access to any page with the word *login* in the URL, which on our setup will just the login page.

## LOCKING DOWN FILES

Some of the pages you are locking down may have images or other files attached to them. If the page is locked down, the link or image will not show up on the site for your users. However, users who know the direct URL to the file will be able to download to the file without first being logged in as a member. This is because when Apache processes a URL like `http://schoolpress.me/wp-content/uploads/logo.png`, it serves the file directly to the user without checking with PHP or WordPress first.

You can change this behavior by adding a rule to your site's `.htaccess` file that redirects any URL like the preceding one through a special script bundled with Paid Memberships Pro. Add the following code to the top of your `.htaccess` file, above the other rewrite rules:

```
RewriteEngine On
RewriteBase /
RewriteRule ^wp-content/uploads/(.*)$ \
    /wp-content/plugins/paid-memberships-
    pro/services/getfile.php [L]
```

How does this work? In WordPress, images and files can be uploaded to a post or page. These files, called *attachments* by WordPress, are

all stored in the `/wp-content/uploads/` folder, but they are also associated with the post they were attached to via an entry in the `wp_posts` table.

Attachments are stored in the `wp_posts` table with the `post_status` set to `attachment` and the `post_parent` set to the ID of the post to which they're attached.

The `getfile.php` script will find the corresponding entry in the `wp_posts` table for the requested file; and if the attachment's parent requires membership, it will check to make sure a valid member is logged in before serving the file.

## CHANGE USER ROLES BASED ON MEMBERSHIP LEVELS

For most of the examples in this section, we assume that members only have access to your site's frontend application. However, sometimes you may want to give members access to the WordPress dashboard, give them the "author" role so they can post to the blog, or otherwise assign them a role other than "subscriber."

This code will add the author role to any new member of a particular level. It will also downgrade the member to a subscriber role if their membership level is removed:

```
function my_pmpo_after_change_membership_level($level_id,  
$user_id)  
{  
    if($level_id == 1)  
    {  
        //New member of level #1.  
    }  
}
```

```

    //If they are a subscriber, make them an author.
    $wp_user_object = new WP_User($user_id);
    if(in_array('subscriber', $wp_user_object-
>roles))
        $wp_user_object->set_role('author');
    }
    else
    {
        //Not a member of level #1.
        //If they are an author, make them a subscriber.
        $wp_user_object = new WP_User($user_id);
        if(in_array('author', $wp_user_object-
>roles))
            $wp_user_object-
>set_role('subscriber');
        }
    }
add_action(
    'pmpro_after_change_membership_level',
    'my_pmpro_after_change_membership_level',
    10,
    2
);

```

For more information on users and roles, see [Chapter 6](#).

## INTERNATIONAL AND LONG-FORM ADDRESSES

By default, the Paid Memberships Pro checkout form will show address fields with the city, state, and postal code formatted on their own lines. A drop-down to choose the country appears below the form, as illustrated in [Figure 15-9](#).

Billing Address

First Name  \*

Last Name  \*

Address 1  \*

Address 2

City  \*

State  \*

Postal Code  \*

Country   \*

Phone  \*

Figure 15-9. Paid Memberships Pro international billing address

You may want to change the default country, change the list of countries, or make some of the address fields not required. Here is some example code doing that:

```
/*
    Change the Default Country from US to GB (Great Britain)
*/
function my_pmpo_default_country($default)
{
    return 'GB';
}
add_filter('pmpo_default_country',
'my_pmpo_default_country');
```

```

/*
     Add/remove some countries from the default list.
*/
function my_pmpo_countries($countries)
{
    //remove the US
    unset($countries['US']);

    //add The Moon (LN short for Lunar?)
    $countries['LN'] = 'The Moon';

    //You could also rebuild the array from scratch.
    // $countries = array('CA' => 'Canada', 'US' =>
'United States',
    //   'GB' => 'United Kingdom');

    return $countries;
}
add_filter('pmpo_countries', 'my_pmpo_countries');

/*
(optional) You may want to add/remove certain
countries from the list.
The pmpo_countries filter allows you to do this.
The array is formatted like
array('US'=>'United States', 'GB'=>'United Kingdom');
with the acronym as the key and the full
country name as the value.
*/
function my_pmpo_countries($countries)
{
    //remove the US
    unset($countries['US']);

    //add The Moon (LN short for Lunar?)
    $countries['LN'] = 'The Moon';

    //You could also rebuild the array from scratch.
    // $countries = array('CA' => 'Canada', 'US' =>
'United States',
    //   'GB' => 'United Kingdom');

    return $countries;
}

```

```

add_filter('pmpro_countries', 'my_pmpro_countries');

/*
    Change some of the billing fields to be not
required.
    Default fields are: bfirstname,blastname,
baddress1,bcity,bstate,
bzipcode,bphone,bemail,bcountry,CardType,
AccountNumber,
ExpirationMonth,ExpirationYear,CVV
*/
function my_pmpro_required_billing_fields($fields)
{
    //remove state and zip
    unset($fields['bstate']);
    unset($fields['bzipcode']);

    return $fields;
}
add_filter('pmpro_required_billing_fields',
'my_pmpro_required_billing_fields');

```

- 
- 1 What counts as an “ecommerce platform”? Should market share be computed based on the number of sites or the number of sales? How do you account for hosted ecommerce solutions like Shopify or Amazon?
  - 2 As of this writing, the [BuiltWith](#) site estimates that 21% of ecommerce sites in the top one million of all sites use WooCommerce.
  - 3 Minus any fees.
  - 4 At the high end, PCI compliance requires more expensive server setups and full-time resources to maintain and document them properly. Some gateways have technology and processes to help you avoid those costs while still keeping your customer data secure.

# Chapter 16. Mobile Apps Powered by WordPress

---

Mobile applications that run on iOS and Android powered by WordPress? Well, sure, why not? You can harness the capabilities of WordPress to power both native and hybrid mobile apps in many ways.

Before we dive into building mobile apps powered by WordPress, let's think about possible use cases for building WordPress-powered apps. Although the sky's the limit, think content-driven apps. We'll also cover some things you should know about building hybrid mobile apps.

## Mobile App Use Cases

OK, so we have mobile apps that can display content from a web app; how interactive can it really be? Hybrid WordPress-powered mobile apps can be as interactive as you want them to be! Anything you can do with WordPress you can do in your hybrid mobile app:

- You can have your app users log in to your app via Facebook, Twitter, or other social networks that allow single sign-on (SSO).
- Maybe your web app is an online store powered by WooCommerce: your mobile app allows your customers to

browse and purchase products right from the app. Maybe your app enables multiple merchants to have stores on your network: with your app they can manage their inventory and upload photos of products in real time directly from their device.

- Maybe you built a dating website with BuddyPress: making a branded mobile app for your dating service could help boost your member base if you gain popularity. With native device features like the camera, your social network members could take and upload selfies directly to their profile.
- Maybe you built a real estate site, and you want agents to be able to create new posts in a CPT called “Homes” for a property they’re trying to sell. With the web app users could be snapping photos and posting them with the longitude and latitude, so the listing appears in Google Maps.
- You could make a mobile app for construction contractors, plumbers, landscapers, electricians, doctors, dentists, lawyers, accountants, or any other type of physical service-based business that could easily assist them in tracking jobs, accessing client details, uploading before-and-after photos or videos, tracking job locations via GPS, collaborating with coworkers, and more.
- Maybe you already have time and money invested in some existing WordPress-powered system that you want to make available via an app in the app stores.

The possibilities of what you can build are limitless, especially the more you incorporate native functionality into your hybrid mobile application. If required, you could always build 100% native mobile

apps and still make them access an online WordPress application so your content and user base is the same across all platforms.

## Native and Hybrid Mobile Apps

Using the WordPress REST API in either a full native or hybrid mobile app would be the preferred way to integrate with WordPress and to optimize the app's performance. If you are already a native app developer or an aspiring one, this might be your preferred method. If you are a web developer and have no intention of learning Objective-C, Swift, Kotlin, and/or Java, a hybrid mobile app is what you want, and you came to the right place.

### CAUTION

Depending on the specific project needs and your skills or those of your development team, a hybrid mobile app might not always be the best decision.

## What Is a Native Mobile App?

A native mobile application is a smartphone and/or tablet application that is developed in a specific programming language for a specific operating system:

- Objective C = iOS
- Java = Android

Here are some advantages of building a native mobile app:

### *Performance*

Native apps are created and optimized for specific platforms utilizing core languages, libraries, and APIs, making them very fast and responsive.

### *Native libraries*

Utilizing and having access to all of the native libraries directly has a huge advantage over having access to native library features only via wrapper plugins.

### *Security*

Native apps can be more secure than hybrid apps because they depend only on their platform, as opposed to hybrid apps, which depend on multiple technologies such as JavaScript, HTML, and CSS.

And some disadvantages of building a native mobile app:

#### *More time*

To place your app on more than one app store, you'll need to build the same thing more than once.

#### *More money*

Besides time spent building the same thing for multiple platforms, native iOS and Android developers generally cost more to hire.

#### *Less talent*

There are far more web developers than there are native mobile app developers.

## **What Is a Hybrid Mobile App?**

Hybrid apps run on smartphones and tablets just like native apps do but are written with web technologies such as JavaScript, HTML5, and CSS. Hybrid apps run within a native container or app wrapper and use the device's browser engine to render the HTML and process the JavaScript locally. You can develop and use native plugins via JavaScript to access native device features like the accelerometer, camera, GPS, and local storage. Basically a hybrid app is a website inside a native app that is made to look and feel like a full native app.

## Why Hybrid over Native?

Nowadays, more than two-thirds of mobile app developers are choosing to build hybrid mobile apps over native mobile apps. Saving time and money are big reasons why hybrid mobile apps are becoming more and more popular. Imagine the amount of time you or your company might invest in building, supporting, and maintaining a brand-new website, a native iOS app, and a native Android app. If you build a similar user experience (UX) and duplicate the same functionality three times, you'll have to maintain three codebases. Developing hybrid apps can take a fraction of the time that it can take to build the equivalent native apps because you can use single codebases like WordPress to power the features and functionality within your apps for multiple platforms like iOS and Android. Yes, in some cases you might essentially be loading your website into an iframe inside an app, but if your app does everything you need it to and has your business or client's business in the app stores, it could be a cost-effective solution for you.

Some benefits of building WordPress-powered hybrid mobile apps:

### *Do what you know*

You and/or your development team are already jamming with WordPress. Spend time building cool features for your app instead of learning how to build an app.

### *Shared codebase*

Why reinvent the wheel? To add custom functionality to your mobile app, head to the WordPress plugin repository or build a custom plugin to do what you want. Power your website, iOS app, and Android app with the same codebase.

### *Consistent UX*

Because you are using web-based technologies like JavaScript, HTML, and CSS, you can easily update and manage your UX to be consistent across all platforms running your app.

### *Future proofing*

Take advantage of advancing web technologies and development stacks used to make developing mobile apps and deploying mobile app updates easier and easier. A web-based platform is where you want to be able to easily extend your app's reach across multiple platforms today and tomorrow.

## **Cordova**

Apache Cordova is a web-based open source framework for building hybrid mobile apps with HTML, CSS, and JavaScript.

## **PHONEGAP**

Owned by Adobe, PhoneGap is the commercial version of Cordova. Much like WordPress.com and WordPress.org, Cordova and

PhoneGap share the same codebase. PhoneGap is Cordova but with an easy-to-use compiler and customer support.

## INSTALLING CORDOVA

Create a directory for your app projects and navigate to it in your terminal. To install Cordova, first install `Node.js`, an open source JavaScript runtime environment. Then type `node -v` in your terminal. This returns the current version and verifies that you have Node.js installed and running.

Cordova uses npm to install: run the command `sudo npm install cordova -g` in the terminal—note the `-g` for installing Cordova globally. Once Cordova is done installing, the terminal should display the current Cordova version information.

To create a new app project, run your own variation of the command `cordova create TestApp com.appresser.apps.testapp TestApp` in the directory where you want to store your app project, using your own values. The `cordova create` command accepts three parameters:

- App Project Directory: `TestApp`
- App Package Name:  
`com.appresser.apps.testapp` (usually a reverse domain name is a good standard to use)
- App Name: `TestApp`

Let's quickly review the files and directories Cordova created:

## *config.xml*

Default settings file pre-Cordova version 7.

## *package.json*

Default settings file starting in Cordova version 7.

## *hooks*

Cordova hooks are custom scripts for customizing Cordova commands.

## *platforms*

By default the platform's directory is empty, but this directory will contain the files and code used to integrate Cordova with each platform software development kit (SDK) like Android and iOS.

## *plugins*

Cordova plugins that your app uses. Any plugins that interact with native features will go here.

## *www*

All your web resources that will be wrapped in your app's web view.

It's almost time to add platforms to your new app project, but first you should check out the platform guides on the Cordova website to make sure your environment meets all the requirements for each platform for which you'll be building an app:

- [Android Platform Guide](#)
- [iOS Platform Guide](#)

## NOTE

A full list of available CLI commands can be found on the [Cordova CLI Commands page](#).

## CORDOVA AND ANDROID

Before you add all the necessary Cordova platform files for using the Android SDK, be sure you have access to the Android SDK via [Android Studio](#). Install Android Studio now, if you haven't done so already.

## NOTE

Make sure you install the Android Emulator from the [Android SDK Manager](#) if you intend on using the emulator.

With Android Studio installed, run the following command in your terminal, from within your new app project directory:

```
cordova platform add android
```

Terminal should output something like this:

```
Brians-MBP:TestApp bmess$ cordova platform add android
Using cordova-fetch for cordova-android@^8.0.0
Adding android project...
Creating Cordova project for the Android platform:
  Path: platforms/android
  Package: com.appresser.apps.testapp
  Name: TestApp
  Activity: MainActivity
  Android target: android-28
```

```
Subproject Path: CordovaLib
Subproject Path: app
Android project created with cordova-android@8.1.0
Plugin 'cordova-plugin-whitelist' found in config.xml...
Migrating it to
package.json
Discovered saved plugin "cordova-plugin-whitelist". Adding
it to the project
Installing "cordova-plugin-whitelist" for android
Adding cordova-plugin-whitelist to package.json
```

If you look in the *platforms* directory of your app project directory, you should see a new *android* directory that will contain all of the default resources your app will need to run on the *Android* platform.

If you want to compile your new app project in order to test it you have to run the command `cordova build` in the terminal. This command will build your app for all available platforms. You can also limit the scope of the build to specific platforms by running the command `cordova build android`.

Once you receive the `BUILD SUCCESSFUL` message in the terminal, you'll notice a path to where a newly built *.apk* file is located. This APK file is your compiled Android app, which you can install and test on any Android device. To plug your Android device directly into your computer and configure it for testing, run your app directly on your device using the `cordova run android` command.

If you do not have an Android device and would like to test your app in the Android emulator, run the `cordova emulate android` command in the terminal.

## CORDOVA AND IOS

Just like how you need access to the Android SDK files from Android Studio, you need the iOS SDK files from Xcode. If you do not have Xcode installed, do it now (or you won't be able to build an iOS app).

To add iOS as a supported platform in your new Cordova project, use the `cordova platform add ios` command in the terminal.

You should get a similar response to the following:

```
Brians-MBP:TestApp bmess$ cordova platform add ios
Using cordova-fetch for cordova-ios@^5.0.0
Adding ios project...
Creating Cordova project for the iOS platform:
  Path: platforms/ios
  Package: com.appresser.apps.testapp
  Name: TestApp
iOS project created with cordova-ios@5.0.1
Installing "cordova-plugin-whitelist" for ios
Brians-MBP:TestApp bmess$
```

The Cordova CLI commands for building, running, and emulating your app for iOS are the same as the Android commands, plus `ios` at the end of each command:

*cordova build ios*

Compiles the app into an app build.

*cordova run ios*

Runs the app directly on an iOS device that is configured for testing and plugged into your computer.

*cordova emulate ios*

Runs the iOS app on an IOS emulator.

## CORDOVA PLUGINS

Just like WordPress has plugins that can extend the functionality of your website, Cordova has plugins that extend the functionality of your app. Cordova plugins interact with native device features via JavaScript.

There are core Cordova plugins and third-party Cordova plugins available to install and use on your app, and you can also build your own! Core Cordova plugins include the following:

### *BatteryStatus*

Provides an API accessing `batterystatus`, `batterycritical`, and `batterylow` events.

### *Camera*

Provides an API for taking pictures and for choosing images from the device image library.

### *Capture*

Provides access to the device's audio, image, and video capture capabilities.

### *Connection*

Provides information about the device's cellular and WiFi connection, and whether the device has an internet connection.

### *Device*

Provides information about the device's hardware and software.

### *Events*

Provides various event listeners to be used within your app including deviceready, pause, resume, backbutton, menubutton, searchbutton, startcallbutton, endcallbutton, volumedownbutton, volumeupbutton, and activated.

#### *File*

Implements a File API allowing read/write access to files residing on the device.

#### *Geolocation*

Provides information about the device's location, such as latitude and longitude.

#### *Globalization*

Provides information and performs operations specific to the device locale, language, and time zone.

#### *InAppBrowser*

Enables the ability to view web pages from within your app without leaving it.

#### *Media*

Provides the ability to record and play back audio files on a device.

#### *Notification*

Provides access to some native dialog UI elements like alert, confirm, prompt, and beep.

#### *Splashscreen*

Displays and hides a splash screen during application launch.

#### *Status Bar*

Provides functions to customize the iOS and Android StatusBar.

### *Storage*

Provides functions for local storage on the device.

### *Vibration*

Provides a way to vibrate the device.

For a list of other available noncore Cordova plugins, check out the [Cordova Plugin search page](#).

To add a Cordova plugin to your project, use the `cordova plugin add` command with the name of the plugin. For example, to add the camera plugin to your project you would use the full camera plugin name at the end of the command:

```
cordova plugin add cordova-plugin-camera
```

You can also add a plugin by using a Git repository URL. For instance:

```
cordova plugin add https://github.com/apache/cordova-plugin-camera.git
```

At any time while in your project directory, you can list the plugins you have installed by running the `cordova plugin ls` command.

## Ionic Framework

Ionic Framework is an open source mobile app development framework for easily building native iOS and Android apps, as well as progressive web apps with web-based technologies like JavaScript, HTML, and CSS.

Ionic works with Cordova and has an extensive CLI library, so that you can run pretty much all the Cordova commands you just learned about.

1. Install Ionic CLI from <http://bwawwp.com/ionic-cli>. Once you've installed Node.js and npm, go ahead and run:

```
sudo npm install -g ionic
```

- Test Ionic version once installed with `ionic -v`.
- Test Node.js version with `node -v`.
- To install Cordova and Ionic use `sudo npm install -g cordova ionic`.

2. After Ionic is installed, start a new app by using the Ionic `start` command and passing in an app name, template type, and framework (we'll use `tabs` here, but note that there are other layout types):

```
ionic start SchoolApp tabs angular
```

3. Navigate to your new Ionic project directory and list the files:

```
cd SchoolApp  
ls
```

4. From within your new app project directory, run `ionic serve` to build and serve your app in a web browser, and boom! Instant progressive web app.
5. To use Ionic with Cordova in your app project to add native capabilities, run:

```
ionic cordova platform add android  
ionic cordova platform add ios
```

This adds the respective Cordova platform files to your new Ionic project.

Run `ionic cordova build android` to build your app and generate an *.apk file*.

6. Run `ionic cordova emulate android` to open up your new Android app in an emulator.

You can also launch it in an emulator:

```
ionic cordova emulate ios --no-native-run
```

Unless you specifically set the target emulator, you may need to have the emulator you want to use already open.

Now you are ready to start building progressive web apps, iOS apps, and Android apps powered by WordPress! You have a few choices. You can build local mobile app pages that pull content from the WordPress REST API with JavaScript, or you can load WordPress pages in a webview and load a special responsive mobile app-specific theme if you have purchased or built one. Both methods have their benefits; also, you can use both in the same app project, depending on what you're trying to do.

## **App Wrapper**

Do you already have (or are you planning to add) custom features and functionality in WordPress that you would like to duplicate in a mobile app? The easiest, quickest way to kill two birds with one stone is to wrap your existing WordPress site in an *app wrapper*.

An app wrapper is basically a native mobile application with a webview or an iframe, meaning the mobile app is really just a web browser embedded within the app itself. This embedded browser's URL points to your WordPress application, hopefully using a custom responsive mobile theme so it makes your website look and function like a mobile app when it's served from the mobile app. You can extend your hybrid app with more native functionality, like accessing the camera, GPS device, contacts, and more, but if you want a downloadable app in various mobile app marketplaces and don't have the time, money, or resources to build full-blown native apps or hybrid apps driven from the WordPress API, a basic app wrapper may be the way to go. Almost everything that can be done with WordPress or a WordPress plugin or theme, you can do in both your iOS and Android mobile apps. Imagine the possibilities!

## **AppPresser**

[AppPresser](#) is one of the easiest ways to start building a mobile app powered by WordPress plugins and themes. Using the AppPresser App Builder, you can visually build iOS and Android mobile apps without touching a line of code. Though if you do jump into the code AppPresser generates for you from the App Builder, you could customize your app and extend it however you want.

AppPresser is more than just a mobile theme and some plugins for WordPress; because it uses Cordova and Ionic Framework, it can integrate with device hardware like the camera, geolocation, and accelerometer. AppPresser-created apps can be uploaded to iTunes and Google Play just like native apps.

Setting up an AppPresser-powered app is easy. There are two parts to AppPresser: your WordPress website and the My AppPresser Dashboard.

## **INSTALLING AND CONFIGURING ON WORDPRESS**

When you sign up for AppPresser, you will gain access to the AppPresser Ionic theme, AppPresser plugins, and a My AppPresser dashboard account. The first thing you will need to do is install and activate the AppPresser core plugin. Log in to your WordPress website, go to Plugins → Add New, and search for AppPresser. Install and activate, then download the AP3 Ion Theme from your AppPresser account page. After the AppPresser core plugin is installed and activated, do the same for any other AppPresser plugins you want to use with your WordPress website.

Next, install the AppPresser AP3 WordPress theme in your themes directory. Go to Appearance → Themes → Add New → Upload, and upload the theme ZIP file. The AppPresser theme cannot be uploaded as a plugin. Don't activate the AppPresser theme—it just needs to be in your themes directory. When your website receives a request from your mobile app, it will serve up the AppPresser theme automatically instead of your default active theme.

## NOTE

If you ever see the AP3 Ion Theme active while logged in, it's probably because you were previewing the app and had a cookie set. It's not a bug; no one else is seeing that theme. Just clear your cookies, and it will reset.

To configure AppPresser and its plugins, visit the AppPresser settings menu on your WordPress dashboard menu. Add your site slug and app ID, which can be found by logging into *myappresser.com* and visiting Your App → General. Add those settings and save.

For each AppPresser plugin you have installed and activated, there will be another tab with configuration options. Some AppPresser plugins essentially bridge WordPress and Cordova plugins like accessing a device camera or GPS data. Other AppPresser plugins include custom integration with popular third-party WordPress plugins like WooCommerce, LearnDash, and BuddyPress via their REST APIs to provide customizable mobile app pages. AppPresser plugins currently include:

- AppBuddy
- AppCamera
- AppCommerce
- AppCommunity
- AppGeolocation
- AppPush
- AppShare

- AppWoo
- App Facebook Connect
- LearnDash
- In App Purchases

## THE APP BUILDER

After setting up the AppPresser plugins on your site, log in to your *myappresser.com* dashboard. Your app dashboard is where all of your apps will be displayed. Click the orange New App button, give your app a title, and then click “create app.”

Once your app is created, you will be redirected to the dashboard for your new app. Here, you can choose to customize your app, set up push notifications, and more. Click “Customize and Build App” to go to the app customizer, where you can change your app colors and add pages, menus, and more. This should look very familiar, as it’s basically the WordPress theme customizer, but instead of previewing your active WordPress theme you are previewing your mobile app (see Figure 16-1).

**Customizing > Menus**

**App Menu**

**Menu Name**

App Menu

Home	Custom Link ▼
About	Custom Link ▼
Bios	Custom Link ▼
Contact	Custom Link ▼
Sign Up	Custom Link ▼

**Photos** Custom Link ▲

**URL**

<https://bwawwp.com/photos/>

**Navigation Label**

Photos

Open link in a new tab

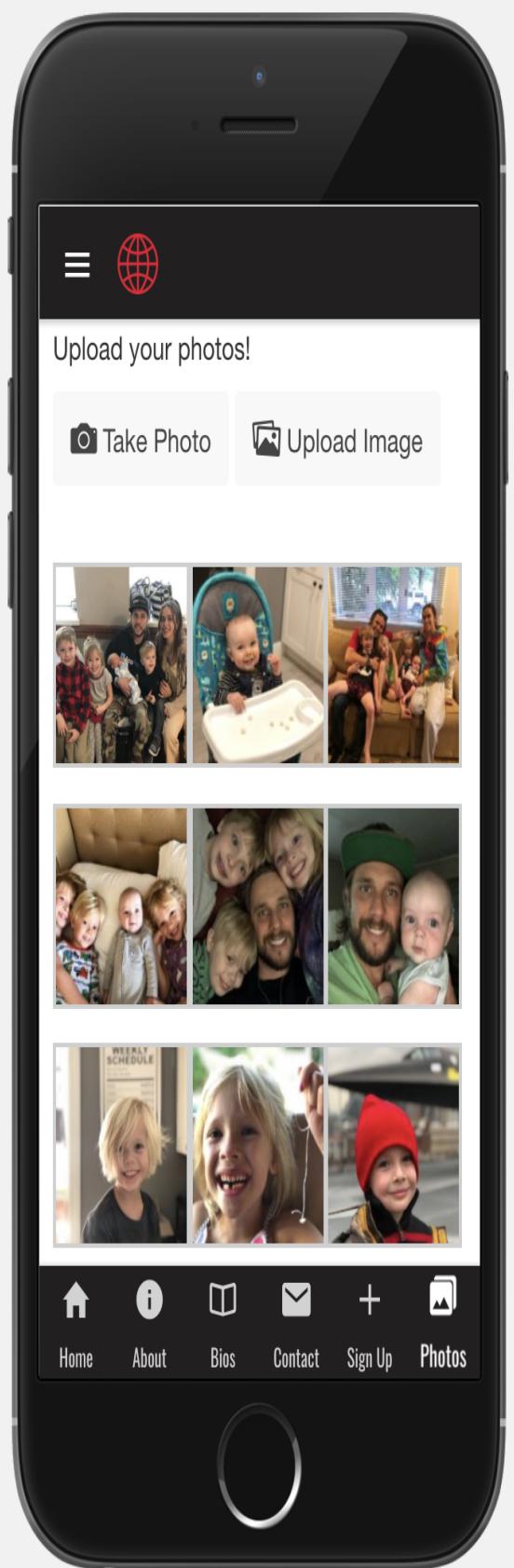
Icon class [Help ?](#)

photos

Extra classes [Help ?](#)

[Remove](#)

[Reorder](#) [+ Add Items](#)



*Figure 16-1. MyAppPresser Customizer*

**NOTE**

MyAppPresser is essentially a WordPress Multisite network that uses the WordPress theme customizer as its App Builder.

In the App Builder you'll put together and customize your app. Using the App Builder and customizing your app settings gives you almost complete control of how your mobile app will look, feel, and function. There are six items in the App Builder main menu:

*Colors*

Visually change the default colors of any of the elements available in your app. The default elements include: Body Background, Text Color, Button Background, Button Text, Left Menu Background, Left Menu Text, Left Menu Icon Color, Link Color, Headings Color, and Android Status Bar Background.

*Design*

Customize some of your app design elements like your heading fonts, body font, and any custom fonts you would like to use. You can also apply any custom CSS here to completely customize your app design.

*Settings*

Enter general information about your app that will be required in an app build, and further customize your app with additional settings. You can choose from using a side menu or a tab menu or both. You can enable the camera, push notifications, and geolocation. You can upload your app icon, splash screen, header

logo, and any additional offline assets such as local images to be displayed or videos to be played. You can upload a JavaScript file to use custom JavaScript within your app. You can choose the default language to be used within your app and even choose right to left (RTL) text if need be.

### *Build and Preview*

Here you'll build your app once you finish customizing it. Click Build App to compile your app for testing and/or app stores. Enter your PhoneGap Build auth token to compile for a device. This will allow you to use a QR code to scan and get the app onto your device. Note that iOS won't work without a signing certificate. You can also download the build files as a ZIP file here if you'd like to extend or customize your app further than what you can do with the App Builder.

### *Menus*

Control what pages will be available in your apps. You can add either custom links or custom pages. Add a page title and custom link with the full URL to the WordPress website page you would like to add to your app; this is how you can display any WordPress pages in your app using the AppPresser theme. If you have any custom app pages built you can also add them to the app here.

### *Custom Pages*

Build custom pages for your app. These are not regular WordPress pages loading from within your app; these will be local app pages that are highly customizable.

## NOTE

About AppPresser custom pages: they can be either static HTML that works offline, or list pages that fetch data from an API. Custom HTML pages are not specific to WordPress and can be any content you want. For example, if you create an About page that displays an image and text, it will be embedded in the app, so it works without an internet connection. You can also create a list page that displays REST API-driven content, such as posts or pages. When creating a custom page, you can choose to start from multiple templates; each custom page template provides various input fields for information such as page title, custom REST API Route URLs, and/or text boxes to provide custom Ionic/HTML5 markup.

When the App Builder is used in conjunction with AppPresser themes and plugins, you can quickly build iOS and Android apps from any WordPress website. Let's say your website has a membership plugin and LearnDash installed, and you want integration plus some WordPress posts in your app. First, install the AppPresser core plugin, add-on plugins, and theme on your site. Nothing changes on the website, because AppPresser is only active within the app.

Next, log in to your visual app builder and create a new app. Choose the pages from your site you want in the app. For example, if you want your BuddyPress activity feed in there, you can add the URL to your activity page in the app menu. You also want your LearnDash courses, so you create a custom REST API-based page for that.

You can choose to add a login button to your menu, so users can log in and out, or register on your site. After that you change the colors,

add your icon and splash screens, and build the app. You can preview it in the browser, or the AppPresser Preview app for iOS and Android, or use PhoneGap Build to test on your device.

## COMPILING AND TESTING AN APP

On the Build and Preview tab of the App Builder/theme customizer, you'll need to integrate with the PhoneGap Build API via an authentication token generated from your PhoneGap Build account to send your app to get compiled and built. If you don't have an account, create a free one and then go to Account → Client applications to access your authentication token. Then save your token in the PhoneGap Build Auth Token field, and click the Build App button. This packs up all your AppPresser app files and sends them PhoneGap Build to be compiled. Once PhoneGap build is finished, a QR code and link is displayed; this allows you to download and install an APK file for Android or iOS app if you have the proper signing key and key password. If you don't yet have an Apple developer account or an iOS signing key, you can test your app on iOS using the [AppPresser Preview app](#). This app can also be used for testing your app on an Android device.

Basically, the AppPresser Preview App allows you to log in to AppPresser so you can test it out on your device while you're building it or implementing updates. On the Build and Preview tab, you will also notice a button called Update Live App. Click here to push changes (such as custom CSS or menu item changes made in the App Builder) to an existing installed app. It basically tells the app to fetch new data from your API.

For example, suppose that you have an app on the app store, and it has a blue background. You want to experiment with changing the color to red, but don't want current app users to see this change. Make your changes in the app customizer, and preview the new color. When you're satisfied, click Go Live, and your live app background will then change to red. Any local updates to your app, such as offline content (new splash screens, custom HTML pages, etc.), will have to be recompiled, the app distributed, and then the app updated on installed devices for any changes to take effect.

#### NOTE

App users must close and reopen your app before they will see any changes applied.

## LINKING BETWEEN APP PAGES

Since there are a few different page types, linking from one app page to another can take a few different forms.

To link from a WordPress page to an app page, add a link inside your WordPress iframe page that links to a page in the app. Use any of the following code in any WordPress post, page, or CPT.

1. Link to a page in your side menu:

```
<a href="#" data-applink="2">Menu 2</a>
```

2. Link to a tab page:

```
<button data-apptablink="1">Tab Menu  
1</button>
```

The number in the link refers to the menu item position, starting with 0. For example, the first page in your menu would be 0, the second page 1, and so on. Link from a custom HTML page to another page (must be in the menu). In this case you are creating a custom HTML page and you want to link to another app page.

3. Add a button to the page:

```
<button ion-button>Visit Page</button>
```

4. Add a click function in that button:

```
<button ion-button (click)="pushPage(  
pages.menus.items[0] )">  
Visit Page  
</button>
```

Based on your menu, the page you are linking to will change.

For side menus, use `pages.menus.items[]`. For a tab menu, use `pages.tab_menu.items[]`, as follows:

```
<button ion-button (click)="pushPage(  
pages.tab_menu.items[1] )">  
Visit Page  
</button>
```

The number refers to menu items, starting with 0. The first item is 0, the second is 1, and so on.

If you do not want a back button, use `openPage` instead of `pushPage`:

```
<button ion-button (click)="openPage(  
  pages.menus.items[3] )">  
  Visit Page  
</button>
```

## APPCAMERA PLUGIN

The AppCamera extension allows your app users to take photos that will be uploaded directly to your WordPress site. You can then display those photos in your app. AppCamera also integrates with WooCommerce, BuddyPress, and a shortcode you can use on any post or page. After you install and activate the AppCamera plugin, you'll find a new AppPresser Camera tab in your AppPresser settings. Click this to configure the Camera extension settings.

The AppCamera extension offers multiple settings to help you configure its behavior:

### *Camera license key*

Add the license key you received, with your purchase. This will enable you to receive updates for the plugin as they're released.

### *Uploaded photos must be moderated*

Use this setting if you want to be able to review/approve/deny all uploaded photos before they are published live.

### *Email new photos to admin email*

Use this setting if you want to be notified by email, using the WordPress set admin email address, whenever a user uploads a

new photo, regardless of moderation being required.

#### *Save photos to featured image*

When you enable this and have uploads set to create new posts, the uploaded image will also be set as the post's featured image.

#### *Photo upload description*

Text that will be displayed as a description on the rendered upload form.

#### *Text to display if logged out*

Text that will be displayed to logged-out users.

Once you are happy with the settings, click the Save Settings button.

AppCamera automatically adds a photo upload button to WooCommerce product photo galleries and the BuddyPress activity modal (with the AppBuddy and AppWoo plugins also installed).

You can also manually add the shortcode (described in a moment) to a page or post, which shows a photo upload button in your app. The Take Photo and Upload Image buttons appear only when a user is logged in; otherwise, they will see “Please login” text. It is not secure to let logged-out users upload images to your site, so this is not an option in our plugin. There are a variety of ways to display images uploaded from your app on your app or website (see Figure 16-2).

# Edit Page

Add New

## Photos

Permalink: <https://bwawwp.com/photos/> Edit

Text Editor

Beaver Builder

Add Media

Add Form

b

i

link

b-quote

del

ins

img

ul

ol

li

cc

[app-camera action="this" force\_login="false"]

[gallery]

Figure 16-2. AppPhoto shortcode

Gallery shortcode

You can add a WordPress gallery shortcode `[gallery]`, on the same page where you have the `[app-camera]` shortcode. This will display your images. Note that this page does not refresh dynamically to display newly uploaded images, but you can use JavaScript to do that if needed.

*Action="this"*

If you are using `[app-camera action="this"]`, each photo from the app will be uploaded to the media library and attached to the current post or page where the shortcode is.

*Action="new"*

If you are using `[app-camera action="new"]`, each photo will create a new post. You can then display these posts any way you like, such as a custom template in your theme, or a plugin like Display Posts Shortcode.

You can use the `post_type` attribute to specify a post type that you have already registered, such as `app_photos` to display only these photos and not other posts. This method supports moderation.

The AppCamera extension also comes with some shortcodes:

- `[app-camera action="new"]`
- `[app-camera post_type="page"]`
- `[app-camera post_title="true"]`
- `[app-camera not_logged_in="Please log in first"]`
- `[app-camera description="You can upload your photos!"]`

## NOTE

You only need to use one shortcode at a time, and you can combine multiple parameters. For example:

```
[app-camera post_type="page" force_login="false"  
action="new"]
```

You can choose to moderate images before they are displayed on your site. To moderate photos, go to the AppPresser settings page, click the Camera tab, select the “Uploaded photos must be moderated?” checkbox, and then save. If you have selected the option that uploads need to be moderated before being added, the section for this will not appear until there are actually photos to moderate. Once moderation-pending photos are available, you will see a new Moderate Photos submenu item in the AppPresser menu. Next to it will be a notification of how many are pending approval. You will also be able to access this page from a Photo Moderation Panel link that will show up next to the Save Settings button in the AppCamera tab.

When viewing the moderation page (shown in Figure 16-3), you’ll see a list of pending photos that you can view, approve, and deny. A thumbnail preview, filename/type, uploader, post it was uploaded to, and date will all also be available. Checkboxes along the left allow you to approve or deny in bulk, and each photo has its own approve or deny links. Once the queue is empty, the menu item and link will disappear until there are more to take care of.

## AppPresser Photo Moderation

Check the photos you would like to Approve or Deny.

	File	Author	Attached to
<input type="checkbox"/>	 103058cdv_photo_006 JPG	appresser	Photos

*Figure 16-3. Moderating uploads*

AppCamera also comes packed with many WordPress hooks you can use for your own app customizations:

*appp\_after\_camera\_buttons*

Hooks immediately after the form for the photo upload has displayed.

*appp\_after\_process\_uploads*

Hooks immediately after the photo upload handling is done.  
Passes in the affected post ID and the new attachment ID from the

upload. This allows you to do more processing on either post or attachment afterwards.

#### *appp\_before\_camera\_buttons*

Hooks immediately before the form where the photo upload is displayed.

#### *active\_plugins*

Allows you to intercept `get_option( 'active_plugins' )` before your extension determines whether the required plugins are presently active. Default value: `get_option( 'active_plugins' )`.

#### *appp\_camera\_description*

Allows you to intercept and modify, if needed, the text to display as the description. This value is originally set in the Camera section of the AppPresser settings page. Default value: *Upload your photos!*

#### *appp\_camera\_not\_logged\_in\_text*

Allows you to intercept and modify, if needed, the text to display to not-logged-in users. This value is originally set in the Camera section of the AppPresser settings page. Default value: *Upload your own customer image!*

#### *appp\_camera\_post\_title\_label*

Allows you to intercept and modify the label for the `post_title` field that is displayed via the [app-camera] shortcode.

Default value: `<label> . __( 'Title:', 'appp' ) . </label>`

#### *appp\_upload\_email\_message*

Allows you to intercept and modify the default message that is set with the email notifications for new uploads.

#### `app_p_upload_email_subject`

Allows you to intercept and modify the default subject that gets set with the email notifications for new uploads. Default value: *A new photo was uploaded.*

#### `app_p_upload_email_to`

Allows you to intercept and modify the default email recipient who will be receiving the email notifications for new uploads. Default value: `get_settings( admin_email )`.

#### `app_p_camera_photo_blog_embedded_img_size`

Allows you to intercept and modify the default size for photo blog image embeds. Default value: `array( 768, 2500, false )`. The first parameter is width in pixels, the second parameter is height in pixels, and the third parameter is whether to crop.

#### `app_p_moderate_maybe_publish`

Allows you to intercept and modify, if needed, the array of arguments used for the publishing of the post once a photo is approved from moderation. Default value: `array( ID => absint( $parent[0] ), post_status => publish ) ;`

#### `app_p_insert_photo_post`

Places the uploaded image into the post content area, if a user has chosen not to have the image applied to the featured image.

## WOOCOMMERCE PLUGINS

AppWoo has been developed specifically to integrate WooCommerce into your app, along with the AppPresser theme. To create your

WooCommerce pages in your app, visit your app customizer menu. Click “add items,” choose “WordPress/external links,” and then add your shop page URL and save. You can also add other pages like account and custom product category pages.

AppWoo is fairly self-contained and does not have any user-configurable settings. However, it does the following under the hood to help optimize AppTheme with WooCommerce for your native app:

- Shapes the page layout for single products to fit nicely with app-like structure. It accomplishes this by removing various WooCommerce hook callbacks as well as registering some new ones for AppTheme to achieve the desired layout.
- Provides integration with AppSwiper and helps with Ajax and sliders for product images. This way you can more easily display multiple product images and variety for your customers.
- If you have AppCamera active, it will automatically add the ability for users to upload and submit images to each product.
- Provides a user profile in the left panel menu for your returning customers. When they toggle the menu, they’ll see their name and profile picture.
- Provides cart information and totals in the left panel menu for your customer. Provide easy access to your customer’s information and orders, directly from the menu.

There are several AppWoo hooks that you can use from within your own custom WordPress plugin:

*app\_after\_product\_images*

Hooks right after the product images gallery display.

*active\_plugins*

Allows you to intercept `get_option( 'active_plugins' )` before the WooCommerce extension determines if WooCommerce is presently active. Default value: `get_option( 'active_plugins' ).`

*appresser\_woocom\_gallery\_ids*

Allows you to intercept and modify, if needed, an array of gallery image IDs used with the gallery display. Default value: `$gallery_ids.`

*appresser\_disable\_woo\_styles*

Allows you to intercept and prevent the disabling of WooCommerce styles. Default value: `true` //Styles will be disabled.

If you are using WooCommerce in your app, a cart icon will automatically appear on single product pages in your app. If you'd like it to appear on every page, you can add the following code to a custom JavaScript file in a plugin or child theme on your WordPress site:

```
jQuery(document).ready(function($) {
    if( $('body').hasClass('woocommerce-page') ) {
        var message_array = {};
        message_array['post_title'] =
window.app.post_title;
        message_array['cart_link'] =
window.appwoo.cart_url;
        parent.postMessage( JSON.stringify( message_array
```

```
) , ' * ' ) ;  
}  
});
```

(Note that this is only an example; you might need to edit this script to work properly for your site.)

With huge updates to WordPress and WooCommerce over the past few years, AppPresser built a new WooCommerce mobile app integration called *AppCommerce*. It's faster, cleaner, and more focused on providing your customers a great experience in your store. AppCommerce connects to your WooCommerce store through the WooCommerce REST API. Your customers can add products to their cart, log in, view past orders and account details, create a wish list, and more. Checkout happens on your website through the in-app browser. It's seamless, and it allows you to use your current payment gateways, shipping, and taxes with no further configuration. If you want, you can fully customize the checkout experience to differ from the website. After payment, your customers are effortlessly directed back to the app. AppCommerce is all fully customizable using the AppPresser app builder. You choose what pages go in the app, what products are displayed, what colors to use, and lots more.

To add your WooCommerce shop, cart, and account pages to your app, visit your app customizer, and go to Custom Pages. There, you need to click Add New Page, and choose AppCommerce.

For a shop page or any app page you would like to add a product list on, all that's required is the `woo-list` tag, and selecting Shop under

the component. Use the following code to show a product list on your custom app page:

```
<woo-list route="products" infiniteScroll="true"></woo-list>
```

To customize the products or product categories displayed, use any URL parameters found in the [WooCommerce REST API documentation](#). For example, you can add "products? type=simple" to show only simple products, or out-of-stock products by adding "products? stock\_status=outofstock":

```
<woo-list route="products?stock_status=outofstock" infiniteScroll="true">
</woo-list>
```

### NOTE

If you used the AppCommerce app template, you may already have these pages. If so, you do not need to create them again.

For a dedicated cart page on your app, you are required to have the `woo-cart` tag somewhere on the page, and Cart selected as the component:

```
<woo-cart></woo-cart>
```

For the account page use the following:

```
<woo-account></woo-account>
```

## NOTE

Remember that a shop, cart, and account page are all required app pages.

You can add other WooCommerce pages to your app, such as a wish list. Customers can click “Add to list” on a product, which saves to their wish list. To create a wish list page, create a new custom page and choose AppCommerce. Set the component to “None” and add the following component code:

```
<woo-list wishlist="true"></woo-list>
```

Checkout happens on your website through the in-app browser. We recommend using the WooCommerce Smart Checkout plugin for the best experience.

## NOTE

Almost 40% of all ecommerce purchases during the 2018 holiday season were made on a smartphone.

## LEARNDASH/APPLMS

The LearnDash integration allows you to add courses, topics, lessons, and quizzes to your app. You can restrict content based on login status, and incorporate a membership plugin, BuddyPress, gamification, and more. To add LearnDash to your app, just create a page that displays your courses. The rest is taken care of. To add courses to your app, create a new custom page in your app customizer

under Custom pages → Add new page to be your course listing page.  
Add the following code:

```
<ap-list wp="true" card="true" class="col-2"  
    route="https://mysite.com/wp-json/wp/v2/sfwd-courses?  
per_page=10">  
    </ap-list>
```

Modify the `per_page` parameter to show any number of courses you like. This will display your courses in a two-column card layout. You can also modify the `ap-list` component to display them in a single column list like this:

```
<ap-list wp="true" route="https://mysite.com/wp-json/wp/  
v2/sfwd-courses" infiniteScroll="true"></ap-list>
```

Add any other text or HTML code you like and then save. Next, add this page to your menu and rebuild your app (Figure 16-4).

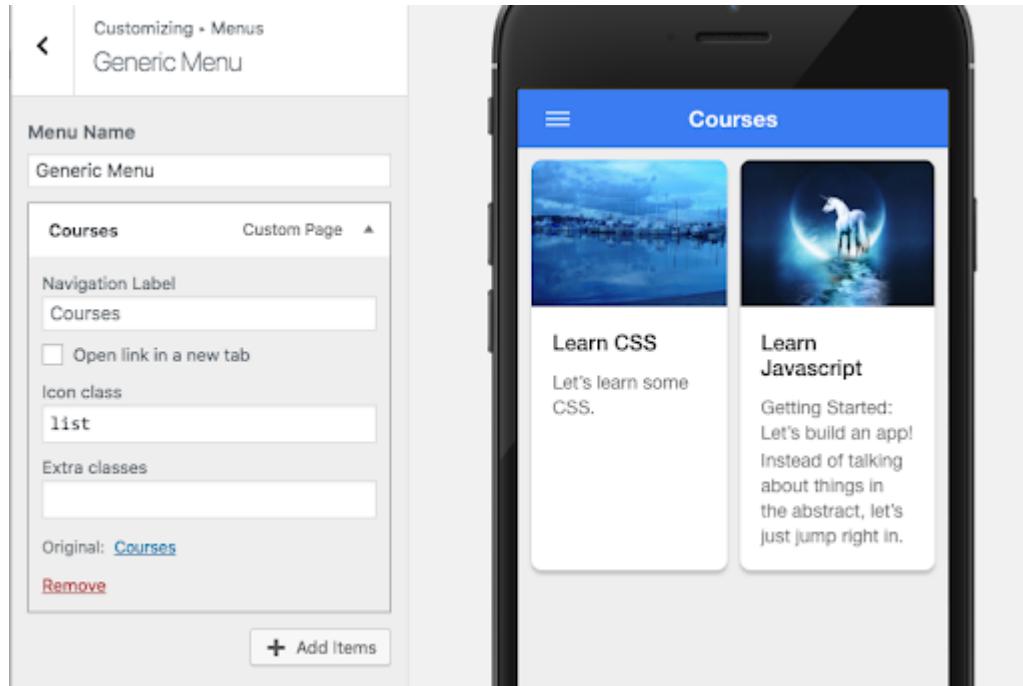


Figure 16-4. Adding courses to your app

If you don't like this API-based approach, or need more customization, you can use a WordPress-based approach instead. Create a new page on your WordPress site and add the LearnDash course list shortcode. If you are a developer, or someone looking to really dig into deep customization, you can build whatever add-on plugin you need to extend this functionality however you want.

## APPPUSH

*Push notifications* send messages to any device that has installed your app and allows push notifications. AppPresser comes with push notifications. They require a bit of setup, so follow the instructions very carefully. A few things to note:

- You cannot test push notifications in a simulator or on the AppPresser Preview app; you must use an actual device.
- Most problems derive from iOS certificates, so make sure to follow the instructions exactly.
- You must have a valid iOS developer account for Apple. You can find more information at [Apple Developer](#).
- Android requires a free Firebase project and API key.

Once you set up your iOS certificates and Firebase project, you can add this information to *myappresser.com* and PhoneGap Build in order to send push notifications from either inside My AppPresser or via the AppPush plugin installed and integrated into your WordPress website.

## **NOTE**

If you would like to learn more about AppPresser and its add-on plugins listed above please refer to our [AppPresser site](#).

# Chapter 17. PHP Libraries, Web Service Integrations, and Platform Migrations

---

At times you may need to integrate features, functionality, and data with WordPress that are not WordPress-specific features, functionality, or data.

In this chapter we review some popular PHP libraries, web services, and APIs that can easily be integrated with WordPress. We also cover a structured process that can be utilized to completely move content/data from just about any third-party software into WordPress, which is commonly referred to as a *platform migration*. Move all the things to WordPress!

## PHP Libraries

Most programming languages, including PHP, have modularized collections of code, classes, and functions. These collections are usually referred to as a *code library* or *extension*.

Don't re-create the wheel! There are PHP libraries that exist to perform very specific functionality that you can easily use to extend the applications you're building.

Lots of WordPress is written in PHP, which means that as a WordPress developer, you can utilize any PHP libraries available to you or even build your own. Check out the [PHP.net Function Reference](#) for an organized list of some common PHP libraries.

The WordPress plugin repository has more than a few plugins that interact with various libraries and external services, so any time you are looking for custom WordPress functionality, you might want to begin at the WordPress plugin repository. Of course, if you don't find exactly what you are looking for in a plugin, you may need to write your own custom plugin or piggyback off of an existing one.

We've mentioned GitHub more than a few times, but it is a great place to find the functionality you're looking for. Another good resource for code can be other PHP-based open source projects like Drupal, Laravel, Joomla, and/or Magento. Maybe the functionality you need doesn't exist as a WordPress plugin but does exist as a Drupal module: take the code and make it work with WordPress.

Let's review some cool PHP libraries that offer extended functionality and that you can easily integrate with WordPress.

### NOTE

Depending on your web-hosting environment, you might not be able to use some PHP libraries. Some PHP libraries are wrappers for their underlying software, which needs to be installed on your web server. If you do not have access to configure your web server, you will not be able to use them.

## Image Generation and Manipulation

Think of Photoshop for your code, which can be useful for building things like mindless meme generators, among other more productive things. Image generation and manipulation can be very useful for a number of things:

- Resizing and scaling images
- Comparing images
- Copying images
- Converting images to various image formats
- Changing colors within an image
- Converting text to images and overlaying text on images
- Combining multiple images
- Creating animated images
- Applying custom image filters (yes, just like Instagram)
- Blurring and sharpening images
- Adding borders and frames to images
- Dynamically creating charts or graph images based on data
- Rendering 3D images

As you can imagine, there are many use cases for these capabilities given that you can pretty much do anything with any images programmatically. Some PHP image-manipulation libraries are better than others depending on what you are using them for.

By default, WordPress uses one of the two most popular PHP image-manipulation libraries to power image-manipulation features of WordPress and in the media library. First, WordPress checks whether the Imagick library is available; if not, then it checks to see if the GD library is available. If neither is available, you get an error message when you try to use WordPress image-manipulation functions. When WordPress crops an image, rotates an image, or does anything to manipulate an image, it is using one of these two libraries in the background.

You can easily check if Imagick or GD is being used on your WordPress installation by executing the following code in your theme *functions.php* file:

```
<?php
// Don't do this on a production site that you care about.
function bwawp_image_library_check() {
    if( extension_loaded('imagick') ) {
        $imagick = new Imagick();
        print_r( $imagick->queryFormats() );
    } else {
        echo 'No ImageMagick!';
    }

    if( extension_loaded('gd') ) {
        print_r( gd_info() );
    } else {
        echo 'No GD!';
    }
    exit();
}
add_action("init", "bwawp_image_library_check");
```

Depending on which library is being used, you could easily start writing code to use that library.

## GD

GD is probably the most popular image-manipulation PHP library and usually comes compiled with default installations of PHP. GD tends to be a little bit faster than Imagick when dealing with larger images and it is generally less of a server resource hog. GD is a great tool and can get you where you need to go most of the time, but Imagick has more features and functionality.

## IMAGICK

One of the most powerful image-manipulation tools and our personal favorite—though we wish someone would update its website to use WordPress—Imagick can be used to manipulate image files programmatically in just about any way you can imagine. You can install Imagick on your server and run it through the command line via the `shell_exec()` or `exec()` PHP functions, or you can use the Imagick PHP library as a wrapper for the underlying software. The Imagick library is not bundled with PHP by default and you must install it separately along with the Imagick software itself.

Justin Sternberg made some really easy-to-use methods using Imagick for overlaying any text on any image and then saving the separate image as an attachment against a WordPress post. This can be very useful if you require all the images on your web application to be watermarked with your URL, so if anybody hijacks your

images, at least they have your web address embedded into them. Check out Justin's code at [WordPress-Image-Watermark](#).

## ZEBRA\_IMAGE

[Zebra\\_Image](#) is a super-lightweight and easy-to-use PHP library for image manipulation. This is a great alternative to Imagick, if Imagick is overkill, you can't install Imagick on your web server, or you only need to perform some basic image-manipulation tasks like resizing, cropping, rotating, flipping, and/or sharpening images. This single-file PHP library requires only the GD extension, which usually comes precompiled with PHP.

## IMAGINE

Somewhere between Imagick and Zebra\_Image is probably [Imagine](#). This object-oriented PHP library is very powerful, and you can use it to do almost any kind of image manipulation. Like Zebra\_Image, Imagine is also a GD-based PHP library.

## DYNAMIC DUMMY IMAGE GENERATOR

The [Dynamic Dummy Image Generator](#) isn't a PHP library, but it's a great little tool that we wanted to mention that can help your WordPress website development efforts. You know when your client never provided you with the images they promised you three weeks ago? Using [Dummy Image](#), you can dynamically create any sized images from a URL.

## SNAPPY

Now this is cool! With [Snappy](#), you can automatically create an image and/or a PDF from any URL or HTML page. Think of all the cool things you can do by autogenerating an image of any website you want.

## PDF Generation

Programmatic PDF document generation and manipulation can be very useful. For instance, you may have been to a school or municipal website that has a ton of PDF documents with content (lunch menus, calendar, upcoming events, event registration, reading lists, meeting agendas/minutes, handbooks, reports, guides, etc.) that should probably be on the actual website and not in a PDF document. One key reason schools and municipalities use PDFs (besides “this is how we’ve always done it”) is that PDFs are easy to physically distribute, via email lists or hard copy. Although physical distribution can be a good thing, there are some disadvantages to having your content only in PDF format:

- PDFs are not ADA compliant. This can open organizations to a lawsuit if they don’t make that content accessible.
- PDFs are not mobile friendly. Even though you can see them on a mobile device, it is like looking at a nonresponsive website, one that you have to pinch and zoom to see what you want.
- If you do have your content on both a web page and a PDF and need to make an update, you have to do it twice.

What if every web page on your website needed to have a downloadable PDF of it available? What if you have a bunch of

online stats and reporting data from which you need to generate PDF documents? What if a calendar of events is constantly being updated, and you need to manually update a PDF and distribute it each time it changed? We can solve these and similar problems with code to streamline processes and save a ton of time and money. Let's talk about some PHP libraries we like to use for automatically generating PDFs.

## SNAPPY

We mentioned [Snappy](#) earlier, but we wanted to reiterate how cool and easy it is to generate PDFs of a web page via a provided URL or HTML. You can even generate a multiple-page PDF document by passing in multiple URLs. Want a detailed PDF document of your entire website? Just loop all of your pages and pass them into Snappy!

Snappy uses two popular open source HTML-to-PDF and HTML-to-image command-line tools: *wkhtmltopdf* and *wkhtmltoimage*. You can download and install the necessary binary files and easily install them on your server with [Composer: A Dependency Manager for PHP](#).

## FPDF

The F in [FPDF](#) stands for “free”! This is a great lightweight alternative to [PDFlib](#) if you are not generating too many PDF documents and they are not too complex. You can use FPDF to generate dynamic PDF documents with just PHP: it does not depend on other PHP libraries except for Zlib to enable compression and GD

for GIF support, which both usually come precompiled with PHP.  
Some of the main functions of FDPF include the following:

*\_\_construct*

Constructor

*AcceptPageBreak*

Accept or not automatic page break

*AddFont*

Add a new font

*AddLink*

Create an internal link

*AddPage*

Add a new page

*AliasNbPages*

Define an alias for number of pages

*Cell*

Print a cell

*Close*

Terminate the document

*Error*

Fatal error

*Footer*

Page footer

*GetPageHeight*

Get current page height

*GetPageWidth*

Get current page width

*GetStringWidth*

Compute string length

*GetX*

Get current x position

*GetY*

Get current y position

*Header*

Page header

*Image*

Output an image

*Line*

Draw a line

*Link*

Put a link

*Ln*

Line break

*MultiCell*

Print text with line breaks

*Output*

Save or send the document

*PageNo*

Page number

*Rect*

Draw a rectangle

*SetAuthor*

Set the document author

*SetAutoPageBreak*

Set the automatic page-breaking mode

*SetCompression*

Turn compression on or off

*SetCreator*

Set document creator

*SetDisplayMode*

Set display mode

*SetDrawColor*

Set drawing color

*SetFillColor*

Set filling color

*SetFont*

Set font

*SetFontSize*

Set font size

*SetKeywords*

Associate keywords with document

*SetLeftMargin*

Set left margin

*SetLineWidth*

Set line width

*SetLink*

Set internal link destination

*SetMargins*

Set margins

*SetRightMargin*

Set right margin

*SetSubject*

Set document subject

*SetTextColor*

Set text color

*SetTitle*

Set document title

*SetTopMargin*

Set top margin

*SetX*

Set current x position

*SetXY*

Set current x and y positions

*SetY*

Set current y position and optionally reset x

*Text*

Print a string

*Write*

Print flowing text

## OTHER PDF-GENERATION PHP LIBRARIES

Depending on your specific needs you may want to do a Google search for some of these and look them over in more detail:

- wkhtmltopdf
- mPDF
- Dompdf
- TCPDF
- HTML2PdF

## Geolocation and Geotargeting

There are many geolocation services available that do different things, like telling you the approximate location of a provided IP address or an approximate longitude and latitude based on a provided

GPS location or WiFi network. If you have access to your visitors' location, you can do so many cool things, including the following:

- Restrict access to websites or specific web pages based on location.
- Serve content on a website based on location like geotargeted ads.
- Track a person or item location in near real time.
- Find the closest location of something based on a provided location.

Maybe a school wants to add an extra layer of security for its web application and wants to give access to the login page to only people from within the same town or state as the school. This type of security feature might be necessary only if you want to lock down your application to a particular geographical location or locations, but it could greatly reduce the amount of potential hacking attempts. Instead of only allowing access to your web application, you could also lock out particular locations—for example, China. Maybe depending on your application, you know that anyone visiting from China should not have access.

Maybe a state website using WordPress wants to be able to show its visitors schools in the area when the website first loads. Suppose that “schools” is a CPT with address information stored as post meta. You also can achieve this by getting the end users’ location and doing a meta query to match schools in their area.

## **MAXMIND GEOIP**

MaxMind GeoIP gets data from a user's IP address, such as their location and internet provider. MaxMind is a great service because it has a very extensive API and a free downloadable database.

You can download the PHP library with all of the functionality from the MaxMind website. To utilize its API, download and install the code from GitHub.

## GEOCODER PHP

Geocoder PHP is a PHP library that will assist you in building geo-aware web and mobile applications. Unlike MaxMind, which is its own geolocation provider, Geocoder PHP allows you to choose from multiple geolocation providers, including MaxMind. This PHP library makes calling and fetching geolocation data from various geolocation providers streamlined and consistent. To use the Geocoder PHP library, you need to choose a provider and an HTTP client/adapter. The following providers are currently available to this PHP library, and you can add your own if need be:

- Algolia Places
- ArcGIS Online
- Bing Maps
- FreeGeoIp
- Geocode Earth
- GeoIP
- GeoIP2
- Geonames

- GeoPlugin
- Google Maps
- Google Maps Places
- GraphHopper
- Here
- HostIp
- IpInfo
- IpInfoDB
- ipstack
- LocationIQ
- Mapbox
- MapQuest
- MaxMind
- MaxMind Binary
- Nominatim
- OpenCage
- Pelias
- Photon
- PickPoint
- TomTom
- Yandex

## WEB-HOSTING GEOLOCATION

Some web hosts have geotargeting services baked into their platforms to offer you geolocation information like a site visitor's continent, country, state/region, or city. You can use this data to display different content to visitors based on their location.

For instance, on top of WP Engine's hosting services, it has an additional geolocation service that integrates with its platform's custom WordPress caching system via a custom WordPress plugin. It stores cached visitor information in buckets that can be accessed via preset PHP variables and WordPress shortcodes. This is probably one of the fastest ways to integrate geolocation data with your website, but you must already be hosted with WP Engine.

## File Compression and Archiving

You can imagine several use cases for being able to zip and unzip files programmatically, but most of the time you'll be doing either of these two things:

- Automatically unzip a single compressed ZIP file of multiple resources.
- Automatically zip multiple resources into a single compressed ZIP file.

WordPress core uses file decompression when you install a plugin. When you upload your own plugin or install one from the WordPress plugin repository, WordPress locates the ZIP file and unzips it into the *wp-content/plugins* directory on your WordPress installation.

At AppPresser.com, we depend on programmatically zipping and unzipping files heavily within our platform. In our App Builder UI, we offer file upload fields in which a user can upload a ZIP file of all their mobile app image resources that they could use locally within their mobile app. Our server-side PHP code receives the ZIP file, decompresses it, and moves all the files into a local directory within that app project directory. When an app is ready to be compiled, we automatically zip up all the source files in that app project directory and send it to PhoneGap Build to be compiled. The end result is all of the image resources originally uploaded in a single ZIP file are loaded locally from within the compiled mobile app.

We once worked on a large platform migration from a proprietary CMS to a WordPress Multisite network. The sites that we were migrating from were powered by individual MySQL databases. We were provided with an SFTP account that had all the websites to be migrated as individual ZIP files. Each ZIP file had a *.sql* export of that website's database along with all of the source files and resources for that individual site. There were more than 8,000 sites to migrate, meaning more than 8,000 ZIP files to decompress and run custom migration scripts on to move all the data to WordPress. We were able to automate the entire migration process by programmatically looping through all the ZIP files in that SFTP account and unzipping them with [PHP ZipArchive](#).

The PHP ZipArchive class has a ton of functionality to help you compress and decompress any of your files, and it's usually baked into PHP by default. You can either compress files one at a time or just point to an entire directory to be compressed at once. The

`ZipArchive::open` method is the main function for opening a new ZIP archive for reading, writing, or modifying. It accepts a `filename` parameter, which is a directory path and filename of the ZIP archive to open, and an optional `flag` parameter that accepts one of four modes:

#### *OVERWRITE*

Overwrites the files in the specified archive if they already exists.

#### *CREATE*

Creates a new archive if it does not already exist.

#### *EXCL*

Used to check whether an archive already exists.

#### *CHECKCONS*

Used to perform additional consistency checks on the archive and give an error if they fail.

## **COMPRESSING INDIVIDUAL FILES INTO AN ARCHIVE**

Here is some basic code for adding files to a ZIP archive that you create one file at a time. As you can imagine, you could write some code to loop through any files you want, adding them to your new ZIP file however you want:

```
<?php  
  
// First create a ZipArchive Instance  
$zip = new ZipArchive();  
  
// Use the open() method to create a zip file on the server  
$zip->open( 'compressed-dir-path/messenlehner-kids.zip',
```

```

ZipArchive::CREATE ) ;

// Use the addFile() method to add a file to your zip
$zip->addFile( 'some-dir-path/index.html', 'index.html' );

// You can also add files to subdirectories within the zip
$zip->addFile( 'some-dir-path/dalya.png', 'images/dalya.png'
);
$zip->addFile( 'some-dir-path/brian.png', 'images/brian.png'
);
$zip->addFile( 'some-dir-path/nina.mp4', 'videos/nina.png'
);

// You can also change a file name while you are adding it
// to the zip
$zip->addFile( 'some-dir-path/CWM.png', 'images/cam.png' );
$zip->addFile( 'some-dir-path/babyA.png', 'images/aksel.png'
);

// Call the close() method to finish and save your new zip
$zip->close();

?>

```

## COMPRESSING MULTIPLE FILES INTO AN ARCHIVE

ZipArchive has functions for retrieving various files from provided directory paths so you can add files to your ZIP archive file in bulk instead of one by one:

### *addGlob()*

This method allows you to identify files of defined file types.

### *addPattern()* -

This method allows you to use regular expressions to identify files in a directory based on any pattern in their filename.

Following is an example using both the `addGlob()` and `addPattern()` methods to add all `.png` and `.jpg` images and `.mp4` videos from provided directories into a ZIP file:

```
<?php

// First create a ZipArchive Instance
$zip = new ZipArchive();

// Use the open() method to create a zip file on the server
$zip->open( 'compressed-dir-path/messenlehner-kids.zip',
ZipArchive::CREATE );

// Set options array to add the directory 'videos' to the
// zip file
$options = array( 'add_path' => 'videos/', 'remove_all_path'
=> TRUE );

// Add 'mp4' files from a given directory to the 'videos'
// directory in the
// zip file
$zip->addGlob( 'some-dir-path/*.mp4', 0, $options );

// Set options array to add the directory 'images' to the
// zip file
$options = array( 'add_path' => 'images/', 'remove_all_path'
=> TRUE );

// Add both 'jpg' and 'png' files from a given directory to
// the 'images'
// directory in the zip file
$zip->addGlob( 'some-other-dir-path/*.{jpg, png}',
GLOB_BRACE, $options );

// You can also use regular expressions to get any files
// from a given directory
$zip->addPattern( '/\.(?:jpg|png)$/', 'another-dir-path',
$options );

// Call the close() method to finish and save your new zip
$zip->close();
```

```
?>
```

## EXTRACTING FILES FROM AN ARCHIVE

To automatically unzip a compressed archive file, you can use the `extractTo()` method, which accepts two parameters:

*destination*

Where the files in the ZIP are being unzipped and saved to.

*entries*

The files you want to extract from the ZIP. You can grab one file at a time, an array of files, or, if you want to extract all the files in the ZIP, you can exclude this parameter:

```
<?php

    // How to extract files from a zip file on your server
    $zip = new ZipArchive();
    $zip->open( 'compressed-dir-path/messenlehner-kids.zip',
        ZipArchive::CREATE );

    // Use the extractTo() method to extract files in a zip
    // on the server
    // Extracts one file
    $zip->extractTo( 'uncompressed-dir-path/',
        'images/aksel.png' );

    // Extracts files in an array
    $zip->extractTo( 'uncompressed-dir-path/',
        array( 'images/cam.png', 'images/nina.png' ) );

    // Extracts all files in the zip
    $zip->extractTo( 'uncompressed-dir-path/' );

    $zip->close();

?>
```

## OTHER COMPRESSION AND ARCHIVE PHP LIBRARIES

By default WordPress depends on ZipArchive for any file compression/decompression. WordPress actually first checks to see whether ZipArchive is available; if not, it falls back to another popular PHP library called *PclZip*. There are also some other compression and archive PHP libraries that you can use if you like:

- [PclZip](#)
- [Bzip2](#)
- [LZF](#)
- [Phar](#)
- [Rar](#)
- [Zlib](#)

## Developer Tools

Following are some PHP libraries to make your life as a developer just a little bit easier. There are a ton of great tools out there, but here are a few goodies.

## PHPUNIT

If you haven't heard about *unit testing*, you should look into it. It's a great method for regression testing your code and for automatically locating bugs/errors within your code. Think of it as writing code to automate the testing of your code. With [\*PHPUnit\*](#) you can set up specific unit tests to be performed on particular functions within your code to verify that those functions are doing everything they are

supposed to be doing. We can't iterate enough how important unit testing can be when you have multiple hands in the cookie jar. Give your clients and yourself more peace of mind by running unit tests before deploying any code updates to a production environment.

### TIP

You can and should charge extra for incorporating unit testing into your development process.

## PHPDOCUMENTOR

Every good developer knows that they should always provide good comments and additional documentation so any other developers looking at their code can easily figure out what is going on. This practice is also good for you if you're looking at old code that you maybe wrote years ago, amiright? At the very least you should always leave good comments in your code. [phpDocumentor](#) is a great tool for automatically generating documentation from your code comments.

## FAKER

[Faker](#) is a great PHP library for automatically generating dummy content that can be used to populate a database, populate fake WordPress users, display fake content on a website you are building before you have real content, generating an XML feed, populating custom API endpoints, unit testing, or whatever you want to use it for. You can use the following formatters to make fake data, or you

can create your own formatters using functions already defined in the Base class:

- Base
- Lorem Ipsum Text
- Person
- Address
- Phone Number
- Company Real
- Text Date and Time
- Internet User Agent
- Payment
- Color
- File
- Image
- Uuid
- Barcode
- Miscellaneous Biased
- Html Lorem

## **GOUTTE, A PHP-BASED WEBSITE SCRAPER**

Don't have access to a website data source and need to migrate it to WordPress? Need product data from multiple websites with no data feeds? Page scraping can be a great solution to some of your problems, but it can also be very time-consuming to

programmatically find the patterns you are looking for within a web page (hopefully there is some consistency) and then parse that web page HTML markup based on the patterns to extract the data you need. If you are scraping content from multiple web pages, this can be a very daunting task.

Goutte is an awesome tool for screen scraping just about any data from any provided HTML markup, and you also can use it to parse XML. Goutte makes it easy for you to parse or crawl the data returned from an HTTP request using predefined methods that will easily assist you in identifying and extracting any HTML elements. Here are some of the cool things you can easily do with Goutte:

- Extract links from content.
- Extract images from content.
- Extract titles out of heading tags.
- Extract content out of `<div>` tags by their ID or class name.
- Extract sibling, parent, or child elements of an ordered or unordered list.
- Extract any content from within any HTML tags.
- Extract specific nodes from XML.
- Automatically click links.
- Automatically submit web forms.
- Automatically log into website (if you have a correct username and password).

## WHOOPS, FOR USER-FRIENDLY PHP ERRORS

Whoops is a PHP library available as a Composer package that allows you to handle errors and exceptions in your code more elegantly. You can completely customize your PHP errors to include more helpful information than just the default PHP error message, which can sometimes be difficult to decipher. This is a great tool for managing and displaying errors to aid you in your troubleshooting efforts.

## External APIs and Web Services

An API can be part of an application in which you have access to the source code, like the various WordPress APIs covered in this book. But an API can also enable you to use an application or service in which you don't have direct access to the source code. Many web-based applications and services offer some kind of API for accessing or manipulating their data from your custom application.

Let's review some of our favorite web service APIs that can easily be integrated with WordPress.

### Elasticsearch

That's Elasticsearch, Logstash, Kibana, and Beats (also known as the *ELK Stack*). Reliably and securely take data from any source, in any format, then search, analyze, and visualize it in real time.

### ElasticPress by 10up

ElasticPress by 10up automatically integrates with Elasticsearch to index your WordPress database and to provide rich WordPress search

features to use in your web application.

## Google Vision

This is super cool, but can be a little scary at the same time. The [Google Vision API](#) incorporates machine learning to analyze provided images and tries to identify the probability of what is portrayed in that image.

## Google Maps

Everyone knows what Google Maps is, and we're sure you have seen some WordPress plugins using it. Google offers a [JavaScript-based API](#) for interacting with its maps, and you can build anything from a basic map zoomed into a specified location to a custom map with markers depicting various WordPress posts from a specific CPT. The sky's the limit! Actually, in this case, the Earth is the limit.

Besides all of the map functionality available via the Google Maps JavaScript API, Google also has some other API services related to its maps.

## DIRECTIONS

You can use this service to calculate directions between various locations and return step-by-step directions of what route(s) to follow. When creating a directions request, you can pass in an origin or a location for your directions to start, and a destination or the end location. You can also specify the travel mode: driving (default), bicycling, transit, or walking.

## DISTANCE MATRIX

The Google Distance Matrix API is a service that provides travel distance and time for a matrix of origins and destinations. The information returned is based on the recommended route between the start and endpoints, as calculated by the Google Maps API, and consists of rows containing duration and distance values for each pair.

## ELEVATION

This API provides a way to query elevation data for provided locations.

## GEOCODING

This API allows you to query geolocation data like latitude and longitude from a provided address. You can also use reverse geocoding to provide the closest address for a given latitude and longitude.

## STREET VIEW SERVICE

This API allows you to interact with Google Street View. This API is really cool because you can access all of the photos and locations available in Google Street View.

## GOOGLE MAPS WORDPRESS PLUGINS

The following worthwhile WordPress plugins already utilize the Google Maps service:

*WP Google Maps*

One of the most popular WordPress plugins using Google Maps that allows you to easily build maps on your WordPress posts using shortcodes.

### Gutenberg Map Block for Google Maps

If you are using Gutenberg and want to offer your users an easy-to-use Gutenberg Block for managing Google Maps, this plugin is your answer.

### gPano

Easy-to-use WordPress plugin for embedding Google Street View panorama images into a post or page.

### WP Store Locator

Easy-to-use WordPress plugin for adding physical locations with custom markers and information to maps. Locations can be searched by name and metadata as well as by radius to a given location. This plugin also leverages Google Maps Directions service to provide directions to a location.

## **Google Translate**

What if you want to auto-translate pieces of custom content into various languages? You can do this utilizing Google's Translation API. There are quite a few WordPress plugins that leverage this API, but depending on your scenario, you may want to translate various strings of text on the fly, and you can certainly use this API to do so. Google also incorporates machine learning into this service with what it calls *AutoML Translation*. This API is not free but can be well worth the cost depending on how important it is to you to automatically translate any of your content.

[Translate WordPress with GTranslate](#) is a great plugin to showcase some of the abilities the Google Translate service offers when integrated with WordPress.

## Twilio

Need to be able to send customized SMS alert messages to your web app members? Maybe your application needs an additional layer of security and you need to verify user accounts via random activation codes text messages to mobile phones. [Twilio](#) has a great web service that lets you send and receive SMS messages between users' cell phones and your account(s) with them.

Maybe a school needs to verify parents' contact numbers so they can later send text message alerts with import information, like if their child is playing hooky and has cut a class. The first thing you should do is [download the Twilio PHP library from GitHub](#). If you are building a custom plugin, drop *Twilio.php* and the Twilio directory into a lib directory in your plugin. Your directory structure should look something like */wp-content/plugins/<your-plugin>/lib/Twilio*.

### TIP

We recommend interacting with the Twilio API over SSL.

## Other Popular APIs

We have just mentioned a few available PHP libraries, web services, and APIs that you could use, depending on what you are trying to

accomplish. These are some of the more popular ones, but they are really just the tip of the iceberg of what's available across the internet publicly and privately.

Remember, don't reinvent the wheel. Before you build anything, look to see what resources are available to you. Use data from external data sources and integrate your data with external social networks and directories. Work smarter, not harder!

Following are some other popular web services you might want to mess around with:

#### *Google APIs Explorer*

A list of Google web services with APIs that include YouTube, Drive, Calendar, Gmail, Analytics, Hangouts, and Firebase, to name a few.

#### *Instagram Graph API*

Use this API to build apps that allow Business and Creator accounts to programmatically manage their photos, videos, stories, albums, comments, and hashtags.

#### *Salesforce API Explorer*

A list of available Salesforce services with APIs for Marketing Cloud, Einstein Prediction, Force.com, and Data.com, to name a few.

#### *Flickr API*

API for programmatically managing Flickr photos and content.

#### *eBay API*

eBay has a few APIs available for programmatically searching, buying, selling, and negotiating, and about anything else you can do as a buyer or seller.

### *Dropbox API*

Manage your Dropbox accounts programmatically. Add Dropbox features to your web apps, like file storage, sharing, previews, and search, to name a few.

### *LinkedIn API*

Use LinkedIn APIs to search and share platform content programmatically.

### *MailChimp API*

Leverage the MailChimp API to synch email activity, organize campaign data, manage audiences, and automate workflows.

### *Constant Contact API*

Manage your Constant Contact accounts programmatically including getting, creating, and updating your contacts.

Just about any big-time web service has some kind of API available so that you can work with its data.

## **Migrations**

Migrations, migrations, migrations: where should we begin? Let's start at the beginning, and the first large-scale migration I ever worked on. Back when I was a lean, mean, green fighting machine and before the days of WordPress, I worked for a Marine Logistics command out of Camp Lejeune, North Carolina. While in the Marine Corps, and as part of my training, I learned how to build large-scale

relational databases from the ground up as well as update and maintain them. I learned the ins and outs of SQL (Structured Query Language) and the differences in syntax when working with various databases. We mostly used Microsoft SQL Server, but for some very large projects, we used Oracle. Most of the databases I built or worked on had web-based frontends built with either ColdFusion, Classic ASP, or ASP.NET, and over time I became very efficient at writing code in these programming languages and would use code to push and pull whatever data I wanted, to and from whatever databases I wanted. One of the largest web-based systems that I worked on while I was in the military, and still to this day, was a web-based system capable of tracking tons of all types of equipment, from tanks to toothbrushes. This logistics system was built in conjunction with civilian contractors to replace a mainframe system the Marines were using at that time to track inventory of deployable units. I was tasked with populating the Oracle database behind the new web application with massive amounts of data from its mainframe systems. I was ordered to eat, sleep, and breathe this massive data migration, so for several months that's just what I did, until all of the data was imported correctly. Even though WordPress wasn't even around yet, most of the lessons I learned while working on this migration project for the Marines more than 15 years ago still apply today.

Let's talk about WordPress migrations and how to move all the things to WordPress. When people say "WordPress migration" without much detail, we automatically think two things: a host migration or a platform migration. Depending on the task at hand, you may need to

know one or both of these. Whatever you need to do, there are tools and existing WordPress plugins we will talk about that you can use to make your task list less daunting. Let's go over the various types of migrations you may encounter.

## Host Migrations

A host migration is when you move your WordPress installation to another server or hosting environment. This happens a lot if you are a developer launching new websites for clients or if you are moving your website to another hosting company. The main objective here, though, is to move or “migrate” your existing website somewhere else. This process is mostly always the same.

1. Make a backup!
2. Move all of your source files to the new server.
3. Export and import your database.
4. Change DNS settings if need be.
5. Update any URLs in the database if need be.
6. Make sure nothing is broken.

## AVAILABLE MIGRATION PLUGINS

[Duplicator WordPress Migration Plugin](#) is a solid plugin for moving your WordPress installations.

[WP Migrate DB Pro](#) is one of our favorite migration plugins for moving data to and from WordPress for simple and complex migrations. With this plugin you can easily export and import any and

all of your WordPress database tables from one site to another. This is also a great plugin for working with production, staging, development, and localhost environments, as you can activate and configure it on all of your environments and easily move data around via the UX. This plugin offers several add-on plugins and you could certainly extend it with your own. Some of the notable add-ons include the CLI Add-on and the Multisite Tools Add-on.

## WP ENGINE SITE MIGRATION TOOL

Some hosting companies like WP Engine offer migration tools you can use to make moving to its hosting platform super-easy. The company tries to take the entire manual process of moving your website or web app out of the equation.

## Platform Migrations

Moving to WordPress from another platform is a popular thing to do but can require a lot of work depending on how much data there is and how well the data is structured. Are you moving from another open source platform like Drupal or Joomla to WordPress, or are you moving from a custom proprietary platform to WordPress? Whatever the scenario, you know that you want to automate importing the data to where it needs to go in WordPress. Following the steps listed here will ensure not only that you get your data moved over to WordPress, but that you don't pull all your hair out in the process.

## UNDERSTAND YOUR DATA DESTINATION

Getting the content into WordPress is half the battle! We've always been big fans of writing scripts to automate populating databases.

Whether by programming a simple web-based form for manually inputting data into a database or pulling out millions of records from one database and importing them into another, it's all about getting the data to where it's supposed to be in the format it's supposed to be in. To do this, it's very important to have a firm understanding of how all of your data is organized and stored in your destination database schema. Understanding the WordPress database schema and the helper functions within core WordPress that interact with the database is essential when it comes to migrating content into WordPress. If the WordPress database is somewhat of a mystery to you or you don't fully understand it, maybe you shouldn't have skipped Chapter 2!

## **UNDERSTAND YOUR DATA SOURCE**

So we know that our data destination is WordPress, but our data source can be anything. It can be another WordPress site; it can be another PHP and MySQL platform like Drupal or Joomla; it can be a proprietary database schema on MS SQL Server, Oracle, or any type of database; it can be data structured in XML or JSON being retrieved via an API; it could be a series of CSV files or even static HTML files. The list goes on and on, your data source can really be anything or go wherever the data you want to get into WordPress is stored. Whatever or wherever it is, you need to understand it, how it's structured, and how the data will map to WordPress. We've seen and worked on just about everything when it comes to moving data into WordPress, and next we provide a short list of some data sources with which we have extensive work experience.

## **DATA FROM DATABASES**

Most of our experience with platform migrations from an external data source to WordPress is from other SQL databases. Depending on the database engine, the database schema, and the database size, you might run into significant challenges when migrating your data:

#### *MySQL database schemas*

MySQL is free, very popular, and commonly used with most open source PHP-based content management systems or web publishing systems like WordPress, Drupal, Joomla!, Magento, ExpressionEngine, SilverStripe, and MediaWiki, to name a few.

#### *Microsoft SQL server database schemas*

SQL Server is not free, but is still very popular and used a lot to power anything from simple websites to enterprise-level web applications. Most of the time, SQL Server databases have a .NET web frontend.

#### *Oracle database schemas*

If you are running an Oracle database, you better be big time! If you have huge amounts of data (as in millions of records), it can be completely justified. However, we have worked with multiple clients that were using Oracle and it was just overkill. A fresh new user interface with WordPress and a MySQL database was the elegant, viable, and scalable solution they were really looking for.

#### *Other database engines*

There are other database engines out there and we've worked with all of them. Some other database engines include PostgreSQL, DB2, mongoDB, Microsoft Access, SQLite, and Sybase, to name a few.

## **DATA FROM FILES**

Sometimes you may not have direct access to a database. You might be running your entire web application from files on a server or maybe you are able to generate only exports of files from your application. Regardless of how you get your hands on the actual files that contain your data, you need to understand how the data is structured within those files. The following are some typical file formats that contain data:

#### *Comma-separated values (CSV)*

You could have any type of delimited files, but CSV is the most common.

#### *Extensible markup language (XML)*

Commonly used to interchange data across the web, XML has a few formats, including RSS, Atom, SOAP, and XHTML.

#### *JavaScript Object Notation (JSON)*

If contained in a file, it is usually in a `.txt` file. JSON is a very popular format for exchanging data that is largely replacing XML.

#### *Hypertext markup language (HTML)*

If you made it this far in the book, we assume we don't need to explain this one.

## **DATA FROM AN EXTERNAL API**

APIs are a very popular method of pushing and pulling data from one web-based system to or from another. We mentioned a few popular web services and APIs earlier in this chapter.

## **PAGE SCRAPING**

If all else fails and you can't access any of your data, you can always scrape your data or content from any live website. The complexity of scraping a website's content is dependent on how well structured the HTML of the website is. What you are looking for when page scraping is consistency.

Whatever your data source is, the idea is always going to be the same: parse the data, loop through it, and put it where it needs to go. Wow, we made that sound super-simple, but in most cases it really is if you have a firm understanding of WordPress and your data source(s).

## Create a Data Mapping Guide

This is one of the most important steps in any migration, no matter how small. Mapping the data from your data source to your data destination is going to save you so much time in the long run. We do this for every project no matter how big or how small, and you probably should too. Building a data mapping guide is actually pretty easy; we recommend using a Google spreadsheet so teammates can easily collaborate in real time. Begin by identifying content types in the data source, like posts, comments, users, and so on. List each content type along with any included metadata fields vertically on the lefthand side of the spreadsheet. Next to the name of each field include the data type (as in, if it's a string, an integer, or a Boolean). Next to the data type, you can include an example of real data. Next to the example data you could provide notes or further comments around that particular field. We recommend doing this for all content types for any data being brought over to WordPress.

Further to the right on the spreadsheet, start a new column called WordPress fields, and start mapping each of the data source fields to WordPress fields. Some fields are very obvious, like the post title. Some of the metadata from the data source will end up as meta in WordPress; if that's the case, add what the meta key will be in WordPress. If you are in the middle of building out a new theme or are using a WordPress plugin that has predefined meta keys baked in, make sure you reference all the correct meta keys. If you're working on a migration for a client, it's a good idea to go over your data mapping guide with them and get them to sign off on it. Make sure you, your client, and anyone else involved with your migration are all on the same page as far as what data is going where. If you skip making a data mapping guide, you may find yourself going back and forth with your client about where the data should go and then either writing a number of cleanup scripts to fix the data or fixing your main migration script and having to rerun it. Depending on how much data you are importing, rerunning your migration from scratch could take a bunch of time that you might not have. Our point is: make a data mapping guide; you will thank us in the long run.

# Chapter 18. The Future

---

The first edition of *Building Web Apps with WordPress* was published in 2014. A lot has changed in the five years between the first and second editions. A lot is going to change in the five years after this second edition is published.

In this final chapter, we will briefly point out some of the big changes in WordPress over the past few years. If you've read the book up to this point, you should have a good handle on these topics.

We will also make some predictions about the future of WordPress. Some of these predictions might miss the mark, but in general these are topics worth keeping tabs on for the next few years.

## Where We've Been

WordPress version 3.0, released in June 2010, was the first to provide support for CPTs (these are covered in detail in [Chapter 5](#)). The addition of CPTs was the final step in the transition of WordPress from blogging software to a content management system. The addition of CPTs empowered developers like us to start using WordPress as an application framework.

WordPress version 4.7, released in December 2016, included REST API endpoints for posts, comments, terms, users, meta, and settings

(the REST API is covered in detail in [Chapter 10](#)). While a feature plugin that enabled REST API support had been available for a couple years before this release, the existence of the REST API in the core WordPress plugin “blessed” the use of the REST API and encouraged developers to start building with it.

WordPress version 5.0, released in December 2018, included the new Gutenberg block editor (Gutenberg, blocks, and CPTs are covered in detail in [Chapter 11](#)). The block editor is new, and it will take time for sites, plugins, and themes to start taking full advantage of it. Still, the block editor enables a new paradigm for managing site content, layouts, and other settings. The Gutenberg project also showcased the use of a modern JavaScript development stack, embracing tools like Node.js, npm, React, webpack, and Git. In many ways, the Gutenberg project and feature plugins are setting the standards for WordPress development in general.

## The REST API

Even though the REST API was fully added to WordPress in version 4.7, it has taken a few years for developers and development stacks to start using and supporting API-enabled apps to their full potential. The REST API is now being used in various areas of WordPress core and many important plugins. Over time, we will see the REST API become an even more important feature of the WordPress platform.

## WordPress Plugins Will Focus More on APIs

Many projects now are starting out with an API-first mindset. Build the APIs your app will need separate from the frontend. Then decide what stack will work best for your frontend. This makes it easier to maintain the data layer of your app and also easier to add more view layers for your apps. The same WordPress content can power a blog website, a mobile app, and an email campaign, and any other app that needs that data.

Plugins like WooCommerce and BuddyPress have APIs with nearly 100% coverage for every plugin feature. This makes it easier to integrate data from those plugins with the apps plugged into your WordPress site.

## **Headless WordPress**

It is becoming more common to see WordPress used *solely* as a content data store for larger web apps that access that data through the REST API. Using WordPress in this way, where the administrator dashboard is used without using the built-in themes and frontend, is known as “headless WordPress.”

Static site generators like GatsbyJS offer a way to host static sites that are secure, fast, and relatively cheap to host. You can configure GatsbyJS to use WordPress as a data source. The WordPress installation managing the content doesn’t even have to be hosted online and could even be run from a local environment.

Even dynamic sites can make use of headless WordPress if the dynamic bits of the sites are coded in JavaScript running in the client

and using third-party services or APIs running on their own servers coded specific to what is required.

This kind of component-based development is popular now. There are pros and cons to having your app distributed across several different servers and services.

On the pro side, using several different services to run your app makes sure that each of those services is optimized for its purpose. If one component fails, the rest of your app may be able to run without it for a while. If one component has a security issue, you may avoid exposing your entire app. If one component needs to be replaced, it should be easier to replace just that one component versus the entire platform.

On the downside, using several different services to run your app means that you will spend more time learning new platforms and working out integration issues. Sometimes the various services won't work together so seamlessly, and you will need to get clever to make things work.

There seems to be a cycle in development best practices from self-contained apps to ones spread out across several services. Over time a collection of individual tools that work together well will get bundled up into hosting packages, integrated development environments, or managed containers. These bundles will mitigate some of those integration issues and make it easier to use a specific collection of tools. Then, over more time, these bundles will be viewed as a single platform. When you sign up for a WordPress-managed host now, you

generally don't have to think about the web server, database, PHP modules, or other tools that are running on the host to get WordPress working. In the future, there will be similar one-click installations for WordPress apps using a different collection of technologies, hosted across different servers.

## GraphQL

GraphQL is a query language for APIs, and is likely to replace REST APIs everywhere, including in WordPress. We spent a lot of time, all of [Chapter 10](#), describing and praising the REST API. So what makes GraphQL so much better?

With a traditional REST API, the API has to define the endpoints, including what data must be passed into and out of requests. With GraphQL, the service making the requests defines the query and the format in which it wants the data returned. This means there is less work required to publish and maintain the API, while giving services using that API more flexibility in how the API is used.

Using a REST API is like using a function in PHP. There are predefined parameters (although some are optional) and data is generally returned in the same format. GraphQL is like making a SQL query. You can query any available data, join data how you want, and define the values you would like returned.

To learn more about GraphQL, visit <https://graphql.org/> or read *Learning GraphQL* by Alex Banks and Eve Porcello (O'Reilly). WP GraphQL is a plugin for WordPress that enables GraphQL queries

against your WordPress site. Learn more at  
<https://www.wpgraphql.com>.

## Gutenberg

Since the introduction of the new block editor in WordPress 5.0, we've seen rapid development of new blocks and use cases for the block editor, both in WordPress core and in other projects. In this section we cover a collection of predictions around Gutenberg and the block editor.

### The Administrator Interface Will Move to React/Gutenberg

Any setting related to how content is displayed visually on the frontend of your website is a perfect candidate to be ported to blocks. And so as part of Phase 2 of the Gutenberg project, widgets and navigation menus will be ported to blocks.

Moving other admin screens to use React and API calls can greatly increase the performance and usability of the WordPress dashboard. Some prototyping around how the WordPress List Tables view can be ported to React can be found on the [New List Tables GitHub repository](#).

### Gutenberg Will Power a Frontend Editing Experience for WordPress

Already, the WordPress core developers and block developers try their best to have the block editor resemble as much as possible how

content is displayed on the frontend. As things like menus, sidebars, and other parts of the layout of a web page move into the block editor, you will basically be able to change any aspect of your site's design by arranging and editing blocks.

There may be disagreement over what it actually means to have a frontend editing experience for WordPress. At the same time, swapping to the dashboard may become so quick and seamless, that the current editor experience may become "good enough" for many who desire frontend editing. While an official "frontend editing experience" may get delayed, there will be something close to it or plugins to fill the gap.

## **Block Templates Will Replace Themes**

WordPress themes define the look and feel of your website through choices of color, font, and layout. Most themes allow for flexibility of color and font through customizer options, and now layouts can be controlled through block templates.

This is a gross simplification of website design, and a good design will include a lot of subtle art outside of colors, fonts, and layouts. At the same time, this is the level of control that users have come to expect when selecting a theme for WordPress.

The job of a good theme will be to make sure that popular blocks and block templates look good with the theme's design.

For a while, themes that added complex features would have those features labeled as "plugin territory." The core principal behind this

label is that a user should be able to change their theme without losing any content or functionality in their website. Anything that breaks this principal should be coded into a plugin instead. Themes in the WordPress.org repository had to follow this rule completely, and many themes sold outside the repository tried their best to honor the rule while still having their themes stand out and support specific use cases.

Many theme shops have basically become plugin shops, selling premium plugins that allow their themes to reach their full potential. The “plugin territory” of many of these themes is now “block territory.”

Themes will not go away. But the theme you select will set the tone for how your website looks, but over time the handling of headers, footers, sidebars, and other layout components will be controlled in the block editor.

## **Blocks Will Replace Plugins**

A common refrain in the WordPress community has been “there’s a plugin for that.” With more than 50,000 plugins in the WordPress.org repository alone, there is a good chance that there is a plugin for nearly any feature or integration you are looking for. You just have to find it.

As the block editor controls more of the attention and focus of the users building sites with WordPress, those users will be trained to look first for blocks to meet their needs. Many of those blocks will be

delivered by plugins, but blocks can also be added by loading a JavaScript file from a shared server.

Plugins will not go away, but users will more and more think about them as they relate to blocks. Instead of a membership plugin, users will search for a membership block. While existing plugins can add blocks,<sup>1</sup> plugins that are built from the ground up as blocks will be more approachable for users and better able to iterate on the new tech stack and development paradigms standardized with Gutenberg.

## **WordPress Market Share Will Increase and Decrease**

WordPress powers about one-third of all websites on the internet. Between 2011 and 2019 WordPress market share has increased from 13.1% of all sites to 34%. Statistically, this rate of increase doesn't seem to be slowing down much.

From an end user standpoint, WordPress competes with self-hosted services like Squarespace and Shopify. WordPress-based hosts have streamlined the process for setting up a new website, and the new block-based editor has streamlined the process for laying out content. These improvements mean that WordPress will continue to compete favorably with the self-hosted solutions.

Vertical consolidation of WordPress companies, with hosting companies buying plugin and theme companies, will further streamline the process of creating WordPress-based websites.

While the number of websites fully developed with WordPress might flatline or decrease a bit, the increase in decoupled websites using WordPress as part of a broader application stack will increase.

## **WordPress Will Become a More Popular Platform for Mobile Development**

The REST API, standards for Progressive Web Apps, and tools like AppPresser make it easier than ever to create mobile apps with WordPress. If you’re building a web app using React and a headless instance of WordPress, it makes sense to use that same WordPress instance to power a mobile frontend developed with React Native.

## **WordPress Will Continue to Be Useful for Developing Apps of All Kinds**

Five years from now, WordPress will still be around and powering even more of the internet. The platform will be better. The APIs will be better. The tools will be better. Hosting and other services will be better. This book will have published another edition or two.

We’re excited by all of the great work you and other WordPress developers will do. If you build something awesome with WordPress, share it with us on Twitter at [@BWAwWP](https://twitter.com/BWAwWP) or contact us at <https://bwawwp.com>.

---

<sup>1</sup> Paid Memberships Pro definitely did soon after the release of the block editor.

# Index

---

## SYMBOLS

\$authordata global variable, [WordPress Global Variables](#)

\$blog\_id global variable, [\\$blog\\_id](#)

\$current\_user object, [Insert, replace, and update](#)

\$post global variable, [WordPress Global Variables](#), [Enqueuing Other JavaScript Libraries](#)

\$shortcode\_tags global variable, [Removing Shortcodes](#)

\$template variable, [Using locate\\_template in Your Plugins](#)

\$temp\_content function, [/pages/](#)

\$text variable, [Other Useful Shortcode-Related Functions](#)

\$wpdb class, [\\$wpdb](#), [Escaping in database queries](#)

SELECT queries with, [SELECT queries with \\$wpdb](#)

\$wpdb object, [Custom SQL Statements](#)

\$wpdb→get\_col() method, [Escaping in database queries](#), [SELECT queries with \\$wpdb](#)

\$wpdb→get\_results() method, [SELECT queries with \\$wpdb](#)

\$wpdb→get\_row() method, [SELECT queries with \\$wpdb](#)

\$wpdb→insert() method, [Insert, replace, and update](#)

\$wpdb→prefix property, [Using custom database tables](#)

\$wpdb→prepare() function, Custom SQL Statements

\$wpdb→prepare() method, Escaping in database queries

\$wpdb→query() method, Using custom database tables, Running queries

\$wpdb→replace() method, Insert, replace, and update

\$wpdb→update() method, Insert, replace, and update

\$wp\_query object, Insert, replace, and update

\$wp\_query→query\_vars[subject] global variable, Adding Rewrite Rules

\$\_REQUEST values, checking, Enqueuing Other JavaScript Libraries

\$\_SERVER[HTTP\_USER\_AGENT] global variable, Device Detection in PHP

% (percent sign), escaping in SQL queries, Escaping in database queries

@import\_url, Creating a Child Theme for Memberlite  
\_ (underscore)

metadata keys starting with, wp\_postmeta

\_s theme framework, \_s (underscores)

\_\_ (double underscore), class methods starting with, Getting User Data

\_\_() function, \_\_( \$text, \$domain = “default” )

\_e() function, \_e( \$text, \$domain = “default” )

\_ex() function, \_x( \$text, \$context, \$domain = “default” )

\_x() function, \_x( \$text, \$context, \$domain = “default” )

→ (arrow) operator, Getting User Data

## A

action hooks, Hooks: Actions and Filters

actions available to users, controlling, User Management Is Easy and Secure with WordPress

active\_sidebar() function, Defining a Widget Area

add-ons to existing plugins, Add-Ons to Existing Plugins

add-term\_meta() function, add\_term\_meta( \$term\_id, \$meta\_key, \$meta\_value, \$unique = false )

add-user\_meta() function, add\_user\_meta( \$user\_id, \$meta\_key, \$meta\_value, \$unique = false )

add\_action() function, Actions, Adding Widgets

add\_cap() method, Creating Custom Roles and Capabilities

add\_comment\_meta() function, get\_comment\_meta( \$comment\_id, \$key = ”, \$single = false )

add\_feed() function, Other Rewrite Functions

add\_filter() function, Filters, Could You Use a Hook or Filter Instead?

add\_meta\_box() function, Metadata with CPTs

add\_option() function, add\_option( string \$option, mixed \$value = ”, string \$deprecated = ”, string|bool \$autoload = ‘yes’ )

add\_post\_meta() function, add\_post\_meta( \$post\_id, \$meta\_key, \$meta\_value, \$unique = false )

`add_rewrite_endpoint()` function, [Other Rewrite Functions](#)

`add_rewrite_rule()` function, [Adding Rewrite Rules](#), [Other Rewrite Functions](#)

`add_role()` function, [Creating Custom Roles and Capabilities](#)

`add_user_to_blog()` function, `add_user_to_blog( $blog_id, $user_id, $role )`

admin bar, hiding from non-admins, [Hide the Admin Bar from Nonadministrators](#)

Admin Columns plugin, [Admin Columns](#)

admin dashboard

settings for plugins, [Settings API](#)

SSL login, [WordPress Login](#) and [WordPress Administrator over SSL](#)

Admin role, [What Are Roles and Capabilities?](#)

admin username, importance of changing, [Don't Use the Username "admin"](#)

admin-ajax.php file, [/wp-admin](#)

admin.css files, [/css/](#)

admin.js files, [/js/](#)

[/adminpages/](#) directory, [/adminpages/](#)

`admin_enqueue_scripts` hook, [Enqueuing Other JavaScript Libraries](#)

Advanced Custom Fields plugin, [Advanced Custom Fields](#)

`advanced-cache.php`, [advanced-cache.php](#) and [object-cache.php](#)

AJAX

admin-ajax.php file, /wp-admin

calls triggered through Heartbeat API, Heartbeat API-Server-side PHP

calls with WordPress and jQuery, Ajax Calls with WordPress and jQuery-Ajax Calls with WordPress and jQuery

check\_ajax\_referer() function, check\_ajax\_referer( \$action = -1, \$query\_arg = false, \$die = true )

defined, What Is Ajax?

managing multiple AJAX requests, Managing Multiple Ajax Requests

PHP code for calls in /services/ directory, /services/

Akismet plugin, Using WordPress Plugins, Akismet

All In One WP Security and Firewall plugin, All In One WP Security & Firewall

ALTER TABLE statement, Using custom database tables

Alternative PHP Cache (APC), Alternative PHP Cache (APC)

anonymous functions, Could You Use a Hook or Filter Instead?

Apache Bench, Apache Bench

graphing results with gnuplot, Graphing Apache Bench results with gnuplot

installing and running, Installing Apache Bench

testing with, Testing with Apache Bench

Apache server

Nginx server in front of, Nginx in front of Apache

setup, [Apache server setup](#)

API, [API](#)

App Builder, [The App Builder](#)

app wrapper, [App Wrapper](#)

AppCamera plugin, [AppCamera plugin](#)

hooks for app customizations for, [AppCamera plugin](#)

AppCommerce, [WooCommerce plugins](#)

apply\_filters() function, [Filters](#), [Nested Shortcodes](#)

AppPresser, [AppPresser](#)

installing and configuring, [Installing and configuring on WordPress](#)

list of plugins for, [Installing and configuring on WordPress](#)

using App Builder for, [The App Builder](#)

AppPush, [AppPush](#)

apps

admin, using global of settings, [Do You Really Need a Settings Page?](#)

developing, themes versus plugins, [Where to Place Code When Developing Apps](#)

AppWoo plugin, [WooCommerce plugins](#)

hooks for, [WooCommerce plugins](#)

archive

compressing files into an, [File Compression and Archiving](#)

compressing individual files into an, [Compressing individual files into an archive](#)

compressing multiple files into an, [Compressing multiple files into an archive](#)

extracting files from an, [Extracting files from an archive](#)

archives

for registered CPTs, [The Theme Archive and Single Template Files](#)

specifying if post type has archive page, `register_post_type( $post_type, $args );`

arrays, storing in user meta, [Add, Update, and Delete Users](#)

arrow operator ( $\rightarrow$ ), [Getting User Data](#)

Ask Apache Password Protect plugin, [AskApache Password Protect](#)  
Asynchronous JavaScript and XML (see AJAX)

asynchronous processing, WordPress limitations with, [WordPress Limitations with Asynchronous Processing](#)

attachments, [Attachment](#)

attributes

HTML, escaping, `esc_attr( $text )`

shortcode, [Shortcode Attributes](#)

authenticating users, [Add, Update, and Delete Users](#)

Author role, [What Are Roles and Capabilities?](#)

upgrading Subscriber to, [Checking a User's Role and Capabilities](#)

[Authorize.net](#), [Payment Gateways](#)

## B

[Backbone.js](#), [Backbone.js](#)

[BackupBuddy plugin](#), [BackupBuddy](#), [BackupBuddy](#)

[backups](#), [Back Up Everything!](#)

[plugins for](#), [Backup Plugins](#)

[Bad Behavior plugin](#), [Bad Behavior](#)

[BadgeOS Community Add-on plugin](#), [BuddyPress plugins](#)

[BadgeOS plugin](#), [BadgeOS](#)

[basic authentication](#), [Basic Authentication](#)

[building](#), [Bundling Basic Authentication with Our Plugin](#)

[Batcache](#), [Batcache](#)

[bbPress plugin](#), [Classes Are BuddyPress Groups](#)

[block editor](#), [JavaScript Frameworks and Workflow](#), [Project](#)

[Gutenberg](#), [Blocks](#), and [Custom Block Types](#), [The WordPress Editor](#)

[CPTs and](#), [Enabling the Block Editor in Your CPTs](#)

[creating your own blocks with](#), [Creating Your Own Blocks](#)

[custom blocks and](#), [Using Custom Blocks to Build App](#)

[Experiences](#)

[filemtime\(\)](#) and, [Use filemtime\(\) for the Script Version](#)

[future of](#), [Gutenberg](#)

[tips for using](#), [Tips](#)

using for content and design, Using Blocks for Content and Design

using for functionality, Using Blocks for Functionality

using meta boxes with, Using Meta Boxes with the Block Editor

WP\_SCRIPT\_DEBUG and, Enable WP\_SCRIPT\_DEBUG

Block Editor Handbook, Project Gutenberg, Blocks, and Custom Block Types, Block Templates

block templates

future of, Block Templates Will Replace Themes

blocks, Project Gutenberg, Blocks, and Custom Block Types-Learn JavaScript, Node.js, and React More Deeply

categories of, Block Categories

future of, Gutenberg

limiting to specific CPTs, Limiting Blocks to Specific CPTs

React and, Learn JavaScript, Node.js, and React More Deeply

reusable, Reusable Blocks

saving data to post meta, Saving Block Data to Post Meta

templates for, Block Templates

using with Node.js, Learn JavaScript, Node.js, and React More Deeply

vanilla JavaScript and, Learn JavaScript, Node.js, and React More Deeply

blocks versus plugins, Blocks Will Replace Plugins

blogs, WordPress and, WordPress is just for blogs

Bootstrap framework, Non-WordPress Theme Frameworks

including in Memberlite theme, Including Bootstrap in Your App's Theme

responsive stylesheet adjusting CSS rules for screen width, Device and Display Detection in CSS

Braintree Payments, Payment Gateways

browscap.ini file, Browser detection with PHP's get\_browser()

Browser Capabilities Project website, Browser detection with PHP's get\_browser()

browser detection

in PHP, Device Detection in PHP

in WordPress core, Browser detection in WordPress core

reasons for sparing use of, Final Note on Browser Detection

with PHP get\_browser(), Browser detection with PHP's get\_browser()

browsers

CSS stylesheet caching, Versioning Your Theme's CSS Files

padlocks, Installing an SSL Certificate on Your Server

brute-force attacks, Why It's Important

plugin protecting against, Limit Login Attempts

BuddyPress Media plugin, BuddyPress plugins

BuddyPress plugin, BuddyPress-BuddyPress plugins

components, setting up, Components

configuring additional settings, Settings

groups, SchoolPress classes as, [Classes Are BuddyPress Groups](#)  
pages, mapping to components used, [Pages](#)  
plugins to extend BuddyPress, [BuddyPress plugins](#)  
profile fields, [Profile fields](#)  
tables created in WordPress database, [Database tables](#)

BuddyPress Registration Options plugin, [BuddyPress plugins](#)  
BuddyPress REST API, [BuddyPress](#)  
buffering output, [/pages/](#)  
[\\_builtin](#), [register\\_post\\_type\( \\$post\\_type, \\$args \);](#)

## C

CA certificates, [Installing an SSL Certificate on Your Server](#)  
caching, [W3 Total Cache](#)  
(see also [W3 Total Cache](#))

[advanced-cache.php](#) and [object-cache.php](#), [advanced-cache.php](#)  
[and object-cache.php](#)

Alternative PHP Cache (APC), [Alternative PHP Cache \(APC\)](#)

Batcache, [Batcache](#)

Memcached, [Memcached](#)

Redis, [Redis](#)

selective, [Selective Caching](#)

multisite transients, [Multisite Transients](#)

Transient API, [The Transient API-The Transient API](#)

Varnish, Varnish

W3 Total Cache plugin, W3 Total Cache

can\_export, register\_post\_type( \$post\_type, \$args );

capabilities

custom post types (CPTs), register\_post\_type( \$post\_type, \$args );

taxonomy, register\_taxonomy( \$taxonomy, \$object\_type, \$args )

user, What Are Roles and Capabilities?

adding new, Creating Custom Roles and Capabilities

checking, Checking a User's Role and Capabilities, Check User Capabilities

removing, Creating Custom Roles and Capabilities

capability\_type, register\_post\_type( \$post\_type, \$args );

CDNs (content delivery networks), CDNs

Certbot, Installing an SSL Certificate on Your Server

changesets, Changesets

checkUsername() function, Ajax Calls with WordPress and jQuery

check\_admin\_referer() function, check\_admin\_referer( \$action = -1, \$query\_arg = '\_wpnonce' )

check\_ajax\_referer() function, check\_ajax\_referer( \$action = -1, \$query\_arg = false, \$die = true )

Chrome Debug Bar, Chrome Debug Bar

Chrome Developer Tools Console, SSL error in, Avoiding SSL Errors with the “Nuclear Option”

/classes/\_ directory, /classes/

Classic Editor, Using Meta Boxes with the Block Editor

Classic Editor plugin, The Classic Editor Plugin

closures, Could You Use a Hook or Filter Instead?

CMS (content management system), WordPress as, Content Management Is Easy with WordPress, WordPress Basics

Code

menu position values, register\_post\_type( \$post\_type, \$args );

Rewrite API and WP\_Rewrite class pages, Other Rewrite Functions

comment , in wp\_commentmeta table, wp\_commentsmeta

comments on posts, in wp\_comments table, wp\_comments

community plugins, Community Plugins-BuddyPress plugins

compression, GZIP Compression

content delivery networks (CDNs), CDNs

content sites, WordPress and, WordPress is just for content sites

content-focused web apps, Features of a Web App

Contributor role, What Are Roles and Capabilities?

Cordova, Cordova

installing, Installing Cordova

list of core plugins for, Cordova plugins

Cordova and Android, Cordova and Android

Cordova and iOS, Cordova plugins

CPTs (see custom post types)

limiting to specific blocks, [Limiting CPTs to Specific Blocks](#)

CREATE TABLE statement, [Using custom database tables](#)

`create_empty_blog()` function, [create\\_empty\\_blog\( \\$domain, \\$path, \\$weblog\\_title, \\$site\\_id = 1 \)](#)

`create_function()` function, [Adding Widgets](#)

cron jobs, [WP-Cron-Using Server Cron Only](#)

adding to an app, [WP-Cron](#)

PHP code for, [/scheduled/](#)

scheduling, [WP-Cron](#)

using server crons only, [Using Server Cron Only](#)

`crontab -e` command, [Kicking Off Cron Jobs from the Server](#)

`cron_schedules` hook, [Adding Custom Intervals](#)

cross-site scripting attacks, [Sample Page Template](#)

CRUD actions, [HTTP](#)

CSRF, [Cookie authentication](#)

CSRF (cross-site request forgery) attacks, [Nonces](#)

CSS

device and display detection in, [Device and Display Detection in CSS](#)

files for an app plugin, [/css/](#)

style.css file for themes, [Style.css](#)

using to show/hide menu items, [Dynamic Menus](#)

current\_user\_can() function, Checking a User's Role and Capabilities, current\_user\_can( \$capability )

custom block type, Does this metadata need to be placed within the post content?

custom block types, Project Gutenberg, Blocks, and Custom Block Types-Learn JavaScript, Node.js, and React More Deeply

custom blocks

building apps with, Using Custom Blocks to Build App Experiences

SchoolPress and, Using Custom Blocks to Build App Experiences

Custom Post Type UI plugin, register\_post\_type( \$post\_type, \$args );

custom post types (CPTs), Content Management Is Easy with WordPress, Custom Post Types, Post Metadata, and Taxonomies

custom wrapper classes for, Custom Wrapper Classes for CPTs-Wrapper Classes Read Better

extending WP\_Post vs. wrapping it, Extending WP\_Post Versus Wrapping It

keeping CPT functionality in wrapper class, Keep It in the Wrapper Class

keeping CPTs and taxonomies together, Keep Your CPTs and Taxonomies Together

making code easier to read, Wrapper Classes Read Better

reasons for using wrapper classes, Why Use Wrapper Classes?

defining and registering, [Defining and Registering CPTs](#)-  
`register_post_type( $post_type, $args );`

in SchoolPress sample app, [Assignments Are a CPT](#)

metadata with, [Metadata with CPTs](#)-`add_meta_box( $id, $title, $callback, $screen, $context, $priority, $callback_args )`

themes and, [Themes and CPTs](#)

using in themes and plugins, [Using CPTs and Taxonomies in Your Themes and Plugins](#)-[Metadata with CPTs](#)

looping through CPTs, [Good Old WP\\_Query](#) and  
`get_posts()`

theme archive and single template files, [The Theme Archive and Single Template Files](#)

`customize_changeset`, [Changesets](#)

`custom_css`, [Custom CSS](#)

## D

dashboard

customizing users table in, [Customizing the Users Table in the Dashboard](#)

WordPress Multisite network, [Dashboard](#)

dashboard widgets, [Dashboard Widgets API](#)

adding your own, [Adding Your Own Dashboard Widget](#)

removing, [Removing Dashboard Widgets](#)

data migrations, [Data from databases](#)

database caching, [Database Caching](#)

database, WordPress

\$wpdb class, \$wpdb

changing default tables prefix, [Change Default Database Tables Prefix](#)

custom tables for performance optimization, [Custom Tables](#)

escaping in values passed to query() method, [Escaping in database queries](#)

Multisite network database, [Multisite Database Structure-Shared Site Tables](#)

individual site tables, [Individual Site Tables](#)

network-wide tables, [Multisite Database Structure](#)

shared site tables, [Shared Site Tables](#)

structure of, [WordPress Database Structure](#)

`wp_set_object_terms( $object_id, $terms, $taxonomy, $append = false )`

functions in /wp-includes/option.php, [Functions Found in /wp-includes/option.php](#)

functions in /wp-includes/pluggable.php file, [Functions Found in /wp-includes/...](#)

wp\_comments table, [wp\\_comments](#)

wp\_commentsmeta table, [wp\\_commentsmeta](#)

wp\_options table, [wp\\_options](#)

wp\_postmeta table, [wp\\_postmeta](#)

wp\_posts table, [wp\\_posts](#)

wp\_terms table, [wp\\_terms](#)

wp\_term\_relationships table, wp\_term\_relationships

wp\_term\_taxonomy table, wp\_term\_taxonomy

wp\_usermeta table, wp\_usermeta

wp\_users table, wp\_users

tables created by BuddyPress, Database tables

using custom tables, Using custom database tables

wp\_p2p and wp\_p2pmeta tables, Posts 2 Posts

datatypes, jQuery.ajax() output, Ajax Calls with WordPress and jQuery

dbDelta() function, Using custom database tables

db\_version, Using custom database tables

delete\_blog\_option() function, delete\_blog\_option( \$id, \$option )

delete\_comment\_meta() function, delete\_comment\_meta( \$comment\_id, \$meta\_key, \$meta\_value = "" )

delete\_option() function, delete\_option( \$option )

delete\_post\_meta() function, delete\_post\_meta( \$post\_id, \$meta\_key, \$meta\_value = "" )

delete\_term\_meta() function, delete\_term\_meta( \$term\_id, \$meta\_key, \$meta\_value = "" )

delete\_user and deleted\_User hooks, Hooks and Filters

delete\_user\_meta() function, delete\_user\_meta( \$user\_id, \$meta\_key, \$meta\_value = "" )

delete\_with\_user, register\_post\_type( \$post\_type, \$args );

deleting users, Add, Update, and Delete Users

denial of service (DoS) attacks, [Why It's Important](#)  
description (CPTs), [register\\_post\\_type\( \\$post\\_type, \\$args \);](#)  
developers, PHP libraries for, [Developer Tools](#)  
device capabilities, web apps, [Features of a Web App](#)  
device detection  
    in CSS, [Device and Display Detection in CSS](#)  
    in JavaScript, [Device and Feature Detection in JavaScript](#)  
    in PHP, [Device Detection in PHP](#)  
directory structure, WordPress, [WordPress Directory Structure](#)  
    /wp-admin directory, [/wp-admin](#)  
    /wp-content directory, [/wp-content](#)  
    /wp-content/plugins directory, [/wp-content/plugins](#)  
    /wp-includes directory, [/wp-includes](#)  
    /wp/content/mu/plugins directory, [/wp-content/mu-plugins](#)  
    /wp/content/themes directory, [/wp-content/themes](#)  
    /wp/content/uploads directory, [/wp-content/uploads](#)  
    root directory, [Root Directory](#)  
DISABLE\_WP\_CRON, [Kicking Off Cron Jobs from the Server](#)  
DISALLOW\_FILE\_EDIT, [Don't Allow Admins to Edit Plugins or Themes](#)  
displays, detection using CSS media queries, [Device and Display Detection in CSS](#)

distributed (source code), [The General Public License, Version 2, License](#)

[do\\_action\(\) function, Actions](#)

[do\\_shortcode\(\) function, Nested Shortcodes](#)

[Dynamic Dummy Image Generator, Dynamic Dummy Image Generator](#)

[dynamic\\_sidebar\(\) function, Defining a Widget Area](#)

## E

[Easy Digital Downloads plugin, Easy Digital Downloads, Easy Digital Downloads code examples](#)

[ECMAScript, JavaScript Frameworks and Workflow, What Is ECMAScript?](#)

[ecommerce, Ecommerce-International and long-form addresses](#)

[choosing a plugin](#)

[digital downloads, Easy Digital Downloads](#)

[membership plugins, Paid Memberships Pro](#)

[merchant accounts, Merchant Accounts](#)

[payment gateways, Payment Gateways](#)

[setting up SaaS with Paid Memberships Pro, Setting Up SaaS with Paid Memberships Pro-International and long-form addresses](#)

[SSL certificates and HTTPS, SSL Certificates and HTTPS](#)

[edge, Terms](#)

[origin versus, Origin Versus Edge](#)

Editor role, [What Are Roles and Capabilities?](#)

removing `edit_pages` capabilities, [Creating Custom Roles and Capabilities](#)

`_edit_link`, `register_post_type( $post_type, $args );`

`edit_user_profile`, hooking into, [Adding Registration and Profile Fields](#)

ElasticPress, [ElasticPress by 10up](#)

Elasticsearch, [Elasticsearch](#)

email addresses

sanitizing, `sanitize_email( $email )`

validating and sanitizing, `wp_kses_post( $data )`

endpoint mask constants, [Other Rewrite Functions](#)

ES6, [What Is ES6?](#)

ES9, [What Is ES9?](#)

escaping data, [Data Validation, Sanitization, and Escaping](#)

while translating strings, [Escaping and Translating at the Same Time](#)

`esc_attr()` function, [Sample Page Template](#), `esc_attr( $text )`

`esc_html()` function, `esc_html( $text )`

`esc_js()` function, `esc_js( $text )`

`esc_sql()` function, [Escaping in database queries](#), [Custom SQL Statements](#)

`esc_textarea()` function, [Sample Page Template](#), `esc_textarea( $text )`

esc\_url() function, esc\_url( \$url, \$protocols = null, \$context = 'display' )

esc\_url\_raw() function, esc\_url\_raw( \$url, \$protocols = null )

ESNext, What Is ESNext?

exclude\_from\_search (CPTs), register\_post\_type( \$post\_type, \$args );

Exploit Scanner plugin, Exploit Scanner

external IP address, Add Custom .htaccess Rules for Locking Down wp-admin

extract() function, Shortcode Attributes

extra\_{context}\_headers, Adding New Headers to Plugins and Themes

## F

Faker, Faker

feature detection in JavaScript, Feature detection in JavaScript

File Header API, File Header API

adding file headers to your files, Adding File Headers to Your Own Files

adding new headers to plugins and themes, Adding New Headers to Plugins and Themes

file structure for an app plugin, File Structure for an App Plugin-/schoolpress.php

/adminpages/ directory, /adminpages/

/classes/ directory, /classes/

/css/ directory, /css/  
/images/ directory, /images/  
/js/ directory, /js/  
/scheduled/ directory, /scheduled/  
/services/ directory, /services/  
main plugin file, /schoolpress.php

## \_\_FILE\_\_, Enqueuing Other JavaScript Libraries

filters

in plugins, Add-Ons to Existing Plugins

using in WordPress core, plugins, or themes, Filters

using instead of settings page, Could You Use a Hook or Filter Instead?

wp\_default\_styles, Versioning Your Theme's CSS Files

flexibility of WordPress, Flexibility Is Important

flexibility, importance of, Flexibility Is Not Important to You

flush\_rewrite\_rules() function, Flushing Rewrite Rules

FORCE\_SSL\_ADMIN constant, WordPress Login and WordPress Administrator over SSL

FORCE\_SSL\_LOGIN constant, WordPress Login and WordPress Administrator over SSL

forms

Gravity Forms plugin, Gravity Forms

page template features for, Sample Page Template

Foundation framework, Non-WordPress Theme Frameworks

FPDF, FPDF

frameworks

importing into themes, Including Bootstrap in Your App's Theme

popular theme frameworks, Popular Theme Frameworks

non-WP frameworks, Non-WordPress Theme Frameworks

frontend.css files, /css/

frontend.js files, /js/

functions to register custom post types, register\_post\_type( \$post\_type, \$args );

functions.php file

for themes, Creating a Child Theme for Memberlite

of the active theme, functions.php

## G

gamification plugin, BadgeOS

GamiPress plugin, BadgeOS

GatsbyJS, Headless WordPress

GD, GD

Genesis theme framework, Genesis

\_\_get() method, WP\_User class, Getting User Data, Getting User Data, Extending the WP\_User Class

gettext translation system, How Localization Is Done in WordPress

get\_blog\_details() function, get\_blog\_details( \$fields = null, \$get\_all = true )

get\_blog\_option() function, get\_blog\_option( \$id, \$option, \$default = false )

get\_blog\_post() function, get\_blog\_post( \$blog\_id, \$post\_id )

get\_blog\_status() function, get\_blog\_status( \$id, \$pref )

get\_browser() function, Browser detection with PHP's get\_browser()

get\_comment() function, get\_comment( \$comment, \$output = OBJECT )

get\_comments() function, get\_comments( \$args = " )

get\_comment\_meta() function, get\_comment\_meta( \$comment\_id, \$key = ", \$single = false )

get\_current\_blog\_id() function, get\_current\_blog\_id()

get\_file\_data() function, Adding File Headers to Your Own Files

get\_locale() function, Setting the Text Domain

get\_object\_taxonomies() function, get\_object\_taxonomies( \$object, \$output = 'names' )

get\_option() function, get\_option( \$option, \$default = false )

get\_plugin\_data() function, File Header API

get\_post() function, get\_post( \$post = null, \$output = OBJECT, \$filter = 'raw' )

get\_posts() function, get\_posts( \$args = null ), Good Old WP\_Query and get\_posts()

get\_post\_meta() function, get\_post\_meta( \$post\_id, \$key = "", \$single = false )

get\_taxonomies() function, get\_taxonomies( \$args = array(), \$output = 'names', \$operator = 'and' )

get\_taxonomy() function, get\_taxonomy( \$taxonomy )

get\_template\_part() function, Theme-Related WordPress Functions

get\_term() function, get\_term( \$term, \$taxonomy, \$output = OBJECT, \$filter = 'raw' )

get\_terms() function, Functions Found in /wp-includes/taxonomy.php, get\_terms( \$taxonomies, \$args = '' )

get\_term\_meta() function, get\_term\_meta( \$term\_id, \$key = '', \$single = false )

get\_userdata() function, get\_userdata( \$userid )

get\_user\_by() function, get\_user\_by( \$field, \$value )

get\_user\_meta() function, get\_user\_meta( \$user\_id, \$key = '', \$single = false ), Getting User Data

looping through all meta data for a user, Getting User Data

GitHub, PHP Libraries

SchoolPress source code, What Is SchoolPress?

global variables, WordPress Global Variables

in wp-includes/vars.php, Browser detection in WordPress core

using to store array of options for plugin or app, Do You Really Need a Settings Page?

GlotPress, GlotPress

translations and, GlotPress

using, [Using GlotPress for Your WordPress.org Plugins and Themes](#)

GlotPress server

creating your own, [Creating Your Own GlotPress Server](#)

GNU General Public License, version 2 (GPLv2), [You Plan to License or Sell Your Site's Technology, The General Public License, Version 2, License](#)

gnuplot, [Graphing Apache Bench results with gnuplot](#)

Google Directions, [Directions](#)

Google Distance Matrix, [Distance Matrix](#)

Google Elevation, [Elevation](#)

Google Geocoding, [Geocoding](#)

Google Maps, [Google Maps](#)

WordPress plugins for, [Google Maps WordPress plugins](#)

Google Street View Service, [Street View service](#)

Google Translate, [Google Translate](#)

Google Vision, [Google Vision](#)

Gravity Forms plugin, [Gravity Forms](#)

Gutenberg editor, [Your App Needs to Be Highly Real Time \(see block editor\)](#)

Gutenberg plugin, [JavaScript Frameworks and Workflow](#)

GZIP compression, [GZIP Compression](#)

H

has\_archive, register\_post\_type( \$post\_type, \$args );

has\_shortcode() function, Other Useful Shortcode-Related Functions

have\_posts() function, The WordPress Loop

header.php file, jQuery and WordPress

headers, Headers

headless WordPress, Headless WordPress

Heartbeat API, Heartbeat API-Server-side PHP

JavaScript events triggered by, Client-side JavaScript

speeding up or slowing down heartbeat, Server-side PHP

heartbeat\_received hook, Server-side PHP

Hello Dolly plugin, Using WordPress Plugins

hidden fields in forms, Sample Page Template

Hide Admin Bar from Non-Admins plugin, Hide the Admin Bar from Nonadministrators

hierarchical option

posts, register\_post\_type( \$post\_type, \$args );

taxonomies, register\_taxonomy( \$taxonomy, \$object\_type, \$args )

hooks

action hooks, Hooks: Actions and Filters

admin\_enqueue\_scripts, Enqueuing Other JavaScript Libraries

apply\_filters(), Filters

cron\_schedules, Adding Custom Intervals

defining AJAX functions in WordPress, Ajax Calls with WordPress and jQuery

delete\_user and deleted\_User, Hooks and Filters

in custom profile field, Adding Registration and Profile Fields

in plugins, Add-Ons to Existing Plugins

user\_register, Hooks and Filters

using for settings, Could You Use a Hook or Filter Instead?

using to copy page templates, Using Hooks to Copy Templates

wp\_dashboard\_setup, Removing Dashboard Widgets, Adding Your Own Dashboard Widget

wp\_enqueue\_scripts, Enqueuing Other JavaScript Libraries

wp\_network\_dashboard\_setup, Removing Dashboard Widgets

host migrations, Host Migrations

hosting, Hosting-Batcache

rolling your own server, Rolling Your Own Server-Batcache

Apache server setup, Apache server setup

Nginx in front of Apache, Nginx in front of Apache

Nginx server setup, Nginx server setup

WordPress-specific hosts, WordPress-Specific Hosts

.htaccess file, Rewrite API, Add Custom .htaccess Rules for Locking Down wp-admin

HTML

detecting HTML5 features, Feature detection in JavaScript

escaping, `esc_html( $text )`

validating and sanitizing with `wp_kses()` function, `wp_kses( $string, $allowed_html, $allowed_protocols = array() )`

HTTP, SSL Certificates and HTTPS, HTTP

HTTP headers, Headers

HTTPS, SSL Certificates and HTTPS, Installing an SSL Certificate on Your Server

URLs, Debugging HTTPS Issues

using symlink for HTTPS directory, Using one directory for HTTPS and HTTP traffic

hybrid app versus native app, Why Hybrid over Native?

hybrid mobile app, What Is a Hybrid Mobile App?

defined, What Is a Hybrid Mobile App?

I

/images/ directory, /images/

Imagick, Imagick

Imagine, Imagine

importing data

frameworks and libraries into themes, Including Bootstrap in Your App's Theme

parent theme's stylesheet to child themes, Creating a Child Theme for Memberlite

WP All Import plugin, WP All Import

/includes/ directory, /includes/

/includes/lib/directory, /includes/lib/

includes/settings.php file, Do You Really Need a Settings Page?

index.php file, The Template Hierarchy

rendering of custom post types, Themes and CPTs

init() method, CPT wrapper class, Keep Your CPTs and Taxonomies Together

INSERT queries, \$wpdb command for, Insert, replace, and update

interactive elements of web apps, Features of a Web App

internationalization, Localizing WordPress Apps

intervals, custom, for cron schedules, Adding Custom Intervals

Ionic Framework, Ionic Framework

IP addresses

blocking access for, Add Custom .htaccess Rules for Locking Down wp-admin

external IP address, Add Custom .htaccess Rules for Locking Down wp-admin

is\_multisite() function, is\_multisite()

iThemes Security plugin, Don't Allow Logins via wp-login.php

## J

JavaScript, Flexibility Is Important, JavaScript Frameworks and Workflow, JavaScript Frameworks and Workflow, jQuery and WordPress

deciding where to put custom code, [Where to Put Your Custom JavaScript](#)

device and feature detection, [Device and Feature Detection in JavaScript](#)

enqueueing jQuery library, [jQuery and WordPress](#)

enqueueing other libraries, [Enqueuing Other JavaScript Libraries](#)

escaping strings in, [esc\\_js\( \\$text \)](#)

events triggered by Heartbeat API, [Client-side JavaScript](#)

feature detection, [Feature detection in JavaScript](#)

files for app plugin in /js/ directory, [/js/](#)

heartbeat.js file, [Initialization](#)

using to increase performance, [Using JavaScript to Increase Performance-Custom Tables](#)

JavaScript API, [Cookie authentication](#)

JavaScript frameworks, [JavaScript Frameworks and Workflow-React](#)

JavaScript Object Notation (see JSON)

[jQuery, jQuery and WordPress](#)

AJAX calls with WordPress and, [Ajax Calls with WordPress and jQuery-Ajax Calls with WordPress and jQuery](#)

and WordPress, [jQuery and WordPress](#)

detecting window and screen sizes and other information about browsers, [Device and Feature Detection in JavaScript](#)

feature detection, [Feature detection in JavaScript](#)

jQuery(document).ready(), Ajax Calls with WordPress and jQuery, Client-side JavaScript

jQuery.ajax(), Ajax Calls with WordPress and jQuery, Managing Multiple Ajax Requests

jQuery.bind(), Ajax Calls with WordPress and jQuery

/js/ directory, /js/

JSON (JavaScript Object Notation)

data returned from AJAX calls in WordPress, Ajax Calls with WordPress and jQuery

defined, What Is JSON?

JSON Web Tokens (JWT)

defined, JSON Web Tokens

using, JSON Web Tokens

json\_encode() and json\_decode() functions, What Is JSON?

L

label

custom post types (CPTs), register\_post\_type( \$post\_type, \$args );

taxonomy, register\_taxonomy( \$taxonomy, \$object\_type, \$args )

labels array

for CPTs, register\_post\_type( \$post\_type, \$args );

for taxonomies, register\_taxonomy( \$taxonomy, \$object\_type, \$args )

LearnDash, LearnDash/AppLMS

Let's Encrypt, Installing an SSL Certificate on Your Server

libraries

feature detection, Feature detection in JavaScript

importing into themes, Including Bootstrap in Your App's Theme

third-party libraries for app plugin, /includes/lib/  
licensing

GNU General Public License, version 2 (GPLv2), You Plan to License or Sell Your Site's Technology

WordPress plugins, The General Public License, Version 2, License

Limit Login Attempts plugin, Limit Login Attempts

lname field in forms, Sample Page Template

load\_plugin\_textdomain() function, Setting the Text Domain

load\_template() function, Theme-Related WordPress Functions

load\_textdomain() function, Setting the Text Domain

load\_theme\_textdomain() function, Setting the Text Domain, Setting the Text Domain

locale, Defining Your Locale in WordPress

localization, Localizing WordPress Apps

localizing WordPress apps, Localizing WordPress Apps-Creating Your Own GlotPress Server

creating and loading translation files, [Creating and Loading Translation Files](#)-[Creating a .mo File](#)

creating a .mo file, [Creating a .mo File](#)

creating a .po file, [Creating a .po File](#)

file structure for localization, [Our File Structure for Localization](#)

generating a .pot file, [Generating a .pot File](#)

setting the text domain, [Setting the Text Domain](#)

defining your locale, [Defining Your Locale in WordPress](#)

determining need for, [Do You Even Need to Localize Your App?](#)

how it's done in WordPress, [How Localization Is Done in WordPress](#)

nonstring assets, [Creating a .mo File](#)-[Creating Your Own GlotPress Server](#)

prepping strings with translation functions, [Preparing Your Strings with Translation Functions](#)

escaping and translating simultaneously, [Escaping and Translating at the Same Time](#)

login error messages, hiding, [Hide Login Error Messages](#)

logins, [Features of a Web App](#), Add, Update, and Delete Users

disallowing logins via wp-login.php, [Don't Allow Logins via wp-login.php](#)

Theme My Login plugin, [Theme My Login](#)

M

magic methods, [Getting User Data](#)

mail() function, [WP Mail](#)

malware, protecting web applications against, [Scan, Scan, Scan!](#)

manage\_users\_columns filter, [Customizing the Users Table in the Dashboard](#)

manage\_users\_custom\_column filter, [Customizing the Users Table in the Dashboard](#)

manage\_users\_sortable\_columns filter, [Customizing the Users Table in the Dashboard](#)

map\_meta\_cap, [register\\_post\\_type\( \\$post\\_type, \\$args \);](#)

mashups, [Features of a Web App](#)

MaxMind GeoIP, [MaxMind GeoIP](#)

media queries, [Device and Display Detection in CSS](#)

Memberlite theme framework

creating child theme, [Creating a Child Theme for Memberlite](#)

Memberlite theme in SchoolPress app, [SchoolPress Uses the Memberlite Theme](#)

Members plugin, [Members, Members](#)

membership levels, [SchoolPress sample app, Membership Levels and User Roles](#)

membership plugins, [Paid Memberships Pro](#)

Memcached, [Memcached](#)

menus, [Menus](#)

dynamic, [Dynamic Menus](#)

navigation, Navigation Menus

storing posts with information for, Navigation Menu Item

menu\_icon, register\_post\_type( \$post\_type, \$args );

menu\_name, register\_post\_type( \$post\_type, \$args );

menu\_position, register\_post\_type( \$post\_type, \$args );

merchant accounts, Merchant Accounts

message body

defined, Message body

meta boxes

default, removing from dashboard pages, Removing Dashboard Widgets

meta capabilities, register\_post\_type( \$post\_type, \$args );

metadata, wp\_postmeta

(see also post meta)

with CPTs, Metadata with CPTs-add\_meta\_box( \$id, \$title, \$callback, \$screen, \$context, \$priority, \$callback\_args )

wp\_usermeta table, wp\_usermeta

meta\_key, Add, Update, and Delete Users

meta\_value, querying wp\_usermeta by, Add, Update, and Delete Users

migrations, Migrations

data, Data from databases

host, Host Migrations

platform, Platform Migrations

plugins for, Available migration plugins

minifying, Minify

.mo files, Setting the Text Domain, Creating a .mo File

mobile apps, Mobile Apps Powered by WordPress-Mobile App Use Cases

app wrapper, App Wrapper

AppPresser, AppPresser

use cases, Mobile App Use Cases

Modernizr.js library, Feature detection in JavaScript

More Privacy Options plugin, More Privacy Options

msg shortcode (example), Shortcode API

mu (must use) plugins directory, /wp-content/mu-plugins

Mullenweg, Matt, WordPress is just for blogs, Your App Needs to Be Highly Real Time

Multisite Global Search plugin, Multisite Global Search

multisite network dashboard, removing widgets, Removing Dashboard Widgets

Multisite networks, WordPress Multisite Networks

Multisite Robots.txt Manager plugin, Multisite Robots.txt Manager

multisite transients, Multisite Transients

MVC frameworks

controllers as template loader, Controllers = template loader

how MVC works, WordPress Versus Model-View-Controller Frameworks

models as plugins, Models = plugins

plugins for WordPress, WordPress Versus Model-View-Controller Frameworks

views as themes, Views = themes

MySQL, Flexibility Is Important

optimization, MySQL optimization-MySQL optimization

MySQL Workbench, Change Default Database Tables Prefix

MY\_SITE\_DOMAIN constant, Avoiding SSL Errors with the “Nuclear Option”

## N

native mobile app, What Is a Native Mobile App?

defined, What Is a Native Mobile App?

native versus hybrid mobile app, Why Hybrid over Native?

navigation menus, Navigation Menus

nav\_menu\_css\_class filter, Dynamic Menus

nested shortcodes, Nested Shortcodes

Nginx server

in front of Apache server, Nginx in front of Apache

setup, Nginx server setup

Node.js, JavaScript Frameworks and Workflow, WordPress Limitations with Asynchronous Processing, Project Gutenberg,

[Blocks, and Custom Block Types](#), [Learn JavaScript, Node.js, and React More Deeply](#)

and blocks, [Learn JavaScript, Node.js, and React More Deeply](#)  
[nonces, Nonces](#)

[check\\_ajax\\_referer\(\)](#) function, [check\\_ajax\\_referer\( \\$action = -1, \\$query\\_arg = false, \\$die = true \)](#)

using, [Cookie authentication](#)

[wp\\_create\\_nonce\(\)](#) function, [wp\\_create\\_nonce\( \\$action = -1 \)](#)

[wp\\_nonce\\_field\(\)](#) function, [wp\\_nonce\\_field\( \\$action = -1, \\$name = "\\_wpnonce", \\$referer = true , \\$echo = true \)](#)

[wp\\_nonce\\_url\(\)](#) function, [wp\\_nonce\\_url\( \\$actionurl, \\$action = -1 \)](#)

[wp\\_verify\\_nonce\(\)](#) function, [wp\\_verify\\_nonce\( \\$nonce, \\$action = -1 \)](#)

note widget, [Adding Widgets](#)

[NS Cloner: Site Copier](#), [NS Cloner: Site Copier](#)

Nuclear Option, avoiding SSL errors with, [Avoiding SSL Errors with the “Nuclear Option”](#)

## O

object caching, [Object Cache](#)

[object-cache.php](#), [advanced-cache.php](#) and [object-cache.php](#)

[oembed\\_cache](#), [oEmbed Cache](#)

offline work, [Features of a Web App](#)

Open Authorization (OAuth)

defined, OAuth authentication

flow outline, OAuth authentication

using, OAuth authentication

WordPress plugins, OAuth authentication

optimization and scaling, WordPress Optimization and Scaling-Bypassing WordPress

bypassing WordPress, Bypassing WordPress

custom tables, Custom Tables

definitions of terms, Terms

hosting, Hosting-Batcache

MySQL optimization, MySQL optimization-MySQL optimization

origin versus edge, Origin Versus Edge

selective caching, Selective Caching-Multisite Transients

testing, Testing

using Chrome Debug Bar, Chrome Debug Bar

using Siege, Siege

what to test, What to Test

using JavaScript for increased performance, Using JavaScript to Increase Performance-Custom Tables

W3 Total Cache, W3 Total Cache-GZIP Compression

origin, Terms

versus edge, Origin Versus Edge

output buffering, [/pages/](#), [Good Old WP\\_Query](#) and [get\\_posts\(\)](#)

## P

P2P plugin, [Posts 2 Posts](#)

padlocks, [Installing an SSL Certificate on Your Server](#)

page scraping, [Page scraping](#)

page templates, [Page Templates-When Should You Use a Theme Template?](#)

copying using hooks, [Using Hooks to Copy Templates](#)

loading, [Sample Page Template](#)

sample, [Sample Page Template](#)

when to use for themes, [When Should You Use a Theme Template?](#)

pages, [Page](#)

caching with W3 Total Cache, [Page Cache Settings](#)

mapping BuddyPress components to new or existing pages, [Pages](#)

Paid Memberships Pro plugin, [WordPress plugins are crap](#), [Creating Custom Roles and Capabilities](#), [Paid Memberships Pro](#), [Do You Even Need to Localize Your App?](#), [Paid Memberships Pro](#)

custom settings pages, [Ignore Standards When Adding Settings](#)

in SchoolPress sample app, [The SchoolPress Business Model](#)

SaaS (software as a service), [Setting Up SaaS with Paid Memberships Pro](#)-International and long-form addresses

using, [Paid Memberships Pro](#)

Paid Memberships Pro Register Helper plugin, Adding Registration and Profile Fields

passwords

encrypted, Add Custom .htaccess Rules for Locking Down wp-admin

examples of bad passwords, Use a Strong Password

payment gateways, Payment Gateways

versus merchant accounts, Merchant Accounts

PayPal, Payment Gateways

PDF generation, PDF Generation

performance, WordPress Optimization and Scaling

(see also optimization and scaling)

limitations of WordPress web apps, Your App Needs to Be Highly Real Time

permalink redirects, Rewrite API

permalink structure of a post, customizing, register\_post\_type( \$post\_type, \$args );

permalink\_epmask, register\_post\_type( \$post\_type, \$args );

PhoneGap, PhoneGap

PHP, Flexibility Is Important

classes in SchoolPress sample app, SchoolPress Has One Main Custom Plugin

device detection, Device Detection in PHP

output buffering, functions for, /pages/

server-side, in Heartbeat API, Client-side JavaScript

PHP libraries, PHP Libraries-Whoops, for user-friendly PHP errors

GD, GD

Imagick, Imagick

PHP libraries for developers, Developer Tools

php.ini file, Browser detection with PHP's get\_browser()

phpDocumentor, phpDocumentor

phpMyAdmin, Change Default Database Tables Prefix

PHPUnit, PHPUnit

platform migrations, Platform Migrations

plugins, WordPress, You Are Already Using WordPress, Plugins, Using WordPress Plugins-BuddyPress plugins

/wp-content/plugins directory, /wp-content/plugins

/wp/content/mu/plugins directory, /wp-content/mu-plugins

add-ons to existing plugins, Add-Ons to Existing Plugins

building your own, Building Your Own Plugin

community plugins, Community Plugins

BuddyPress, BuddyPress-BuddyPress plugins

criticisms concerning quality of, WordPress plugins are crap

favorites, Plugins

file headers, File Header API

adding, Adding New Headers to Plugins and Themes

file structure, [File Structure for an App Plugin-/schoolpress.php](#)

for custom settings pages, [Ignore Standards When Adding Settings](#)

for ecommerce

digital downloads, [Easy Digital Downloads](#)

membership plugins, [Paid Memberships Pro](#)

for Multisite networks, [Random Useful Multisite Plugins-Multisite Robots.txt Manager](#)

for security, [Don't Allow Logins via wp-login.php, Useful Security Plugins](#)

backup plugins, [Backup Plugins](#)

scanner plugins, [VaultPress](#)

spam blocking plugins, [Akismet](#)

free plugins, [Free Plugins](#)

installing, [Installing WordPress Plugins](#)

JavaScript code in, [Where to Put Your Custom JavaScript](#)

licensing, [The General Public License, Version 2, License](#)

loop for displaying posts, [The WordPress Loop](#)

MVC framework models as plugins, [Models = plugins](#)

not allowing admins to edit, [Don't Allow Admins to Edit Plugins or Themes](#)

plugin repository, [Using WordPress Plugins](#)

premium plugins, [Premium Plugins](#)

themes versus, Themes Versus Plugins-Where to Place Code  
When Developing Themes

user management, Plugins

using action hooks, Hooks: Actions and Filters

using custom database tables, Using custom database tables

using custom post types and taxonomies, Using CPTs and  
Taxonomies in Your Themes and Plugins-Metadata with CPTs

using filters, Filters

using global variables, WordPress Global Variables

\$wpdb, \$wpdb

using locate\_template() in, Using locate\_template in Your  
Plugins

plugins\_url() function, Enqueuing Other JavaScript Libraries

plugin\_locale filter, Setting the Text Domain

PMPro Network plugin, The SchoolPress Business Model

PMPro Register Helper plugin, The SchoolPress Business Model,  
PMPro Register Helper

.po files, Creating a .po File

post meta

functions for manipulation of, Functions Found in /wp-  
includes/post.php

storage in wp\_postmeta table, wp\_postmeta

taxonomies versus, Taxonomies Versus Post Meta

with CPTs, Metadata with CPTs-add\_meta\_box( \$id, \$title, \$callback, \$screen, \$context, \$priority, \$callback\_args )

posts, Defining and Registering CPTs

(see also custom post types)

default post types and custom post types, Custom Post Types, Post Metadata, and Taxonomies-Navigation Menu Item

attachments, Attachment

definition of posts, Post

navigation menu item, Navigation Menu Item

revisions, Revisions

display by WordPress loop, The WordPress Loop

relating taxonomies to, wp\_term\_relationships

storage in wp\_posts table, wp\_posts

themes and custom post types, Themes and CPTs

Posts 2 Posts plugin, Posts 2 Posts

post\_type\_supports, register\_post\_type( \$post\_type, \$args );

.pot file, Generating a .pot File

prepare() method, Custom SQL Statements

pre\_user\_query filter, Customizing the Users Table in the Dashboard

primitive capabilities, register\_post\_type( \$post\_type, \$args );

default, register\_post\_type( \$post\_type, \$args );

profile fields

adding, Adding Registration and Profile Fields-Adding Registration and Profile Fields

manually, Adding Registration and Profile Fields

creating for BuddyPress, Profile fields

progressive web apps (PWAs), Progressive Web Apps, Flexibility Is Important, Ionic Framework, Ionic Framework

public

```
post, register_post_type( $post_type, $args );  
register_taxonomy( $taxonomy, $object_type, $args )
```

```
taxonomy, register_taxonomy( $taxonomy, $object_type, $args )
```

```
publicly_queryable, register_post_type( $post_type, $args );
```

## Q

query() method, escaping in values passed to, Escaping in database queries

query\_var

```
post, register_post_type( $post_type, $args );
```

```
taxonomies, register_taxonomy( $taxonomy, $object_type, $args )
```

## R

Random.org, Use a Strong Password

React, JavaScript Frameworks and Workflow, React, Learn JavaScript, Node.js, and React More Deeply

React Native library, React

Redis, Redis

register\_activation\_hook() function, WP-Cron

functions adding new roles and capabilities, Creating Custom Roles and Capabilities

register\_deactivation\_hook() function, WP-Cron

register\_meta\_box\_cb, register\_post\_type( \$post\_type, \$args );

register\_nav\_menu() function, Navigation Menus

register\_nav\_menus() function, Navigation Menus

register\_post\_type() function, register\_post\_type( \$post\_type, \$args );-register\_post\_type( \$post\_type, \$args );

examples of registering custom post types, register\_post\_type( \$post\_type, \$args );

register\_sidebar() function, Defining a Widget Area

register\_taxonomy() function, register\_taxonomy( \$taxonomy, \$object\_type, \$args = array() ), register\_taxonomy( \$taxonomy, \$object\_type, \$args )-register\_taxonomy( \$taxonomy, \$object\_type, \$args )

register\_taxonomy\_for\_object\_type() function,  
register\_taxonomy\_for\_object\_type( \$taxonomy, \$object\_type )

registration

adding fields to registration page, Adding Registration and Profile Fields-Adding Registration and Profile Fields

PMPRO Register Helper plugin, PMPRO Register Helper

remove\_cap() method, Creating Custom Roles and Capabilities

remove\_meta\_box() function, Removing Dashboard Widgets

remove\_role() function, Creating Custom Roles and Capabilities

remove\_shortcode() function, Removing Shortcodes

REPLACE command (MySQL), Insert, replace, and update

Request line, Request

resizing page elements, Detecting the screen and window size with JavaScript and jQuery

responsive design, Responsive Design

browser detection and, Final Note on Browser Detection

browser detection in PHP's get\_browser(), Browser detection with PHP's get\_browser()

browser detection in WordPress core, Browser detection in WordPress core

device and display detection in CSS, Device and Display Detection in CSS

device and feature detection in JavaScript, Device and Feature Detection in JavaScript

device detection in PHP, Device Detection in PHP

REST, REST

REST API

authenticating, Authentication

cookie authentication, Cookie authentication

defined, What Is a REST API?

in WordPress, Discovery

namespace, What is a namespace?

reasons for using, [Why Use the WordPress REST API?](#)

request, [Requests](#)

request methods, [Requests](#)

response, [Responses](#)

using, [Using the WordPress REST API V2](#)

REST API endpoint, [Routes and Endpoints](#)

adding, [Adding Your Own Routes and Endpoints](#)

defined, [What is an endpoint?](#)

REST API route, [Routes and Endpoints](#)

adding, [Adding Your Own Routes and Endpoints](#)

defined, [What is a route?](#)

RESTful, [REST](#)

`restore_current_blog()` function, [restore\\_current\\_blog\(\)](#)

Retina displays, [Device and Display Detection in CSS](#)

revisions, [Revisions](#)

rewrite

`post, register_post_type( $post_type, $args );`

`taxonomy, register_taxonomy( $taxonomy, $object_type, $args )`

Rewrite API, [Rewrite API-Other Rewrite Functions](#)

adding rewrite rules, [Adding Rewrite Rules](#)

flushing rewrite rules, [Flushing Rewrite Rules](#)

other rewrite functions, [Other Rewrite Functions](#)

robots.txt files, [Multisite Robots.txt Manager](#)

roles, [What Are Roles and Capabilities?](#)

checking for a user, [Checking a User's Role and Capabilities](#)

creating custom roles, [Creating Custom Roles and Capabilities](#)

in SchoolPress sample app, [Membership Levels and User Roles](#)

Roles and Capabilities system, [User Management Is Easy and Secure with WordPress](#)

upgrading Subscribers to Authors, [Checking a User's Role and Capabilities](#)

root directory (WordPress), [Root Directory](#)

route, adding new, [Adding the /wp-sso/v1/check Route](#)

## S

SaaS (software as a service), [Setting Up SaaS with Paid Memberships Pro](#)

setting up on Paid Memberships Pro, [Step 0: Establishing How You Want to Charge for Your App-International and long-form addresses](#)

sanitize\_email() function, [Sample Page Template](#), sanitize\_email(\$email)

sanitize\_file\_name() function, [sanitize\\_file\\_name\( \\$filename \)](#)

sanitize\_option() function, [sanitize\\_option\( \\$option, \\$value \)](#)

sanitize\_text\_field() function, [Sample Page Template](#), [sanitize\\_text\\_field\( \\$str \)](#)

sanitize\_title() function, [sanitize\\_title\( \\$title, \\$fallback\\_title = "" \)](#)

sanitize\_user() function, `sanitize_user( $username, $strict = false )`

sanitizing data, Data Validation, Sanitization, and Escaping

saving block data to post meta, Saving Block Data to Post Meta

scalability, Terms

scaling, WordPress Optimization and Scaling

(see also optimization and scaling)

criticism of WordPress about, WordPress doesn't scale

defined, Terms

scanner plugins, VaultPress

/scheduled/ directory, /scheduled/

SchoolPress sample web app, What Is a Web App?

anatomy of, Anatomy of a WordPress App

business model, The SchoolPress Business Model

classes as BuddyPress groups, Classes Are BuddyPress Groups

CPTs (custom post types), Assignments Are a CPT

main custom plugin, SchoolPress Has One Main Custom Plugin

Memberlite theme, SchoolPress Uses the Memberlite Theme

membership levels and user roles, Membership Levels and User Roles

multisite version of WordPress, SchoolPress Runs on a WordPress Multisite Network

other custom plugins, [SchoolPress Uses a Few Other Custom Plugins](#)

note widget, [Adding Widgets](#)

screens

checking widths using CSS media query, [Device and Display Detection in CSS](#)

detecting size with JavaScript and jQuery, [Detecting the screen and window size with JavaScript and jQuery](#)

search engine optimization (see SEO)

Secure Sockets Layer (see SSL)

security, [Secure WordPress](#)-check\_ajax\_referer( \$action = -1, \$query\_arg = false, \$die = true )

backing up everything, [Back Up Everything!](#)

criticisms of WordPress about, [WordPress is insecure](#)

frequent security updates for WordPress, [Frequent Security Updates](#)

frequent updates of WordPress and plugins/themes, [Update Frequently](#)

hardening your WordPress install, [Hardening WordPress](#)

adding custom .htaccess rules to lock down wp-admin, [Add Custom .htaccess Rules for Locking Down wp-admin](#)

changing default database tables prefix, [Change Default Database Tables Prefix](#)

hiding login error messages, [Hide Login Error Messages](#)

hiding WordPress version, [Hide Your WordPress Version](#)

moving wp-config.php, [Move wp-config.php](#)

not allowing admins to edit plugins or themes, [Don't Allow Admins to Edit Plugins or Themes](#)

not allowing logins via wp-login.php, [Don't Allow Logins via wp-login.php](#)

not using username admin, [Don't Use the Username "admin"](#)

plugins for, [Useful Security Plugins](#)

backup plugins, [Backup Plugins](#)

scanner plugins, [VaultPress](#)

spam blocking plugins, [Akismet](#)

scanning or monitoring for attacks, [Scan, Scan, Scan!](#)

using strong password, [Use a Strong Password](#)

writing secure code, [Writing Secure Code](#)

checking user capabilities, [Check User Capabilities](#)

custom SQL statements, [Custom SQL Statements](#)

data validation, sanitation, and escaping, [Data Validation, Sanitization, and Escaping](#)

nonces, [Nonces](#)

SELECT queries, \$wpdb object methods for, [SELECT queries with \\$wpdb](#)

SEO (search engine optimization), [Plugins](#)

theme development and, [Where to Place Code When Developing Themes](#)

Yoast SEO, [Yoast SEO](#)

servers

detection in WordPress core, [Browser detection in WordPress core](#)

kicking off cron jobs from web server, [Kicking Off Cron Jobs from the Server](#)

rolling your own, [Rolling Your Own Server-Batcache](#)

sending email from, [Sending Nicer Emails with WordPress](#)

URL rewriting systems, [Rewrite API](#)

/services/ directory, [/services/](#)

\_\_set() method, WP\_User class, [Getting User Data](#)  
settings

configuring BuddyPress settings, [Settings](#)  
for Multisite networks, [Settings](#)

Settings API, [Settings API-Rewrite API](#)

deciding if you really need a settings page, [Do You Really Need a Settings Page?](#)

ignoring standards when adding settings, [Ignore Standards When Adding Settings](#)

using hook or filter instead of settings page, [Could You Use a Hook or Filter Instead?](#)

using standards when adding settings, [Use Standards When Adding Settings](#)

set\_transient() function, [Multisite Transients](#)

[shortcodes](#), [Shortcode API](#)-[Other Useful Shortcode-Related Functions](#)

[attributes](#), [Shortcode Attributes](#)

[creating](#), with [attributes](#) and enclosed content, [Shortcode API](#)

[nested](#), [Nested Shortcodes](#)

[other useful functions](#) for, [Other Useful Shortcode-Related Functions](#)

[removing](#), [Removing Shortcodes](#)

[using in widgets](#), [Before You Add Your Own Widget](#)

[shortcode\\_atts\(\)](#) function, [Shortcode Attributes](#)

[shortcode\\_parse\\_atts\(\)](#) function, [Other Useful Shortcode-Related Functions](#)

[show\\_admin\\_column](#), [register\\_taxonomy\( \\$taxonomy, \\$object\\_type, \\$args \)](#)

[show\\_in\\_admin\\_bar](#), [register\\_post\\_type\( \\$post\\_type, \\$args \)](#);

[show\\_in\\_menu](#), [register\\_post\\_type\( \\$post\\_type, \\$args \)](#);

[show\\_in\\_nav\\_menus](#), [register\\_post\\_type\( \\$post\\_type, \\$args \)](#);

[taxonomy](#), [register\\_taxonomy\( \\$taxonomy, \\$object\\_type, \\$args \)](#)

[show\\_tagcloud](#), [register\\_taxonomy\( \\$taxonomy, \\$object\\_type, \\$args \)](#)

[show\\_ui](#)

[custom post type \(CPT\)](#), [register\\_post\\_type\( \\$post\\_type, \\$args \)](#);

[taxonomy](#), [register\\_taxonomy\( \\$taxonomy, \\$object\\_type, \\$args \)](#)

[show\\_user\\_profile](#), hooking into, [Adding Registration and Profile Fields](#)

sidebars, Defining a Widget Area

(see also widgets)

embedding widget outside of dynamic sidebar, Embedding a Widget Outside of a Dynamic Sidebar

Siege, Siege

single events, scheduling, Scheduling Single Events

single.php file, Themes and CPTs

creating for registered CPTs, The Theme Archive and Single Template Files

site\_url() function, Avoiding SSL Errors with the “Nuclear Option”

slug-name.php file, Theme-Related WordPress Functions

Snappy, Snappy, Snappy

software as as service (see SaaS)

software-as-a-service (SaaS), Ecommerce

source code, distributed, The General Public License, Version 2, License

spam blocking plugins, Akismet

sp\_assignments\_dashboard\_widget() function, Adding Your Own Dashboard Widget

sp\_assignments\_dashboard\_widget\_configuration() function, Adding Your Own Dashboard Widget

sp\_manage\_users\_custom\_column() function, Customizing the Users Table in the Dashboard

sp\_stub, /pages/

SQL (Structured Query Language)

CREATE TABLE statement, Using custom database tables

updating existing table names in database with new prefix,  
Change Default Database Tables Prefix

writing custom statements, Custom SQL Statements

SQL clients, Change Default Database Tables Prefix

SQL injection attacks, Change Default Database Tables Prefix

SSL, SSL Certificates and HTTPS, WordPress Login and WordPress Administrator over SSL, Avoiding SSL Errors with the “Nuclear Option”

SSL (Secure Sockets Layer), SSL Certificates and HTTPS

SSL certificates, Installing an SSL Certificate on Your Server, Installing an SSL Certificate on Your Server, Installing an SSL Certificate on Your Server

State of WordPress presentation (Mullenweg), WordPress is just for blogs

Status Code header, Headers

Status line, Request

Stripe, Payment Gateways

strip\_shortcodes() function, Other Useful Shortcode-Related Functions

strong passwords, Use a Strong Password

str\_replace() function, Debugging HTTPS Issues

style.css file, The Template Hierarchy

for themes, [Style.css](#)  
child themes, [Creating a Child Theme for Memberlite](#)  
versioning, [Versioning Your Theme's CSS Files](#)  
styling, [Defining a Widget Area](#)  
(see also CSS; themes)  
for widgets and titles, [Defining a Widget Area](#)  
subdirectories, [Setting Up a Multisite Network](#)  
subdomains, [Setting Up a Multisite Network](#)  
setting up, [Setting Up a Multisite Network](#)  
[SUBDOMAIN\\_INSTALL](#), [Setting Up a Multisite Network](#)  
Subscriber role, [What Are Roles and Capabilities?](#)  
upgrading to Author, [Checking a User's Role and Capabilities](#)  
Sucuri, [Scan, Scan, Scan!](#)  
Super Admin role, [What Are Roles and Capabilities?](#), [Users](#)  
supports array (CPTs), [register\\_post\\_type\( \\$post\\_type, \\$args \)](#)  
switch\_to\_blog() function, [switch\\_to\\_blog\( \\$new\\_blog \)](#)  
symlinks, [Using one directory for HTTPS and HTTP traffic](#)

**T**

Tag Cloud Widget, taxonomy's inclusion in, [register\\_taxonomy\( \\$taxonomy, \\$object\\_type, \\$args \)](#)  
task focus in web apps, [Features of a Web App](#)

taxonomies, What Is a Taxonomy and How Should I Use It?-  
register\_taxonomy\_for\_object\_type( \$taxonomy, \$object\_type )

creating custom taxonomies, Creating Custom Taxonomies

register\_taxonomy() function, register\_taxonomy( \$taxonomy, \$object\_type, \$args )-register\_taxonomy( \$taxonomy, \$object\_type, \$args )

register\_taxonomy\_for\_object\_type() function,  
register\_taxonomy\_for\_object\_type( \$taxonomy, \$object\_type )

custom post type (CPT), register\_post\_type( \$post\_type, \$args );

defined, What Is a Taxonomy and How Should I Use It?

keeping together with CPTs, Keep Your CPTs and Taxonomies Together

relating taxonomy terms to posts, wp\_term\_relationships  
terms and, wp\_terms

using custom taxonomies in themes and plugins, Using CPTs and Taxonomies in Your Themes and Plugins-Metadata with CPTs

versus post meta, Taxonomies Versus Post Meta

wp-term\_taxonomy table, wp\_term\_taxonomy

Template Hierarchy, Controllers = template loader, The Template Hierarchy

documentation, The Template Hierarchy

template loader, MVC controllers versus, Controllers = template loader

templates

locating in plugins, [Using locate\\_template in Your Plugins](#)

page templates, [Page Templates-When Should You Use a Theme Template?](#)

terms, [wp\\_terms](#), Taxonomies Versus Post Meta

[wp\\_term\\_relationships](#) table, [wp\\_term\\_relationships](#)

text widgets, [Widgets API](#)

uses of, [Before You Add Your Own Widget](#)

<textarea> element, encoding text for, [esc\\_textarea\( \\$text \)](#)

Theme Developer Handbook, [Themes](#)

Theme My Login, [Theme My Login](#)

themes, [Themes-Final Note on Browser Detection](#)

[/wp/content/themes](#) directory, [/wp-content/themes](#)

and custom post types, [Themes and CPTs](#)

creating child theme for Memberlite, [Creating a Child Theme for Memberlite](#)

embedding widget area into, [Defining a Widget Area](#)

embedding widget directly into using the\_widget(), [Embedding a Widget Outside of a Dynamic Sidebar](#)

file header information, getting, [File Header API](#)

file headers, adding, [Adding New Headers to Plugins and Themes](#)

files containing the WordPress loop, [The WordPress Loop](#)

for WordPress Multisite networks, [Themes](#)

functions for, [Theme-Related WordPress Functions](#)

using `locate_template()` in your plugins, [Using locate\\_template in Your Plugins](#)

`functions.php` file, [functions.php](#)

including Bootstrap in app's theme, [Including Bootstrap in Your App's Theme](#)

JavaScript code in, [Where to Put Your Custom JavaScript](#)

licensing, [The General Public License, Version 2, License](#)

Memberlite theme in SchoolPress app, [SchoolPress Uses the Memberlite Theme](#)

menus, [Menus](#)

dynamic menus, [Dynamic Menus](#)

navigation menus, [Navigation Menus](#)

MVC views and, [Views = themes](#)

not allowing admins to edit, [Don't Allow Admins to Edit Plugins or Themes](#)

page templates, [Page Templates-When Should You Use a Theme Template?](#)

using hooks to copy templates, [Using Hooks to Copy Templates](#)

when to use theme template, [When Should You Use a Theme Template?](#)

registering sidebar for, [Defining a Widget Area](#)

[responsive design](#), [Responsive Design](#)

[browser detection in PHP's get\\_browser\(\)](#), [Browser detection with PHP's get\\_browser\(\)](#)

[browser detection in WordPress core](#), [Browser detection in WordPress core](#)

[device and display detection in CSS](#), [Device and Display Detection in CSS](#)

[device and feature detection in JavaScript](#), [Device and Feature Detection in JavaScript](#)

[device detection in PHP](#), [Device Detection in PHP](#)

[style.css file](#), [Style.css](#)

[versioning](#), [Versioning Your Theme's CSS Files](#)

[template hierarchy](#), [The Template Hierarchy](#)

[theme frameworks](#), [Popular Theme Frameworks](#)

[Genesis](#), [Genesis](#)

[non-WP frameworks](#), [Non-WordPress Theme Frameworks](#)

[\\_s](#), [\\_s \(underscores\)](#)

[using custom post types and taxonomies in](#), [Using CPTs and Taxonomies in Your Themes and Plugins-Metadata with CPTs](#)

[theme archive and single template files](#), [The Theme Archive and Single Template Files](#)

[versus plugins](#), [Themes Versus Plugins-Where to Place Code When Developing Themes](#)

[when developing apps](#), [Where to Place Code When Developing Apps](#)

when developing plugins, [When Developing Plugins](#)

when developing themes, [Where to Place Code When Developing Themes](#)

the\_content filter, [Nested Shortcodes](#)

the\_post() function, [The WordPress Loop](#)

the\_widget() function, [Embedding a Widget Outside of a Dynamic Sidebar](#)

transients, [The Transient API-Multisite Transients](#)

multisite, [Multisite Transients](#)

translate() function, [\\_\\_\( \\$text, \\$domain = “default” \)](#)

translation files, creating and loading, [Creating and Loading Translation Files-Creating a .mo File](#)

translation functions, [Preparing Your Strings with Translation Functions](#)

Twenty Nineteen theme, [The Template Hierarchy](#)

Twenty Thirteen theme, [Defining a Widget Area](#)

Twilio, [Twilio](#)

## U

UI frameworks, [Non-WordPress Theme Frameworks](#)

UPDATE queries, [Insert, replace, and update](#)

updates, managing for Multisite networks, [Updates](#)

update\_blog\_details() function, [update\\_blog\\_details\( \\$blog\\_id, \\$details = array\(\) \)](#)

`update_blog_option()` function, `update_blog_option( $id, $option, $value )`

`update_blog_status()` function, `update_blog_status( $blog_id, $pref, $value )`

`update_comment_meta()` function, `add_comment_meta( $comment_id, $meta_key, $meta_value, $unique = false )`

`update_count_callback`, `register_taxonomy( $taxonomy, $object_type, $args )`

`update_option()` function, `add_option( string $option, mixed $value = "", string $deprecated = "", string|bool $autoload = 'yes' )`

`update_post_meta()` function, `get_post_meta( $post_id, $key = "", $single = false )`, Taxonomies Versus Post Meta

`update_term_meta()` function, `get_term_meta( $term_id, $key = "", $single = false )`

`update_user_meta()` function, `get_user_meta( $user_id, $key = "", $single = false )`, Add, Update, and Delete Users

updating users, Add, Update, and Delete Users

uploads, /wp/content/uploads directory, /wp-content/uploads

URL rewriting (see Rewrite API)

URLs

adding nonces to, `wp_nonce_url( $actionurl, $action = -1 )`

escaping, `esc_url( $url, $protocols = null, $context = 'display' )`

for AJAX queries, Ajax Calls with WordPress and jQuery

`plugins_url()` function, Enqueuing Other JavaScript Libraries

user agent strings, [Device Detection in PHP](#)

User class (see [WP\\_User class](#))

user management, [User Management Is Easy and Secure with WordPress](#)

adding registration and profile fields, [Adding Registration and Profile Fields-Adding Registration and Profile Fields](#)

adding, updating, and deleting users, [Add, Update, and Delete Users](#)

customizing users table in dashboard, [Customizing the Users Table in the Dashboard](#)

getting user data, [Getting User Data](#)

hooks and filters, [Hooks and Filters](#)

plugins for, [Plugins](#)

roles and capabilities, [What Are Roles and Capabilities?](#)

checking, [Checking a User's Role and Capabilities, Check User Capabilities](#)

WordPress Multisite networks, [Users](#)

user meta

accessing in wp\_usermeta table, [Getting User Data](#)

updating, [Add, Update, and Delete Users](#)

user roles (see roles)

usernames

admin username, not using, [Don't Use the Username “admin”](#)

sanitizing, [sanitize\\_user\( \\$username, \\$strict = false \)](#)

users

extending WP\_User class, Extending the WP\_User Class

not trusting, Data Validation, Sanitization, and Escaping

user\_can() function, Checking a User's Role and Capabilities, Check User Capabilities, user\_can( \$user, \$capability )

user\_login, Add, Update, and Delete Users

user\_register, Hooks and Filters

## V

validation of data, Data Validation, Sanitization, and Escaping

email addresses, wp\_kses\_post( \$data )

wp\_kses() function, wp\_kses( \$string, \$allowed\_html, \$allowed\_protocols = array() )

Varnish, Varnish

VaultPress plugin, VaultPress

versions

hiding your WordPress version, Hide Your WordPress Version

updating for WordPress and plugins, Update Frequently

## W

W3 Total Cache, W3 Total Cache

CDNs (content delivery networks), CDNs

database caching, Database Caching

GZIP compression, GZIP Compression

minifying, [Minify](#)

object cache, [Object Cache](#)

Page Cache settings, [Page Cache Settings](#)

W3 Total Cache plugin, [W3 Total Cache](#)

web apps

defined, [What Is a Web App?](#)

features of, [Features of a Web App](#)

scanning or monitoring for attacks, [Scan, Scan, Scan!](#)

web services, [PHP Libraries](#)

MaxMind GeoIP, [MaxMind GeoIP](#)

web services and external APIs, [External APIs and Web Services](#)

websites

defined, [What Is a Website?](#)

typical progression for lean startup running on WordPress,  
[Flexibility Is Important](#)

widgets, [Widgets API-Adding Your Own Dashboard Widget](#)

adding, [Adding Widgets](#)

defining a widget area, [Defining a Widget Area](#)

embedding widget outside of dynamic sidebar, [Embedding a Widget Outside of a Dynamic Sidebar](#)

checking out existing widgets, [Before You Add Your Own Widget](#)

dashboard, [Dashboard Widgets API](#)

adding your own, [Adding Your Own Dashboard Widget](#)  
removing, [Removing Dashboard Widgets](#)

windows, browser, detecting size of, [Detecting the screen and window size with JavaScript and jQuery](#)

WooCommerce plugin, [WooCommerce](#), [Choosing a Plugin](#)  
and AppCommerce, [WooCommerce plugins](#)  
and AppWoo plugin, [WooCommerce plugins](#)  
custom settings pages, [Ignore Standards When Adding Settings](#)  
plugin and extensions, [The WooCommerce plugin and extensions](#)  
using, [Example: Hide sale banners for paying customers](#)

WordFence, [WordFence](#)

WordPress

anatomy of a WordPress app, [Anatomy of a WordPress App](#)-  
[SchoolPress Uses the Memberlite Theme](#)

as application framework, [WordPress as an Application Framework](#)

MVC frameworks versus, [WordPress Versus Model-View-Controller Frameworks](#)

building web apps with, [Building Web Apps with WordPress](#)

reasons to use WordPress, [Why Use WordPress?](#)

responses to common criticisms, [Responses to Some Common Criticisms of WordPress](#)

when not to use WordPress, [When Not to Use WordPress](#)

database structure, [WordPress Database Structure](#)  
`wp_set_object_terms( $object_id, $terms, $taxonomy, $append = false )`

directory structure, [WordPress Directory Structure](#)-/wp-content/mu-plugins

[jQuery](#), [jQuery and WordPress](#)

limitations with asynchronous processing, [WordPress Limitations with Asynchronous Processing](#)

optimization and scaling (see optimization and scaling)

[plugin repository](#), [Using WordPress Plugins](#)

theme frameworks, [WordPress Theme Frameworks](#)

WordPress Multisite networks, [WordPress Multisite Networks](#)-  
Functions We Didn't Mention

basic functionality, [Basic Multisite Functionality](#)

`$blog_id, $blog_id`

`add_user_to_blog(), add_user_to_blog( $blog_id, $user_id, $role )`

`create_empty_blog(), create_empty_blog( $domain, $path, $weblog_title, $site_id = 1 )`

`delete_blog_option(), delete_blog_option( $id, $option )`

`get_blog_details(), get_blog_details( $fields = null, $get_all = true )`

`get_blog_option(), get_blog_option( $id, $option, $default = false )`

`get_blog_post(), get_blog_post( $blog_id, $post_id )`

`get_blog_status()`, `get_blog_status( $id, $pref )`  
`get_current_blog_id()`, `get_current_blog_id()`  
`is_multisite()`, `is_multisite()`  
`restore_current_blog()`, `restore_current_blog()`  
`switch_to_blog()`, `switch_to_blog( $new_blog )`  
`update_blog_details()`, `update_blog_details( $blog_id, $details = array() )`  
`update_blog_option()`, `update_blog_option( $id, $option, $value )`  
`update_blog_status()`, `update_blog_status( $blog_id, $pref, $value )`

database structure, Multisite Database Structure-Shared Site Tables

individual site tables, Individual Site Tables

network-wide tables, Multisite Database Structure

shared site tables, Shared Site Tables

managing, Managing a Multisite Network

dashboard, Dashboard

settings, Settings

sites, Sites

themes, Themes

updates, Updates

users, Users

plugins, Random Useful Multisite Plugins-Multisite Robots.txt Manager

setting up a network, Setting Up a Multisite Network

WordPress REST API, WordPress REST API-Example: Check whether a certain email address has a membership

popular plugins that use, Popular Plugins Using the WordPress REST API

WordPress Site Health Tool, The WordPress Site Health Tool

work, offline, Features of a Web App

WP Admin plugin, Don't Allow Logins via wp-login.php

WP All Import plugin, WP All Import

WP Mail, WP Mail-Sending Nicer Emails with WordPress

sending nicer emails with, Sending Nicer Emails with WordPress

WP Single Sign-On plugin, Adding Your Own Routes and Endpoints

and checking user credentials, Using the Endpoint We Set Up to Check User Credentials

setting up the, Setting Up the WordPress Single Sign-On Plugin

wp-admin directory, /wp-admin

locking down, Add Custom .htaccess Rules for Locking Down wp-admin

wp-config.php file, Root Directory

moving for security reasons, Move wp-config.php

/wp-content directory, /wp-content

/wp-content/plugins directory, [/wp-content/plugins](#)

WP-Cron, [WP-Cron-Using Server Cron Only](#)

kicking off cron jobs from the server, [Kicking Off Cron Jobs from the Server](#)

scheduling single events, [Scheduling Single Events](#)

using server crons only, [Using Server Crons Only](#)

wp-cron.php file, [Kicking Off Cron Jobs from the Server](#)

WP-Doc plugin, [Could You Use a Hook or Filter Instead?](#)

/wp-includes directory, [/wp-includes](#)

/wp-includes/comment.php file, functions in, [Functions Found in /wp-includes/comment.php](#), [Functions Found in /wp-includes/comment.php](#)

/wp-includes/option.php file, functions in, [Functions Found in /wp-includes/option.php](#)

/wp-includes/pluggable.php file, [Functions Found in /wp-includes/...](#)

/wp-includes/post.php file, [Functions Found in /wp-includes/post.php](#), [Functions Found in /wp-includes/post.php](#)

/wp-includes/taxonomy.php file, functions in, [Functions Found in /wp-includes/taxonomy.php](#), [/wp-includes/taxonomy.php](#)

wp-includes/vars.php file, [Browser detection in WordPress core](#)

wp-login.php file, [Theme My Login](#)

(see also logins)

not allowing logins via, [Don't Allow Logins via wp-login.php](#)

/wp/content/mu/plugins directory, [/wp-content/mu-plugins](#)

/wp/content/themes directory, [/wp-content/themes](#)

/wp/content/uploads directory, [/wp-content/uploads](#)

wpdoc\_caps, Could You Use a Hook or Filter Instead?

wpdoc\_template\_redirect() function, Could You Use a Hook or Filter Instead?

wpmu\_delete\_user() function, Add, Update, and Delete Users

wp\_add\_dashboard\_widget() function, [Adding Your Own Dashboard Widget](#)

wp\_ajax\_nopriv\_{action} hook, [Ajax Calls with WordPress and jQuery](#)

wp\_ajax\_{action} hook, [Ajax Calls with WordPress and jQuery](#)

WP\_ALLOW\_MULTISITE, [Setting Up a Multisite Network](#)

wp\_blog\_versions table, [wp\\_blog\\_versions](#)

wp\_comments table, [wp\\_comments](#)

functions for interactions with, [Functions Found in /wp-includes/comment.php](#)

wp\_commentsmeta table, [wp\\_commentsmeta](#)

functions for interactions with, [Functions Found in /wp-includes/comment.php](#)

wp\_create\_user() function, [wp\\_create\\_user\( \\$username, \\$password, \\$email \)](#)

wp\_dashboard\_setup, [Removing Dashboard Widgets, Adding Your Own Dashboard Widget](#)

wp\_default\_styles filter, [Versioning Your Theme's CSS Files](#)

`wp_delete_post()` function, `wp_delete_post( $postid = 0, $force_delete = false )`

`wp_delete_term()` function, `wp_delete_term( $term, $taxonomy, $args = array() )`

`wp_delete_user()` function, `wp_delete_user( $id, $reassign = 'novalue' )`, Add, Update, and Delete Users

`wp_email` filter, Sending Nicer Emails with WordPress

`wp_enqueue_script()` function, jQuery and WordPress

`wp_enqueue_scripts` hook, Enqueuing Other JavaScript Libraries

`wp_enqueue_style()` function, Versioning Your Theme's CSS Files

media query in, Device and Display Detection in CSS

`wp_get_object_terms()` function, `wp_get_object_terms( $object_ids, $taxonomies, $args = array() )`

`wp_get_theme()` function, File Header API

`wp_head` hook, jQuery and WordPress

`wp_insert_comment` function, `wp_insert_comment( $commentdata )`

`wp_insert_post()` function, `wp_insert_post( $postarr, $wp_error = false )`

`wp_insert_term()` function, `wp_insert_term( $term, $taxonomy, $args = array() )`

`wp_insert_user()` function, `wp_insert_user( $userdata )`, Add, Update, and Delete Users

`wp_is_mobile()` function, Browser detection in WordPress core

`wp_kses()` function, `wp_kses( $string, $allowed_html, $allowed_protocols = array() )`

`wp_mail()` function, [WP Mail](#)

sending nicer emails, [Sending Nicer Emails with WordPress](#)

`wp_mail_content_type` filter, [Sending Nicer Emails with WordPress](#)

`wp_mail_from` filter, [Sending Nicer Emails with WordPress](#)

`wp_mail_from_name` filter, [Sending Nicer Emails with WordPress](#)

`wp_nav_menu()` function, [Navigation Menus](#)

`wp_network_dashboard_setup`, [Removing Dashboard Widgets](#)

`wp_nonce_field()` function, `wp_nonce_field( $action = -1, $name = "wpnonce", $referer = true , $echo = true )`

`wp_nonce_url()` function, `wp_nonce_url( $actionurl, $action = -1 )`

`wp_options` table, [wp\\_options](#)

functions for interactions with, [Functions Found in /wp-includes/option.php](#)

`wp_user_roles` option, [Creating Custom Roles and Capabilities](#)

`wp_p2p` table, [Posts 2 Posts](#)

`wp_p2pmeta` table, [Posts 2 Posts](#)

`WP_Post` class

extending versus wrapping, [Extending WP\\_Post Versus Wrapping It](#)

`wp_postmeta` table, [wp\\_postmeta](#), [Taxonomies Versus Post Meta](#)

functions for interactions with, [Functions Found in /wp-includes/post.php](#)

`wp_posts` table, [wp\\_posts](#)

limiting number of revisions stored in, [Revisions](#)

## WP\_POST\_REVISIONS, Revisions

WP\_Query class, `get_posts( $args = null )`, Good Old WP\_Query and `get_posts()`

`wp_registration_log` table, `wp_registration_log`

## WP\_Rewrite class, Other Rewrite Functions

`wp_schedule_event()` function, WP-Cron

intervals, Adding Custom Intervals

`wp_schedule_single_event()` function, Scheduling Single Events

`wp_set_object_terms()` function, `wp_set_object_terms( $object_id, $terms, $taxonomy, $append = false )`

`wp_signon()` function, Add, Update, and Delete Users

`wp_signups` table, `wp_signups`

`wp_site` table, `wp_site`

`wp_sitemeta` table, `wp_sitemeta`

`wp_specialchars()` function, `esc_html( $text )`

`wp_termmeta` table

functions for interactions with, `get_term_meta( $term_id, $key = "", $single = false )`

`wp_terms` table, `wp_terms`, Taxonomies Versus Post Meta

functions for interactions with, Functions Found in /wp-includes/taxonomy.php

`wp_terms_relationships` table, Taxonomies Versus Post Meta

`wp_terms_taxonomy` table, Taxonomies Versus Post Meta

wp\_term\_relationships table, wp\_term\_relationships

wp\_term\_taxonomy table, wp\_term\_taxonomy

wp\_update\_comment() function, wp\_update\_comment( \$commentarr )

wp\_update\_post() function, wp\_update\_post( \$postarr = array(), \$wp\_error = false )

wp\_update\_term() function, wp\_update\_term( \$term\_id, \$taxonomy, \$args = array() )

wp\_update\_user() function, wp\_update\_user( \$userdata ), Add, Update, and Delete Users

WP\_User class, get\_user\_by( \$field, \$value )

extending, Extending the WP\_User Class

Teacher and Student classes, Extending the WP\_User Class-Extending the WP\_User Class

getting a WP\_User object to work with, Getting User Data

getting user data from WP\_User object, Getting User Data

using overloaded properties or \_\_get() magic method, Getting User Data

wp\_usermeta table, wp\_usermeta, Shared Site Tables

accessing data stored in, Getting User Data

functions for interactions with, get\_user\_meta( \$user\_id, \$key = "", \$single = false )

storing arrays in, different methods, Add, Update, and Delete Users

wp\_users table, wp\_users

accessing data stored in, [Getting User Data](#)  
demonstration of functions interacting with, `wp_delete_user($id, $reassign = 'novalue')`  
[wp\\_user\\_roles](#) option, [Creating Custom Roles and Capabilities](#)  
[wp\\_verify\\_nonce\(\)](#) function, `wp_verify_nonce( $nonce, $action = -1 )`  
WP\_Widget class, [Adding Widgets](#)  
wrapper classes for CPTs, [Custom Wrapper Classes for CPTs](#)-  
[Wrapper Classes Read Better](#)  
extending WP\_Post vs. wrapping it, [Extending WP\\_Post Versus Wrapping It](#)  
keeping CPT functionality together, [Keep It in the Wrapper Class](#)  
keeping CPTs and taxonomies together, [Keep Your CPTs and Taxonomies Together](#)  
making code easier to read, [Wrapper Classes Read Better](#)  
reasons for using, [Why Use Wrapper Classes?](#)

WYSIWYG editor, [Content Management Is Easy with WordPress](#)

Y

Yoast SEO, [Yoast SEO](#)

Z

Zebra\_Image, [Zebra\\_Image](#)

## About the Authors

**Brian Messenlehner** started his career as a software developer for the United States Marine Corps in 2000. Brian is the cofounder of AlphaWeb.com, AppPresser.com, and SchoolPresser.com, all web-based companies that specialize in custom WordPress and mobile app development. Brian has been working with WordPress at an expert level since 2008, building custom web and mobile solutions for clients such as *TIME* magazine, NBC, Microsoft, Discovery Channel, Constant Contact, Uber, Campbell's Soup, HEB, Starbucks, YMCA, Newark New Jersey Public Schools, and the National Park Services, to name a few. Find Brian on Twitter [@bmess](#).

**Jason Coleman** is the CEO of Stranger Studios and lead developer of Paid Memberships Pro, a membership platform for WordPress. He has been developing applications in PHP and on top of WordPress for over five years. Jason enjoys helping his customers *get paid* through Paid Memberships Pro, enabling them to start new businesses and expand current ones. You can find Jason at <https://therealjasoncoleman.com> or on Twitter [@jason\\_coleman](#).

## Colophon

The animal on the cover of *Building Web Apps with WordPress*, Second Edition, is the green iguana (*Iguana iguana*). This reptile is native to Central and South America and the Caribbean Islands, with a mainland range extending from Mexico to southern Brazil.

Additional populations in South Florida, Hawaii, and the Rio Grande Valley descend from escaped or abandoned pets. When the Spanish arrived in the Bahamas in 1492 the island where they made first landfall was known to its Taino inhabitants as Guanahani, or “iguana island,” and the indigenous name for these creatures was absorbed into Spanish as *iguana*.

Iguanas are large lizards; they grow up to 7 feet long (nose to tail), and can weigh up to 20 pounds. Females dig deep holes in which to lay their 40 or so eggs, leaving their young to hatch and be self-sufficient from birth. Iguanas eat mostly fruits and other plant matter, the types of which vary across their range. Despite its common name of *green iguana*, this species can be many different colors, depending on their area of origin. In the southern part of their range, iguanas appear more bluish in color, with bright blue markings. On islands like Aruba and Grenada, their skin can be lavender or black; individuals from the west coast of Costa Rica are red; and Mexican iguanas tend to be a light orange.

In the trees, green iguanas are excellent climbers as well as resilient fallers; they can fall up to 50 feet without injury. With their claws and strong back legs they securely grasp branches and make long leaps from tree to tree. In addition to being at home in the canopy, iguanas

are also natural swimmers, using their powerful tails to propel them through the water. Their tails can also be used as weapons to protect them from predators or to incapacitate a rival. One special feature (which it shares with some other lizards) of the iguana's tail is that it is physically able to break off and regrow, an ability known as "caudal autotomy": if an iguana's tail is grabbed by a predator or is injured, the tail is able to simply drop off at certain points; a new tail will eventually grow out to replace the old one. The remaining section of tail moves, still animated by electric impulses, giving the iguana additional time to escape.

Because of their dramatic looks and laid-back nature, green iguanas are popular pets. However, they require specialized care, and it is sadly very common for iguanas to be abandoned or disposed of because an owner could not provide the correct environment. They require varied leafy vegetables and access to fresh water, constant tropical temperatures of around 80°F, and long daily exposure to UVA and UVB lighting. An iguana can live up to 20 years, so the decision to keep one as a pet should be given much consideration.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Dover. The cover fonts are Gilroy and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

