



Quick answers to common problems

WordPress Plugin Development Cookbook

Over 80 step-by-step recipes to extend the most popular CMS
and share your creations with its community

Yannick Lefebvre

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

WordPress Plugin Development Cookbook

Over 80 step-by-step recipes to extend the most popular CMS and share your creations with its community

Yannick Lefebvre

[PACKT] open source 
PUBLISHING community experience distilled
BIRMINGHAM - MUMBAI

WordPress Plugin Development Cookbook

Copyright © 2012 Packt Publishing

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

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

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

First published: July 2012

Production Reference: 1190712

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

ISBN 978-1-84951-768-3

www.packtpub.com

Cover Image by Andrée Caron (info@nayanna.biz)

Credits

Author

Yannick Lefebvre

Project Coordinator

Michelle Quadros

Reviewers

Liina Buckingham

Joachim Kudish

Proofreaders

Aaron Nash

Mario Cecere

Acquisition Editor

Usha Iyer

Indexer

Monica Ajmera Mehta

Lead Technical Editor

Sonali Tharwani

Production Coordinator

Aparna Bhagat

Technical Editors

Joyslita D'Souza

Veronica Fernandes

Cover Work

Aparna Bhagat

Copy Editor

Laxmi Subramanian

About the Author

Yannick Lefebvre is a plugin developer who has published eight projects to the official WordPress repository to this day. His first creation, Link Library, has been used on hundreds of sites around the world. With a background in Computer Science and working for Presagis—a company providing software tools in the modeling and simulation industry—he started writing plugins for his own WordPress site in 2004 and quickly started sharing his creations with the community. He is actively involved in the Montreal WordPress community and has presented multiple times at WordCamp Montreal. You can find out more about him and his plugins on his blog, Yannick's Corner (<http://ylefebvre.ca>).

I would like to thank the WordCamp Montreal organizers for giving me a chance to speak at multiple editions of the event, for creating great videos of the presentations, and giving me the opportunity to get involved in the community. This project would not have existed without them.

I would also like to thank Richard Archambault for his great feedback and encouragement during the writing process as well as the entire Packt Publishing team for proposing this great project to me and supporting me through the entire process.

Finally, I would like to thank my parents for always believing in me and encouraging me in all my projects.

About the Reviewers

Liina Buckingham has been developing websites since 2000. She is passionate about web usability, minimalistic design, and engaging content.

Joachim Kudish is a Web Developer specialized in WordPress development. He is currently employed at Automattic as a Code Wrangler. He works primarily at WordPress.com, while dabbling in several other of Automattic's web products. Previously, he was a freelancing web developer building high-scale WordPress sites and private WordPress plugins. He is author and contributor to several WordPress plugins as well as a contributor to WordPress core. He is originally from Montreal, Quebec but is currently located in Vancouver, BC. He is active in the local Vancouver WordPress community and is a regular WordCamp speaker.

Thank you Yannick for asking me to participate in the making of this book. It was a great learning experience and a true pleasure to be part of the project.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

*To my wife, Andrée, for her love, her patience throughout the writing process,
and being a great first proofreader*

*To my daughters, Évelyne and Gabrielle, for always making me smile and
giving the best hugs in the world*

Table of Contents

Preface	1
Chapter 1: Preparing a Local Development Environment	5
Introduction	6
Installing a web server on your computer	6
Downloading and configuring a local WordPress installation	11
Creating a local Subversion repository	15
Importing initial files to a local Subversion repository	17
Checking out files from a Subversion repository	19
Committing changes to a Subversion repository	22
Reverting uncommitted file changes	25
Viewing file history and reverting content changes to older revisions	27
Installing a dedicated code/text editor	29
Installing and configuring the NetBeans Integrated Development Environment	31
Interacting with a Subversion repository from the NetBeans interface	34
Managing a MySQL database server from the NetBeans interface	36
Chapter 2: Plugin Framework Basics	39
Introduction	39
Creating a plugin file and header	40
Adding output content to page headers using plugin actions	44
Using WordPress path utility functions to load external files and images	48
Modifying the page title using plugin filters	50
Adding text after each item's content using plugin filters	54
Inserting link statistics tracking code in page body using plugin filters	56
Troubleshooting coding errors and printing variable content	59
Creating a new simple shortcode	63
Creating a new shortcode with parameters	65
Creating a new enclosing shortcode	67

Loading a stylesheet to format plugin output	69
Writing plugins using object-oriented PHP	70
Chapter 3: User Settings and Administration Pages	73
Introduction	74
Creating default user settings on plugin initialization	74
Storing user settings using arrays	78
Removing plugin data on deletion	80
Creating an administration page menu item in the Settings menu	82
Creating a multi-level administration menu	85
Hiding items which users should not access from the default menu	87
Rendering the admin page contents using HTML	89
Processing and storing plugin configuration data	92
Displaying a confirmation message when options are saved	95
Adding custom help pages	97
Rendering the admin page contents using the Settings API	100
Accessing user settings from action and filter hooks	107
Formatting admin pages using meta boxes	109
Splitting admin code from the main plugin file to optimize site performance	115
Storing stylesheet data in user settings	117
Managing multiple sets of user settings from a single admin page	122
Chapter 4: The Power of Custom Post Types	129
Introduction	129
Creating a custom post type	130
Adding a new section to the custom post type editor	135
Displaying single custom post type items using custom templates	138
Creating an archive page for custom post types	143
Displaying custom post type data in shortcodes	146
Adding custom categories for custom post types	150
Hiding the category editor from the custom post type editor	153
Displaying additional columns in the custom post list page	157
Adding filters for custom categories to the custom post list page	161
Updating page title to include custom post data using plugin filters	164
Chapter 5: Customizing Post and Page Editors	167
Introduction	167
Adding extra fields to the post editor using custom meta boxes	168
Displaying custom post data in theme templates	172
Hiding the Custom Field section in the post editor	175
Extending the post editor to allow users to upload files directly	177

Chapter 6: Accepting User Content Submissions	183
Introduction	183
Creating a client-side content submission form	183
Saving user-submitted content in custom post types	187
Sending e-mail notifications upon new submissions	191
Implementing a captcha on user forms	194
Chapter 7: Creating Custom MySQL Database Tables	199
Introduction	199
Creating new database tables	200
Deleting custom tables on plugin removal	205
Updating custom table structure on plugin upgrade	207
Displaying custom table data in an admin page	209
Inserting and updating records in custom tables	213
Deleting records from custom tables	218
Displaying custom database table data in shortcodes	222
Implementing a search function to retrieve custom table data	224
Importing data from a user file into custom tables	227
Chapter 8: Leveraging JavaScript, jQuery, and AJAX Scripts	231
Introduction	231
Safely loading jQuery onto WordPress web pages	232
Displaying a pop-up dialog using the built-in ThickBox plugin	234
Controlling pop-up dialog display using shortcodes	237
Displaying a calendar day selector using the Datepicker plugin	240
Adding tooltips to admin page form fields using the TipTip plugin	243
Using AJAX to dynamically update partial page contents	246
Chapter 9: Adding New Widgets to the WordPress Library	253
Introduction	253
Creating a new widget in WordPress	254
Displaying configuration options	256
Validating configuration options	259
Implementing the widget display function	261
Adding a custom dashboard widget	264
Chapter 10: Enabling Plugin Internationalization	267
Introduction	267
Changing the WordPress language configuration	268
Adapting default user settings for translation	269
Making admin page code ready for translation	270
Modifying shortcode output for translation	273

Translating text strings using Poedit	275
Loading a language file in the plugin initialization	277
Chapter 11: Distributing Your Plugin on wordpress.org	281
Introduction	281
Creating a readme file for your plugin	282
Applying for your plugin to be hosted on wordpress.org	285
Uploading your plugin using Subversion	286
Providing a plugin banner image	289
Index	291

Preface

Developing plugins for WordPress is the next big thing for you if you are an administrator looking to enhance a personal site with custom functionality for which no plugin exists, a developer looking to enhance the WordPress platform with new ideas for the community, or a website designer building a specific project for a client. Learning how to create WordPress plugins will allow you to unleash the full potential of the most popular web content management system.

As an early WordPress adopter, I started building plugins to add functionality to my personal site. Once I got these new elements in place, I quickly realized that other users could benefit from these extensions, and started distributing them online. To this day, I always love hearing back from users of my creations and finding out how they have put them to use and what new functionality they think would make them even better.

While developing plugins might initially sound a little bit like black magic, this book shows you how easy creating plugins actually is through a series of step-by-step recipes. If you have previously added code to a theme's functions file, you may even be familiar with some of the mechanics explained in this book. With all of the information contained in this book, you will quickly be able to create your own plugins or dissect existing ones to add that extra bit of missing functionality that you require. Before you know, you'll be publishing your own creations to the official WordPress plugin repository!

Let's start learning how to cook up great WordPress plugins!

What this book covers

Chapter 1, Preparing a Local Development Environment, shows plugin developers how to install and configure an efficient development environment.

Chapter 2, Plugin Framework Basics, explains the basic mechanics of registering user functions with WordPress to be executed at key points when web pages are displayed, forming the basis of plugin creation.

Chapter 3, User Settings and Administration Pages, covers the creation of administration pages that will allow the users to configure the plugins you create.

Chapter 4, The Power of Custom Post Types, empowers developers to add whole new content management sections to the WordPress environment.

Chapter 5, Customizing Post and Page Editors, demonstrates how to alter the default administration post and page editing environment to add new capabilities.

Chapter 6, Accepting User Content Submissions, allows users to submit their own content to new content sections that will be managed by your plugins.

Chapter 7, Creating Custom MySQL Database Tables, leverages the power of MySQL to create custom database tables in a site database to store and retrieve custom data.

Chapter 8, Leveraging JavaScript, jQuery, and AJAX Scripts, makes plugin output very dynamic by using a number of popular script libraries.

Chapter 9, Adding New Widgets to the WordPress Library, indicates how to add new widgets that users will be able to easily drag-and-drop to add content to their web pages.

Chapter 10, Enabling Plugin Internationalization, prepares your plugin to be translated to any language to make it easier to be used by non-English speakers.

Chapter 11, Distributing Your Plugin on wordpress.org, shows you how to prepare your plugin for sharing with the global WordPress community.

What you need for this book

Chapter 1, Preparing a Local Development Environment, walks you through all of the tools that are useful to have when developing plugins for WordPress, including XAMPP, TortoiseSVN, and NetBeans.

While this book will always describe all of the steps necessary to perform its recipes, having a good understanding of WordPress will allow you to fully appreciate the information contained in these pages.

Who this book is for

This book is for WordPress users, developers, or site integrators with basic knowledge of PHP and an interest in creating new plugins to address their personal needs, client needs, or share new ideas with the WordPress community.

Conventions

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

Code words in text are shown as follows: "Save and close the `httpd.conf` file."

A block of code is set as follows:

```
add_settings_field( 'Select_List', 'Select List',
    'ch3sapi_select_list',
    'ch3sapi_settings_section', 'ch3sapi_main_section',
    array( 'name' => 'Select_List',
        'choices' => array( 'First', 'Second', 'Third' ) ) );
```

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

```
<h2>My Google Analytics</h2>

<?php if ( !empty( $_GET['message'] ) ) { ?>
    <div id="message" class="updated fade"><p><strong>Settings Saved
</strong></p></div>
<?php } ?>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Settings** section of the administration menu."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Preparing a Local Development Environment

We will cover the following topics in this chapter:

- ▶ Installing a web server on your computer
- ▶ Downloading and configuring a local WordPress installation
- ▶ Creating a local Subversion repository
- ▶ Importing initial files to a local Subversion repository
- ▶ Checking out files from a Subversion repository
- ▶ Committing changes to a Subversion repository
- ▶ Reverting uncommitted file changes
- ▶ Viewing file history and reverting content changes to older revisions
- ▶ Installing a dedicated code/text editor
- ▶ Installing and configuring the NetBeans Integrated Development Environment
- ▶ Interacting with a Subversion repository from the NetBeans interface
- ▶ Managing a MySQL database server from the NetBeans interface

Introduction

Before we start writing our first WordPress plugin, it is important to have a good set of tools in place that will allow you to work locally on your computer and be more efficient in your work. While it is possible to perform some development tasks with the built-in tools that are provided with the operating system, creating a solid local development environment will help you develop plugins quickly and have full control over your server settings to be able to test different configurations.

This chapter proposes a set of free tools that can easily be installed on your computer, regardless of your preferred operating system, to facilitate the development of your future WordPress plugins. These tools include a local web server to speed up page access and avoid sending files constantly to a remote server, a version control system to keep incremental backups of your work, a code editor for basic file editing capabilities, and an integrated development environment to accelerate your development tasks. In addition to installing and learning how to use these tools, this chapter also shows how to download and configure a local WordPress installation on a local web server.

Installing a web server on your computer

The first step to configure a local development environment is to install a local web server on your computer. This will transform your computer into a system capable of displaying web pages and performing all tasks related to rendering a WordPress website locally.

Having a local web server has many benefits:

- ▶ Provides a quick response to the frequent page refreshes that are made as plugin code is written, tested, and refined, since all information is processed locally
- ▶ Removes the need to constantly upload new plugin file versions to a remote web server to validate code changes
- ▶ Allows development to take place when no Internet connection is available (for example, when traveling on an airplane)
- ▶ Offers a worry-free programming environment where you cannot bring down a live website with a programming error or an infinite loop

There are many free packages available online that contain all of the web server components necessary to run a WordPress installation. This recipe shows you how to easily install one of these packages.

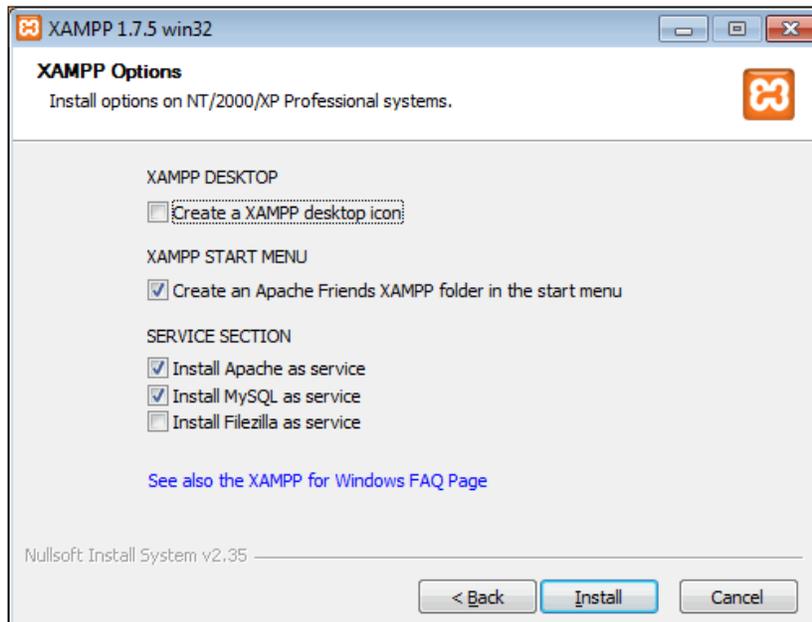
How to do it...

1. Visit the XAMPP website (<http://www.apachefriends.org/en/xampp.html>) and download the appropriate XAMPP package for your computer.

 XAMPP is available for the Windows, Mac OS X, and Linux platforms. The screenshots in this recipe were taken from XAMPP version 1.7.5 for Windows. The installation steps and exact dialog contents might vary slightly based on your choice of platform.

2. Optional on Windows: Disable the Windows **User Access Control (UAC)** feature to give full permissions to XAMPP to install itself on your system (visit <http://windows.microsoft.com/en-US/windows7/Turn-User-Account-Control-on-or-off> for more information on how to perform this procedure).
3. Launch the XAMPP installer (`xampp-win32-1.7.5-VC9-installer.exe` on the Windows platform).
4. Select your language of choice and acknowledge the warning message about User Access Control (UAC).
5. If possible, do not modify the default installation directory of `c:\xampp` since some references to this folder will be made in this book.
6. Uncheck the **Create a XAMPP Desktop icon** option, unless you want to have an icon for the web server on your desktop.
7. Check the **Install Apache as service** option to automatically start the web server when your computer starts.
8. Check the **Install MySQL as service** option to automatically start the database server when your computer starts.

9. Leave the **Install Filezilla as service** option unchecked since we will not need a local FTP server during the development of WordPress plugins.



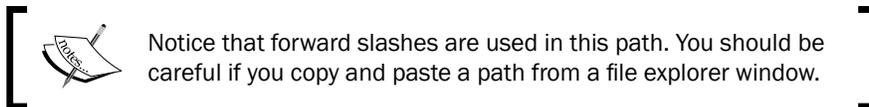
10. Click on the **Install** button to proceed with the web server installation.
11. Click on **Finish** once the installation is complete. The XAMPP installer will register the necessary services with Windows based on the options selected during the installation process.
12. Launch the **XAMPP Control Panel** using the **Start Menu** shortcut to verify the status of the web and database servers. Both the **Apache** and **MySql** services should be displayed as **Running** in **XAMPP Control Panel Application**.



13. Open a web browser and navigate to the address `http://localhost` to display your local web server's welcome page.



14. Open the `c:\xampp\apache\conf\httpd.conf` file in a text editor (for example, Notepad).
15. Search for the `DocumentRoot` configuration option and change its value to a different location on disk to avoid keeping your project files under the original installation directory. For example, you could set it to a new directory designed to hold your local development installation of WordPress, such as `DocumentRoot "C:/WPDev"`.



16. Search for the `Directory` option and change it to the same path that was used for the `DocumentRoot`, that is `<Directory "C:/WPDev">`.
17. Save and close the `httpd.conf` file.
18. Create the directory specified as `DocumentRoot`, if it does not already exist on your computer.
19. Open the **XAMPP Control Panel**.
20. Stop and re-start the **Apache** service for the new configuration to take effect.



Trying to access the local web server's welcome page will no longer work after having performed steps 14 through 20, since the new directory specified is currently empty.

How it works...

The XAMPP package contains all of the components necessary to run a web server capable of hosting a WordPress website on your computer. These components include:

- ▶ Apache web server
- ▶ PHP interpreter
- ▶ MySQL database server
- ▶ phpMyAdmin database management interface

The XAMPP package also includes an FTP server tool called FileZilla Server. We do not need to install this service since we can just locally access the web server files.

Once XAMPP is installed and started, the keyword `localhost` that we type in the web browser is recognized by the operating system as a request to communicate with the web server on the local computer and the Apache web server displays the welcome page from its documentation.

The XAMPP documentation is a set of flat HTML files located in the `c:\xampp\htdocs` directory on the Windows platform. This is the web server's default working directory.

The last few steps of the recipe instruct the Apache web server to look for the local website's content in a new directory. This is a safety precaution to be sure that site files are not deleted inadvertently if XAMPP is uninstalled. It can also help in managing multiple sites on a single computer.

There's more...

While XAMPP is a full-featured local web server package and is available on the three major operating systems, there are many others available online. Here is a list of some of the most popular ones:

- ▶ For Windows:
 - WampServer (<http://www.wampserver.com/en/>)
 - EasyPHP (<http://www.easyphp.org/>)
- ▶ For Mac OS X:
 - MAMP (<http://www.mamp.info/en/index.html>)



For a more complete list of web server packages, visit http://en.wikipedia.org/wiki/List_of_AMP_packages.

Creating a remote web development environment

If it's not possible for you to set up a local web server to develop WordPress plugins, or if you are planning to share the development tasks with one or more people, then an alternative to setting up a local web server is to create a remote development environment.

The easiest way to create such an environment, assuming that you already have a web hosting account set up, is to create a subdomain off your main domain. This will allow you to create a standalone test installation for WordPress that will still provide safety from affecting a live site but will not carry the other benefits of a local installation.

See also

- ▶ *Downloading and configuring a local WordPress installation recipe*

Downloading and configuring a local WordPress installation

The next component of our local development environment is to install WordPress on your local web server to run a fully working website and have all of its files hosted locally.

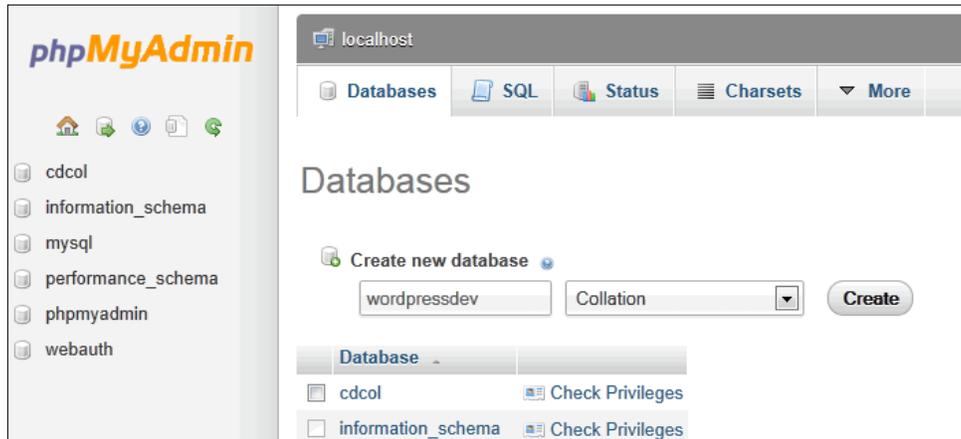
WordPress has always prided itself with its easy five-minute installation process. Installing it on a local web server is even easier and quicker than it would be on a live remote server. This recipe covers the creation of a MySQL database to store all data related to our new WordPress installation and the actual setup process.

Getting ready

This recipe assumes that you have a local web server installed on your computer. This web server can be a fresh install performed using the previous recipe or can be from a previous installation. The steps in the following section are written with a focus on new web servers. If you have created a new account to access the MySQL database or changed the root user's password, some of the steps will change slightly. The location of the phpMyAdmin tool might also be different if you are using a different web server than XAMPP. You should refer to your web server's documentation to find out what that address is.

How to do it...

1. In the web browser, navigate to the address `http://localhost/phpmyadmin/` to access your web server's database administration tool.
2. Click on the **Databases** tab in **phpMyAdmin**.
3. Type the name of the new database to be created in the empty field below the words **Create new database**. In this case, we will use the name `wordpressdev`.

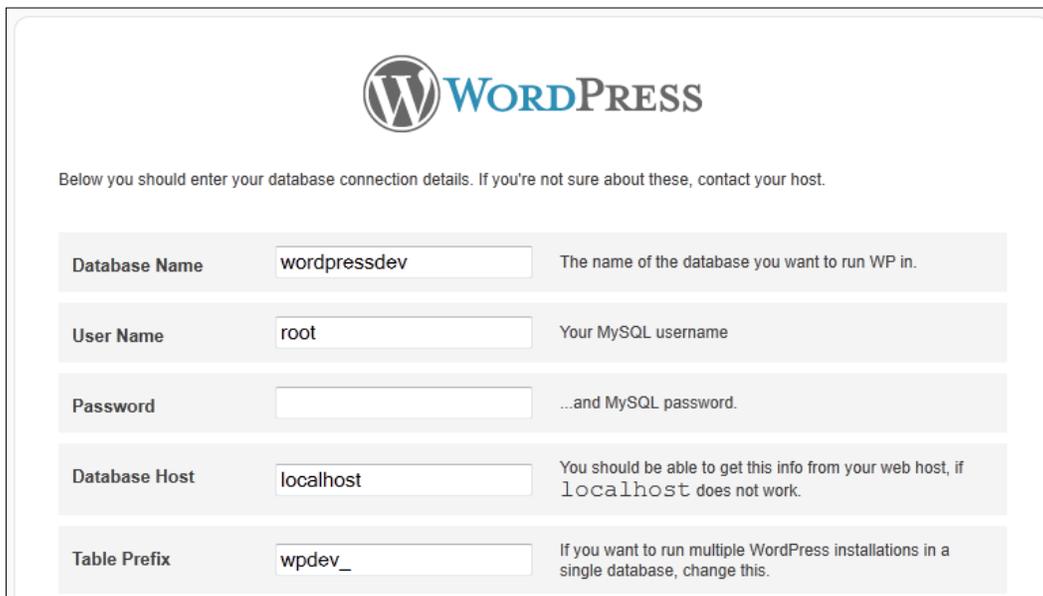


4. Click on the **Create** button to complete the database creation process.
5. Download the latest WordPress installation package from the official `wordpress.org` site. The download link can be found on the very first page of the site and the download package will work on any web server, local or remote.

 The following instructions have been tested against WordPress version 3.4. While the installation process does not usually change much between versions, there may be slight differences in these steps on newer versions.

6. Extract the WordPress archive file contents using your favorite file archiver utility or your operating system's built-in capabilities.
7. Copy the contents of the resulting `wordpress` folder to your local web server's web content directory (`c:\WPDev`, if you followed the previous recipe). You should not copy the `wordpress` folder itself unless you want the address of your WordPress website to be `http://localhost/wordpress`.

8. Direct your web browser to `http://localhost` to start the WordPress installation process. Click on the **Create a Configuration File** button to start the process. Click on the **Let's Go** button to start the configuration process.
9. Update the **Database Name** field to reflect the name of our newly-created database (`wordpressdev`).
10. Set the MySQL **User Name** to `root`.
11. Delete all characters from the MySQL **Password** to leave it empty, since local MySQL server root accounts are typically configured without any password.
12. Leave the **Database Host** field with its default value (`localhost`).
13. Change the **Table Prefix** field from its default value to `wpdev_`.

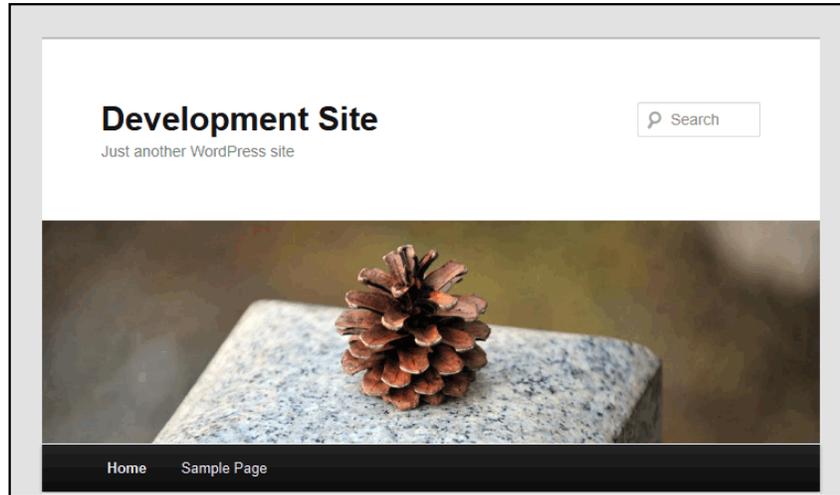


The screenshot shows the WordPress installation configuration page. At the top is the WordPress logo. Below it is a note: "Below you should enter your database connection details. If you're not sure about these, contact your host." The form consists of five rows, each with a label, an input field, and a description:

Database Name	<input type="text" value="wordpressdev"/>	The name of the database you want to run WP in.
User Name	<input type="text" value="root"/>	Your MySQL username
Password	<input type="password"/>	...and MySQL password.
Database Host	<input type="text" value="localhost"/>	You should be able to get this info from your web host, if <code>localhost</code> does not work.
Table Prefix	<input type="text" value="wpdev_"/>	If you want to run multiple WordPress installations in a single database, change this.

14. Click on the **Submit** button to validate the information entered. If any parameters are not entered correctly, or if the WordPress installation process cannot correctly access your database server, it will display an error page and give you an opportunity to make corrections. Click on the **Run the install** button for WordPress to create the required table structure in the designated MySQL database.
15. Specify a **Site Title** (for example, `Development Site`).
16. Set a **Password** for the admin user.
17. Enter your **E-mail** address in the appropriate field (although no e-mail will actually be sent on most local development installations).

18. If you are configuring a live external development server, uncheck the **Allow search engines to index this site** option since we do not want this development site to appear anywhere. Click on **Install WordPress** to complete the installation. Click on the **Log In** button to navigate to your site's login screen.
19. Click on the **Back to Development Site** link to see your new site.



How it works...

In the first few steps, the phpMyAdmin interface is used to create a database on the local MySQL server. This web-based database management tool comes bundled with XAMPP and most other web servers. The `http://localhost/phpmyadmin` address will always take you to the database administration tool, even if you relocate your web server's document root directory as documented in the previous recipe.

Once a database is created and the WordPress files have been copied to the correct location, pointing your browser to the local web server gets it to search through the document root directory to find HTML files to send back to the browser or PHP files to execute. In the case of WordPress, the web server finds the `index.php` file and executes it using its PHP interpreter. As the WordPress code is executed, it checks if a configuration file is present and launches the installation process when it does not find it. The WordPress code does not see any difference between the local web server that we are running it on and a remote live web server that would be accessible anywhere online.

While we specified an e-mail address for the administrator during the installation, some local web servers are not configured to send out e-mail messages so we will never receive any e-mail communication in these cases. It is preferable to use a remote server when developing and testing e-mail functionality in a plugin.

Once this recipe has been completed, you will have a fully functional WordPress installation in place.

Creating a local Subversion repository

Version control is an important part of any code development project to keep track of a project's history, to have full and organized backups, and to be able to easily roll back changes to get back to a known working state. Version control is also the best and most efficient way to share code and other files when developing a project in a team environment. In addition to being a great version control system that is easy to use and configure, Subversion (often referred to as SVN) is also the technology that manages all submissions on the official WordPress plugin directory. Therefore, by setting up and using a local Subversion repository during your initial plugin development, you will immediately be ready to share your creations with the community.

How to do it...

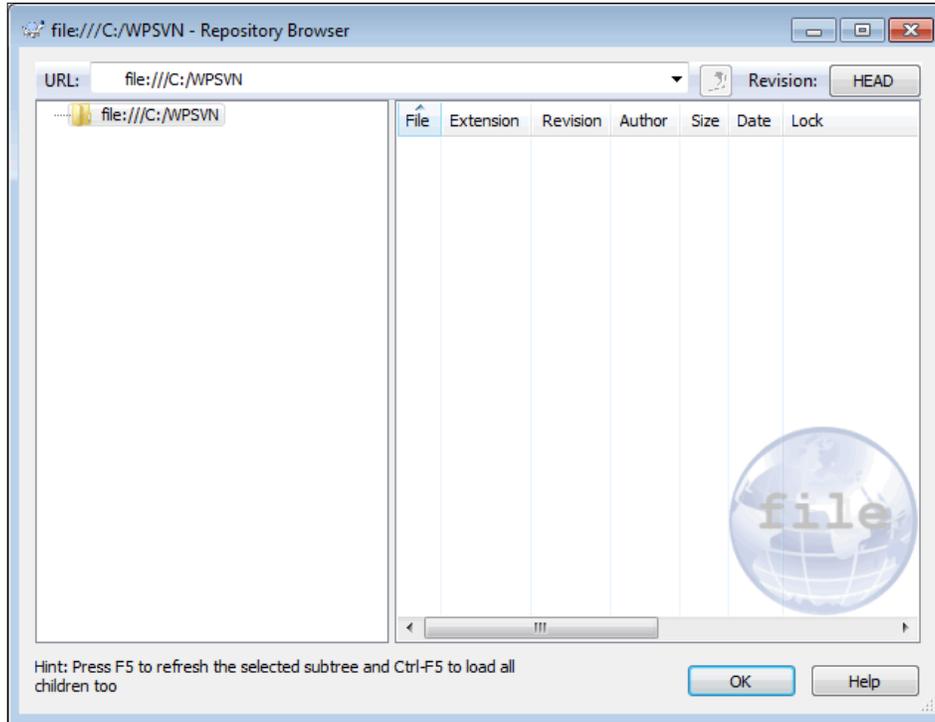
1. Visit the TortoiseSVN site (<http://tortoisesvn.net/downloads.html>) and download the free Subversion client for your version of Windows (32- or 64-bit).



While this recipe focuses on the creation of a local repository on the Windows platform, equivalent tools for other platforms are discussed after the recipe steps, in the *There's more...* section.

2. Launch the TortoiseSVN installation program and install it using all the default installation options.
3. Create a new folder on your hard drive that will host the local Subversion repository (for example, `c:\WPSVN`).
4. Right-click on the new folder and select the **TortoiseSVN | Create Repository Here** menu item. TortoiseSVN will create the required file structure in the target directory and display a message indicating that the repository has been created successfully.
5. Click on the **Start Repobrowser** button to launch the repository navigation tool.

6. Type `file:///C:/WPSVN` in the **URL** selection dialog box. At this time, TortoiseSVN displays the contents of the repository, which is currently empty.



How it works...

Subversion is a free open source version control system that is designed to keep file revisions organized and backed up over the course of a project's development, as well as provide access to older versions of all files at any time. If you have ever found yourself copying a directory on your computer and giving each copy sequentially numbered names or adding dates to their names, then you will recognize that version control is really just a more organized and efficient method of achieving the same goal of keeping backups of known working versions of code files, and being able to access any older version of a file.

While the default Subversion interface is a set of command-line utilities, TortoiseSVN and many other client applications provide graphical tools to create, access, and manage local and remote repositories.

In addition to familiarizing yourself with this system for later use on `wordpress.org`, using a local Subversion repository will ensure that you will always have older versions of your plugins easily accessible in case a code change that you perform breaks your work and you cannot figure out how to get back to a working state.

There's more...

While there are many Subversion clients available online to interact with a repository, not all of them include the necessary administration tools to easily create a repository as shown in this recipe. You should look for these administration capabilities when searching for a Subversion client for non-Windows platforms.

On Mac OS X, Versions (<http://versionsapp.com/>) and Cornerstone (<http://www.zennaware.com/cornerstone/index.php>) offer similar capabilities but are paid applications.

On Linux, the PagaVCS tool (<http://code.google.com/p/pagavcs/>) is a free clone of TortoiseSVN that includes both client and administration capabilities to create local repositories.

Manual repository creation

If your Subversion client does not offer the ability to create a local repository, then you can download the Subversion command-line tools from the official Subversion website (<http://subversion.apache.org/packages.html>) and create a repository manually following instructions found in the online Subversion reference manual (<http://svnbook.red-bean.com/>).

Other version control systems

While Subversion is easy to learn and is the system that is used by WordPress on its official plugin repository, other version control systems such as Git (<http://git-scm.com/>) and Mercurial (<http://mercurial.selenic.com/>) are gaining traction in the open source development community and could also be considered to manage your plugin code.

See also

- ▶ *Importing initial files to a local Subversion repository recipe*

Importing initial files to a local Subversion repository

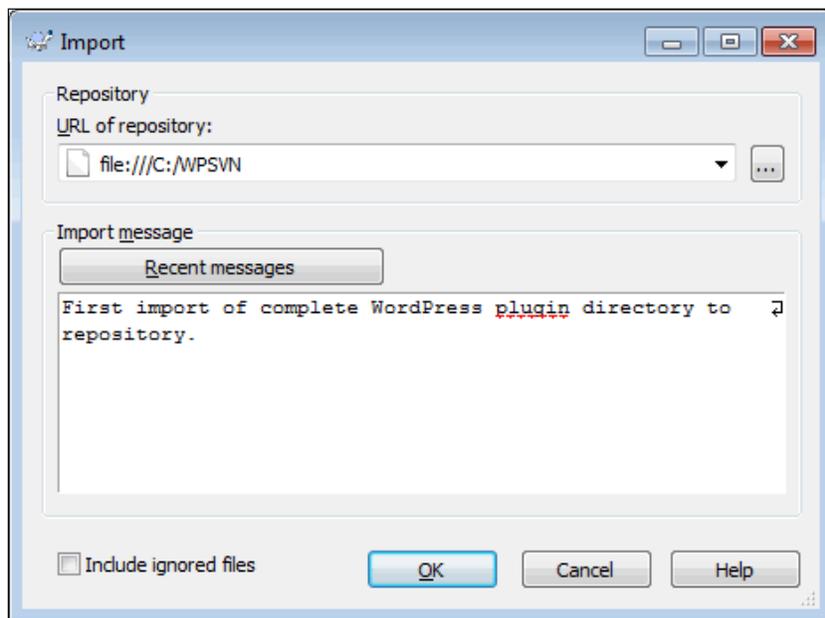
Once you have a local repository in place, this recipe describes the steps required to add files and start tracking their revisions over time. To have the flexibility to create multiple plugins as discussed throughout this cookbook without having to worry about adding each of them to the repository individually, we will add the entire WordPress plugin directory to your local repository.

Getting ready

You should have already installed a Subversion client on your computer and created a local repository as described in the *Creating a local Subversion repository* recipe. These steps will be slightly different based on the Subversion client that you have selected and your operating system.

How to do it...

1. Navigate to the `wp-content\plugins` directory of your local WordPress installation (for example, `c:\WPDev\wp-content\plugins`, if you followed the previous recipe) with the file explorer.
2. Right-click in the folder and select the **TortoiseSVN | Import** menu item.
3. Enter the file location of your local Subversion repository in the **URL of repository** field (for example, `file:///C:/WPSVN`), if it is not already specified.
4. Write a message in the **Import message** field that gives an overview of the files that are being imported in the repository.



5. Click on the **OK** button to complete the Import process.

Once the Import operation has started, TortoiseSVN sends all selected files to the repository, displaying each of their names in the process. At the end of the Import operation, it also displays the revision number that it assigned to this first set of files.

How it works...

Using the **Import** Subversion feature copies all selected files to the repository. In addition to storing the files themselves, Subversion identifies each file with a revision number and an import message. The revision number is generated by Subversion and incremented every time a group of files is added. It is especially useful when searching through a file's history.

The import message is specified by the user and is actually optional. That being said, it is important to set meaningful import messages when adding files to a repository as it will make it easier for you to identify what these files are, the state that they are in, and the reason they were added to the repository when performing future searches.

While these steps have led to a successful import, you may be wondering why nothing changed in the plugin directory. The reason is that the import process only makes copies of the selected files to the Subversion repository. An additional step, called the Checkout process, needs to take place to start keeping track of changes and file history.

See also

- ▶ *Checking out files from a Subversion repository recipe*

Checking out files from a Subversion repository

After performing an initial import of files to a Subversion repository, the files need to be checked out to really start working in a version control environment. This recipe explains how to check out files from your local repository and what the resulting file structure changes will be.

Getting ready

You should have already installed a Subversion client, created a local repository, and imported files before following this recipe. These steps will be slightly different based on the Subversion client that you have selected and the operating system you are using.

How to do it...

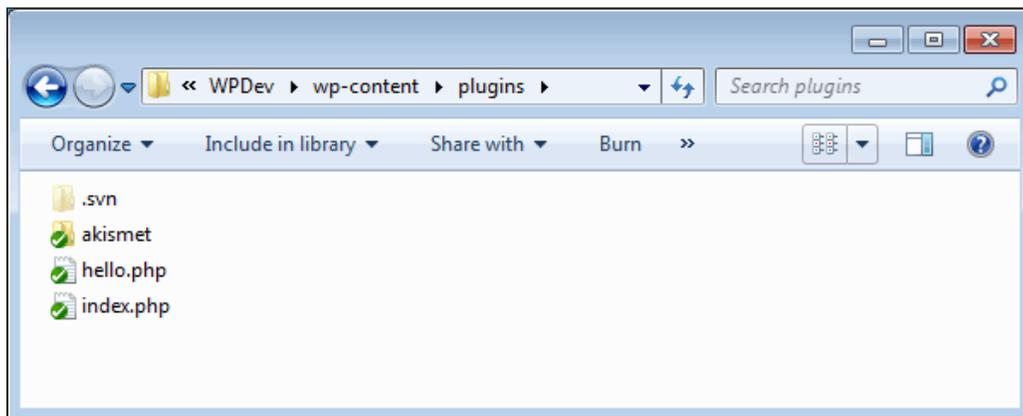
1. Navigate to the WordPress plugin directory of your local installation in the file explorer if you are not already there.
2. Right-click in the whitespace of the directory window and select the **SVN Checkout...** menu item.

3. Enter the file location of your local Subversion repository in the **URL of repository** field (for example, `file:///c:/WPSVN`), if it is not already specified.
4. Set the **Checkout directory** to the plugin folder of your local WordPress installation (for example, `C:\WPDev\wp-content\plugins`).



By default, the TortoiseSVN client adds the word `WPSVN` at the end of the path used when performing checkouts. Be sure to remove that last part of the path so that all files that are checked out go to the correct location.

5. Click on **Yes** on the dialog asking if files should be checked in a folder that is not empty. At this time, TortoiseSVN will retrieve all files that were added to the repository and copy them locally.
6. Once the operation is complete, look back at the file listing in the `plugins` directory to see that it has changed from its previous state.



How it works...

Performing a Checkout operation takes copies of all files from the repository and places them in the target directory. It also creates `.svn` directories at all levels of the file hierarchy. Looking at the default WordPress plugin directory, we can see that two `.svn` folders have been created. The first is directly in the `plugins` directory while the second is in the `akismet` subdirectory.

By default, most operating systems do not show folders that have a period at the beginning of their name since this usually identifies hidden files and directories. To display hidden folders on the Windows platform, carry out the following steps:

1. Press the *Alt* key in the file explorer to display the menu system.
2. Select the **Tools | Folder options...** menu item.
3. Select the **View** tab.
4. Set the **Hidden files and folders** radio button to **Show hidden files, folders and drives**.

The `.svn` directory contains information on the address of the repository that is associated with the files in the current folder. It also contains an original version of each file that was checked out. These original files are used for Subversion to determine when changes have been made to each file relative to their state when they were checked out or updated. While it might seem a bit redundant to have an original copy of all files in the `.svn` folders when our repository is locally hosted, this functionality allows Subversion to identify file changes when working on a remote repository, such as the official WordPress plugin server, even when your computer is not connected to the Internet.

There's more...

As you work with Subversion and TortoiseSVN, files that you create, modify, and delete will go through a number of different states. The following section explains what each of them represents.

Subversion file statuses

The green check mark indicator shown over each file icon, after performing this recipe, shows us that our files and directories have not been modified since they were last checked out or updated. These indicators will change over time as we start modifying existing files and creating new ones. The following is a list of the most common statuses that files will have as you work on a project, along with their associated TortoiseSVN icons:

- ▶ **Normal** (green check mark): The file or directory is in a normal state and has not changed since it was last checked out or updated.
- ▶ **Modified** (red exclamation mark): The file or directory has been modified since it was last checked out or updated.
- ▶ **Non-versioned** (blue question mark): The file or directory is not under version control.
- ▶ **Added** (blue plus sign): The file or directory is new and has been marked to be committed to the repository in the next commit operation.
- ▶ **Deleted** (red x icon): The directory has been deleted and will be removed from the repository in the next commit operation.

- ▶ **Ignored** (grey do not pass symbol): This file or directory will never be sent to the repository and Subversion should stop checking for changes. This state is useful to keep private files, such as personal documentation or to-do lists, in the same directory as the plugin but without uploading them to the repository and tracking their history over time.
- ▶ **Conflicted** (yellow exclamation mark): This icon appears in situations of conflict, typically when more than one person works on the same repository and multiple users made changes to the same file. While the Subversion client will normally try to merge these changes to create a single file, a conflicted state indicates that the system was not able to merge these changes automatically. Conflicted files need to be manually merged or the user needs to indicate if the file has priority over the version that is currently stored in the repository.

See also

- ▶ *Committing changes to a Subversion repository* recipe

Committing changes to a Subversion repository

During the course of a project, plugin files will typically be created, modified, or deleted. These changes should be transmitted regularly to the Subversion repository to have proper backups of all files in a project. A good practice is to commit changes at least once a day, with more frequent commit operations taking place when specific milestones are reached in the implementation of a plugin's features.

This recipe indicates how to manage file creation, modification, and deletion operations to keep everything organized and mirrored in the Subversion repository.

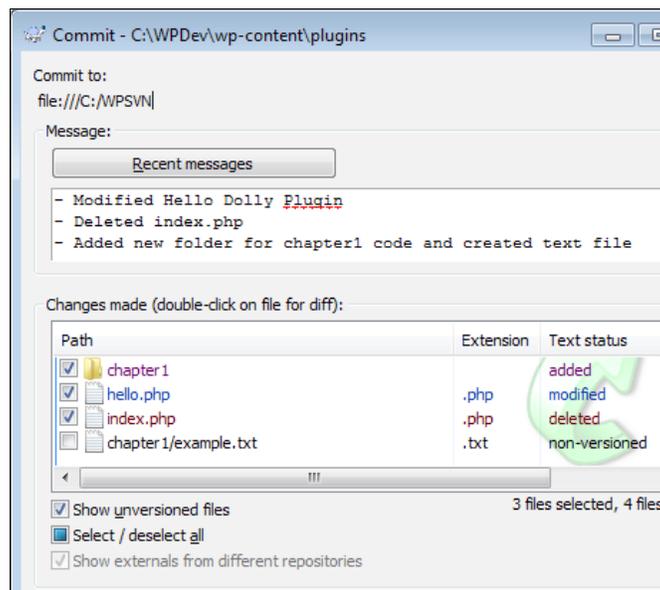
Getting ready

You should have already installed a Subversion client, created a local repository, and imported and checked out files before performing the steps in this recipe. These steps will be slightly different based on the Subversion client that you have selected and the operating system you are using.

How to do it...

1. Navigate to the WordPress plugin directory of your local installation in the file explorer if you are not already there.
2. Open the `hello.php` file in a text editor.

3. Edit the plugin name on line 7 to change it from `Plugin Name: Hello Dolly` to `Plugin Name: Goodbye Dolly`.
4. Save and close the file. You should now notice that the modified file is identified by a red exclamation mark icon in the file explorer, indicating that it has been modified.
5. Create a new folder in the `plugins` directory named `chapter1`. The new folder will be displayed, along with a blue question mark icon, indicating that it is not currently under version control.
6. Right-click on the new folder and select the **TortoiseSVN | Add...** menu item to bring up the **Add** dialog.
7. Click on the **OK** button to queue the file to be added to the repository when changes are next committed.
8. Navigate to the `chapter1` directory and create a new text file named `example.txt`.
9. Navigate back to the `plugins` directory.
10. Right-click on the `index.php` file and select the **TortoiseSVN | Delete** menu item. The selected file is immediately deleted and disappears from the file explorer.
11. Right-click in an empty part of the `plugins` directory and select the **SVN Commit...** menu item. This last step will display the **Commit** dialog, with a top section to write a message detailing the changes that are being committed, and a bottom section containing a file listing. Notice that all files but one have check marks next to them since they have either been recognized as being changed by the Subversion client or have been added or deleted through the TortoiseSVN interface. The file that does not have a check mark next to it is the text file that was created but not tagged to be included in the next commit operation using the TortoiseSVN contextual menu.



12. Type a message in the appropriate field indicating the reason for the operation.
13. Right-click on the `chapter1/example.txt` file and select the **Add** menu item to add it to the operation.
14. Click on the **OK** button to send all changes to the Subversion repository.

How it works...

Using the local data stored in the `.svn` folders, the Subversion client is able to analyze the directory contents and identify all files that are new, have been modified, or are missing since the last checkout or update operation was performed, and then generate a list of these changes.

When the commit operation is performed, new files are added to the repository, modified files are uploaded and stored next to their previous versions, while deleted files are tagged as no longer being part of the current project version. While some of these behaviors might seem strange, it's by preserving previous versions of files and even keeping files that are no longer part of a project that Subversion is able to let us navigate through a project's entire history.

While it is preferable to use the TortoiseSVN menu to mark files and directories for addition and to delete items that are no longer needed, it is also possible to perform these operations when the commit is about to take place, as we saw in the recipe steps.

There's more...

Before files are committed to the repository, many programmers and developers want to see what changes were made to the modified files, especially in an environment that promotes peer reviews before committing code changes.

Viewing the differences in modified files

By right-clicking on any modified file in the Commit dialog and selecting the **Diff** menu item, the TortoiseSVN client will display its built-in file differencing tool, highlighting the parts that are different between the last version of the files in the repository and the current version of this file. This allows users to see what changed at a glance and be sure that no code was modified inadvertently.

Updating files to latest repository version

If you are the only person committing files to a repository, and you are working on a single computer, then you will never need to use the **SVN Update** menu item. This function is designed to compare your local files with the repository and check if new files or new revisions are available in the repository that are not present locally. It will then apply all necessary changes to the local versions of these files. Remember to use the **SVN Update** option in TortoiseSVN regularly if you are working in a team environment or are developing a project across multiple systems.

See also

- ▶ *Reverting uncommitted file changes* recipe

Reverting uncommitted file changes

Until a file is committed to a repository, it is possible to revert all changes made to it since the last checkout, update, or committal of that file. This recipe shows us how to revert changes made to one or more files.

Getting ready

You should have already installed a Subversion client, created a local repository, and imported and checked out files before performing the steps in this recipe. These steps will be slightly different based on the Subversion client that you have selected and the operating system that you are using.

How to do it...

1. Navigate to the WordPress plugin directory of your local installation in the file explorer if you are not already there.
2. Open the `hello.php` file in a text editor.
3. Edit the plugin name on line 7 to change it back from `Plugin Name: Goodbye Dolly` to `Plugin Name: Hello Dolly`.
4. Save and close the file. When using the TortoiseSVN client, the file is marked as modified in the file explorer with a red exclamation mark icon.
5. To show the changes that occurred between the current version of the file and its original state, right-click on the file and select the **TortoiseSVN | Diff** menu item.
6. Close the **Diff** tool.
7. To revert the file back to its state from the last committal, right-click on the file and select the **TortoiseSVN | Revert** menu item. The **Revert** dialog will be displayed with the selected file listed and checked.
8. Click on **OK** on the **Revert** dialog to restore the file.



To revert multiple files to their previous state, right-click on an empty area of the file explorer and select the **TortoiseSVN | Revert** menu item. This will display the same dialog with a list of all files that have changed since the last commit operation. Select the files which should be reverted and click on the **OK** button to perform the reversions.

How it works...

The revert operation uses the saved copies of files that are stored in the `.svn` directories located throughout the file structure. These local files allow the Subversion client to perform a file reversion operation without needing to contact the repository, which may be remote once you start working in a team environment or publishing plugins to the official WordPress repository. The reversion operation simply discards the modified files and copies back the last saved copy to the working directory.

There's more...

While Subversion clients such as TortoiseSVN and many others offer a fairly full set of functionality to simplify file comparisons and other file manipulations, some of them also allow support for your favorite tools.

Configuring TortoiseSVN to use an external diff viewer

While TortoiseSVN offers a diff viewer, its results are not always easy to analyze. Therefore, it can be really useful to substitute it with an external tool such as WinMerge (winmerge.org). To perform this substitution, carry out the following steps:

1. Right-click in an empty area of a file explorer or the desktop and select the **TortoiseSVN | Settings** menu item.
2. Navigate to the **External Programs | Diff Viewer** section.
3. Set the top radio button to **External**.
4. Use the browse button (...) to navigate to the location of your external diff viewer and select its main program executable (for example, `C:\Program Files (x86)\WinMerge\WinMergeU.exe`).

See also

- ▶ [Viewing file history and reverting content changes to older revisions recipe](#)

Viewing file history and reverting content changes to older revisions

As multiple versions of files are committed to a repository over time, Subversion keeps track of all versions of these files along with the messages that were associated with each commit operation. On occasion, during a plugin's development, it may be necessary to go back to older versions to bring back functionality that was removed unintentionally or to restore something that stopped working after changes were made. This recipe shows you how to view a file and a project's history, see all changes that occurred between two versions of a file, and restore a version of a file older than the last commit operation.

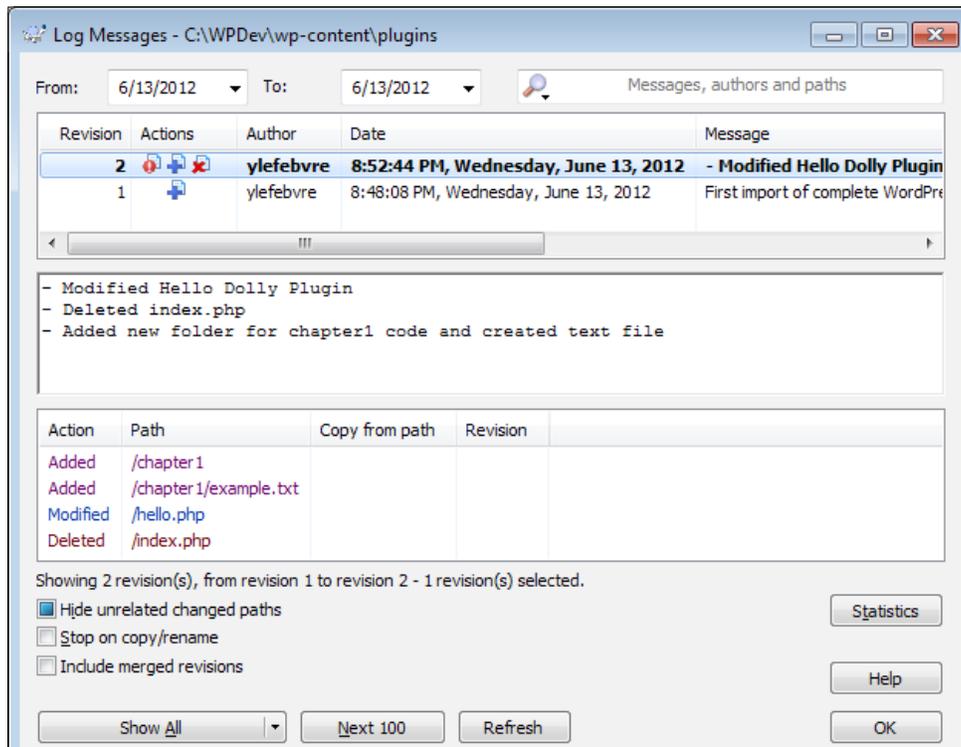
Getting ready

You should have already installed a Subversion client, created a local repository, imported and checked out files, as well as performed at least two commit operations, before performing the steps in this recipe. These steps will be slightly different based on the Subversion client that you have selected and the operating system that you are using.

How to do it...

1. Navigate to the WordPress plugin directory of your local installation in the file explorer if you are not already there.
2. Right-click on the `hello.php` file and select the **TortoiseSVN | Show log** menu item. This command displays a chronological listing of all modifications to this file, starting from its initial import to the repository.
3. Right-click on the oldest revision available for the file and select the **Compare with working copy** menu item. A diff view of the selected version of the file compared to the current working version will be displayed on-screen.
4. Right-click on the oldest revision available for the file and select the **Revert to this revision** menu item. Confirm the request by selecting the **Revert** option. The `hello.php` plugin file will revert back to the state it was in at the time of its initial import to the repository.

- Right-click in an empty area of the file explorer and select the **TortoiseSVN | Show Log** menu item. The resulting dialog shows a chronological listing of all import and commit operations, indicating the dates when each action occurred and displaying the messages that were associated to each of them. It also displays a listing of all files that were involved in each action.



- Click on each entry in the revision list to see the files that were affected and the type of operation that was performed on each of them.
- Find the revision where the `index.php` file was deleted from the plugin directory.
- Right-click on the `index.php` file and select **Revert changes from this revision** to restore the file. Select the **Revert** option to confirm the request.

How it works...

The Subversion client connects to the repository to generate a full history log for the selected file or directory. It will then traverse all information stored during all import and commit operations to produce a full listing. Since a Subversion repository keeps track of all versions of files along with the messages that were stored with each operation, it can easily show how files change during a project's history or restore a file to a previous state.

Of course, this flexibility comes at a price since all of these file versions use space on the hard drive that hosts the repository. However, this is a small price to pay for the peace of mind of having easy access to organized versions of files over the lifetime of a project.

Installing a dedicated code/text editor

Most operating systems provide a built-in text editor. While it is possible to create WordPress plugins using such a simple tool, it is highly recommended to install a dedicated code editor on your computer to simplify your plugin development work.

Of course, not all code editors are equal. Here are some of the features that you should look for when selecting a code editing application:

- ▶ PHP syntax highlighting
- ▶ Completion of PHP function names
- ▶ Ability to search in multiple files simultaneously
- ▶ Ability to highlight all instances of search keyword(s) or selected text
- ▶ Line numbering
- ▶ Ability to resize the editor text or specify a replacement font
- ▶ Possibility of opening multiple files simultaneously

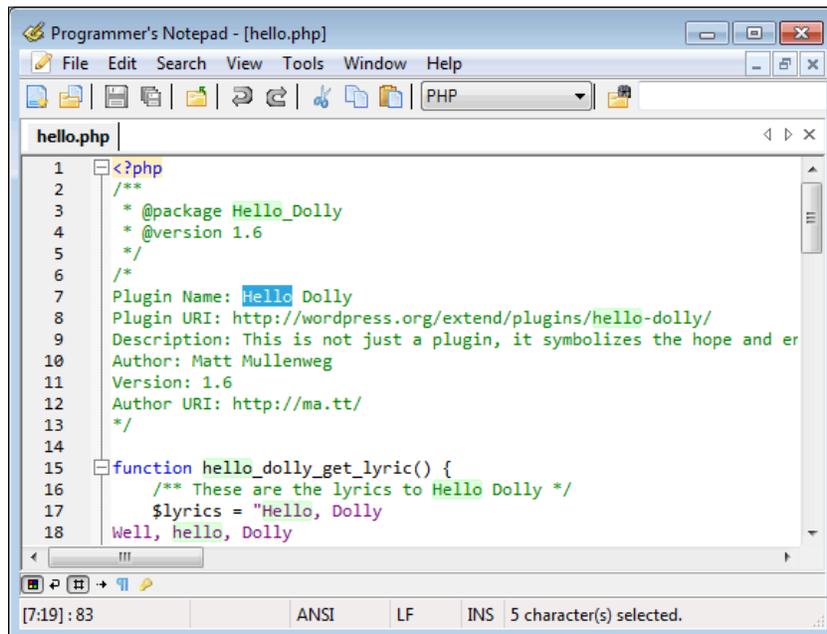
The following free editors contain most or all of these key features:

- ▶ On the Windows platform:
 - Programmer's Notepad (<http://www.pnotepad.org>)
 - Notepad++ (<http://notepad-plus-plus.org/>)
- ▶ On the Mac platform:
 - Sublime Text 2 (<http://www.sublimetext.com/2>)
 - TextMate (<http://macromates.com>)
 - TextWrangler (<http://www.barebones.com/products/TextWrangler>)
- ▶ On the Linux platform:
 - Screem (<http://www.screem.org/>)

This recipe explains how to install a dedicated code editor and shows basic editor operations. It provides detailed steps using Programmer's Notepad for Windows.

How to do it...

1. Download the installation package for one of the text editors listed previously.
2. Run the installation program for the editor and select the default settings.
3. Launch the text editor.
4. Open the `hello.php` file from the `plugin` directory of your local WordPress installation. You will see that different parts of the code are displayed in different colors based on the type of code.
5. Double-click on a word to select it. You will see any other instance of that same word highlighted across the file contents.
6. Select the **View | Line Numbers** menu item (or similarly named item based on your selected text editor) to display line numbers in the editor.



```
1 <?php
2 /**
3  * @package Hello_Dolly
4  * @version 1.6
5  */
6 /**
7  Plugin Name: Hello Dolly
8  Plugin URI: http://wordpress.org/extend/plugins/hello-dolly/
9  Description: This is not just a plugin, it symbolizes the hope and er
10 Author: Matt Mullenweg
11 Version: 1.6
12 Author URI: http://ma.tt/
13 */
14
15 function hello_dolly_get_lyric() {
16     /** These are the lyrics to Hello Dolly */
17     $lyrics = "Hello, Dolly
18     Well, hello, Dolly
```

How it works...

Code editors have built-in parsers that enable them to identify the parts of the code that are comments, PHP language functions, text strings, and a variety of other elements. Having these elements colored on-screen makes it much easier to read through code and to see that a function's name is not spelled correctly, or to quickly identify comments.

Another functionality that is crucial when developing plugins for WordPress is the ability to see line numbers in the editor. This function comes in handy especially when PHP code errors come up, since the filename and line of code that was being processed at the time of the error are normally displayed. In most code editors, the developer can either scroll to the specific line or enter the line number in a quick **Go To** dialog box to jump to that line right away.

Installing and configuring the NetBeans Integrated Development Environment

If you have enjoyed moving to a dedicated code editor in the previous recipe but want a solution that provides even more integration to perform all of your WordPress plugin development tasks in a single place, an **integrated development environment (IDE)** such as the free NetBeans platform will be the perfect solution for you. In addition to having all of the core features of a code editor, NetBeans has the ability to constantly parse your code to identify syntax errors and highlight changes made since your last commit or update operation, right in the editor.

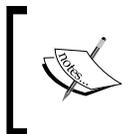
It also features built-in Subversion and MySQL clients to be able to commit code changes and manage database records straight from its interface. The NetBeans application is cross-platform, available on the Windows, Mac, and Linux operating systems, and can be quickly configured to work with a local WordPress installation. This recipe explains how to perform these installation and configuration tasks.

Getting ready

You should already have set up a local WordPress installation on your computer.

How to do it...

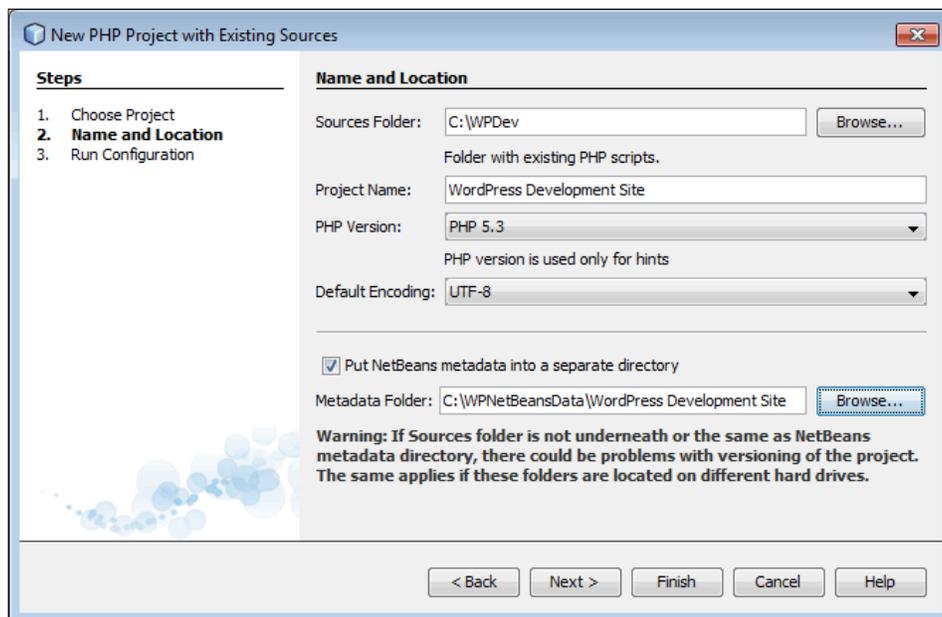
1. Download the PHP NetBeans installer for your choice of platform from the NetBeans website (<http://netbeans.org/downloads/index.html>).



While you can choose a version of NetBeans that will support a large variety of programming languages (Java, C/C++, and so on), the PHP version of NetBeans contains all the necessary elements to develop plugins for WordPress.

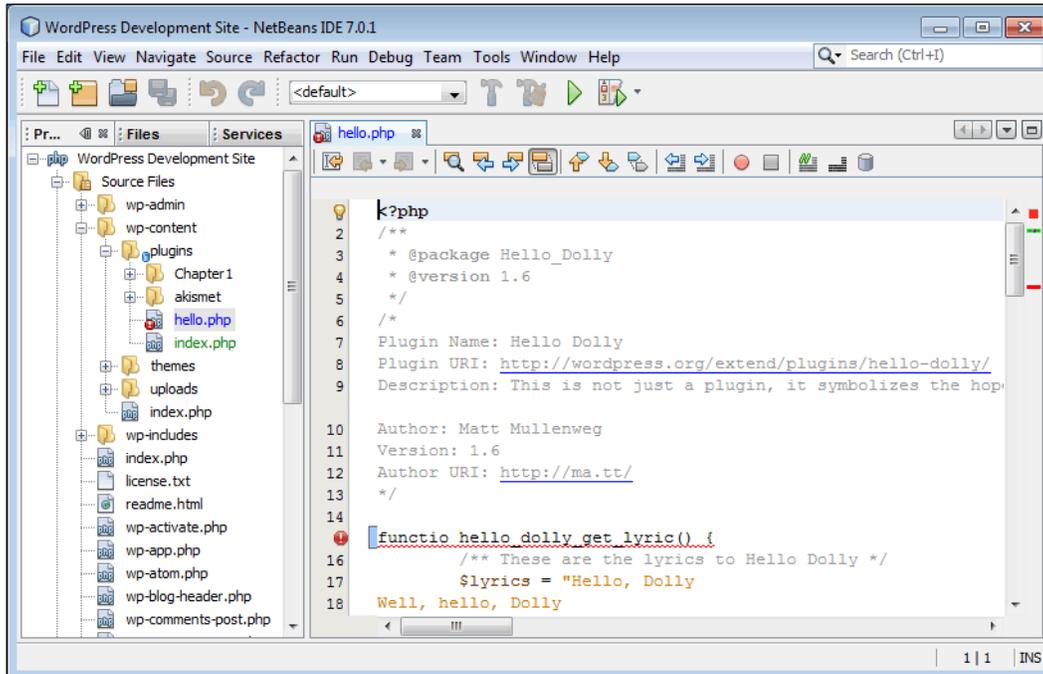
2. Install the NetBeans tool by running the installer, selecting all default options, and accepting the license agreement.
3. Launch the **NetBeans IDE** using the shortcut it created during the installation.

4. Install the latest updates, if applicable, and restart the IDE to run the latest version.
5. Select the **File | New Project** menu item.
6. Select **PHP** in the **Categories** section and **PHP Application with Existing Sources** in the **Projects** section.
7. Click on **Next**.
8. Set the **Sources Folder** to the location of your local WordPress installation (c:\WPDev on Windows, if you followed the previous recipe).
9. Specify a **Project Name** (for example, WordPress Development Site).
10. Select **PHP 5.3** in the **PHP Version** field.
11. Check the **Put NetBeans metadata into a separate directory** option and create a new folder to hold this data (for example, c:\WPNetBeansData).



12. Click on **Next**.
13. Set the **Project URL** to `http://localhost/`.
14. Click on **Finish**.
15. Once the project is loaded, close the **Tasks** panel since it will be populated with a long list of to-do tasks that are extracted from the WordPress source code.
16. Using the **Projects** view, navigate to the `wp-content/plugins` directory of your WordPress installation and double-click on the `hello.php` file to see it in the NetBeans editor.

17. Search for the keyword `function` in the file and remove its last letter `n` to see a red exclamation mark displayed in the left margin of the code editor. This indicates that a PHP syntax error was detected.



18. Undo this last change.
19. Press the `F6` key to launch a web browser session pointing to your local development site.

How it works...

The NetBeans editor works by creating a project that points to your website's directory structure and loading all files that are found in that location. With its integrated project browser, it is very easy to find and edit multiple plugin files by starting a single tool. NetBeans combines all of the functionality of a dedicated code editor, a Subversion client such as TortoiseSVN, and the phpMyAdmin database administration interface to make it possible to perform all tasks related to WordPress plugin development in a single environment.

See also

- ▶ *Interacting with a Subversion repository from the NetBeans interface* recipe

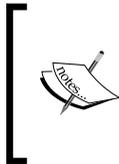
Interacting with a Subversion repository from the NetBeans interface

One of the multiple benefits of using the NetBeans IDE is that it is pre-integrated with the Subversion version control system. Whether you're developing plugins for your own private use, for customers, or for public distribution on `wordpress.org`, Subversion is a great system to use to keep track of all important revisions of your work over time.

This recipe explains how to use the built-in Subversion functionality in NetBeans to interact with a file repository.

Getting ready

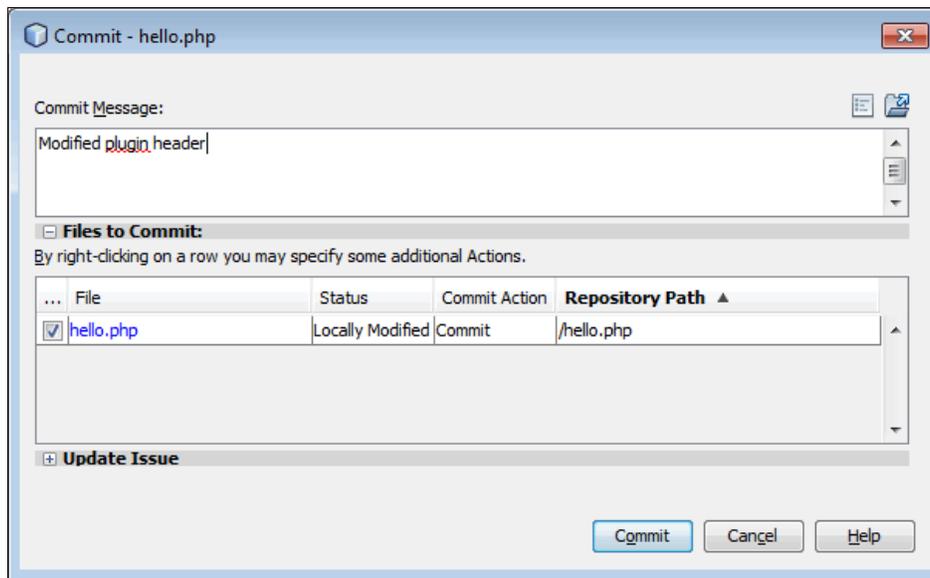
You should have already installed the NetBeans IDE and created a project pointing to your local WordPress development site. You should also have created a Subversion repository on your system and imported the contents of the WordPress plugin directory.



At the time of writing, an incompatibility between NetBeans 7.1 and repositories created by TortoiseSVN 1.7.x requires additional steps to be executed before performing this recipe. These steps can be found on the Netbeans website at http://netbeans.org/projects/versioncontrol/pages/Subversion1_7.

How to do it...

1. Using the Projects view in NetBeans, navigate to the `wp-content/plugins` directory of your WordPress installation and double-click on the `hello.php` file to display it in the code editor.
2. Change any line of code in the plugin header (top section of the plugin information on the plugin name, author, and so on). Notice that colored bands start appearing on the left margin of the code editor as lines are modified, added, or deleted.
3. Position the mouse cursor over the colored area that is displayed next to the modified line to see a tooltip indicating that the line has been modified.
4. Click on the colored notification area to see the previous content of that line and have the opportunity to roll back the modified content to its last known state from a previous insert, commit, or update operation.
5. In the project window, right-click on the `hello.php` file and select the **Subversion | Commit** menu. This command displays the **NetBeans Subversion Commit** interface. While it is slightly different from the equivalent **TortoiseSVN** dialog, you should still recognize the field used to specify a commit message and the list of all files that were identified as having changed from the last insert, commit, or update operation.



6. Enter a message in the **Commit Message** field.
7. Click on the **Commit** button to send your changes to the Subversion repository.



By right-clicking on any file from the plugin directory and selecting the Subversion menu, we can see that all the functionalities we explored in the previous Subversion-related recipes are available in the NetBeans environment. However, some of them have different names. For example, the **Show log** menu item is called **Search History** and has some more advanced features than the TortoiseSVN client.

How it works...

Similar to the way the TortoiseSVN client works, the NetBeans interface has been built using the Subversion client libraries to provide us with a full-featured tool that can access any Subversion repository. Since our WordPress plugin directory files were already imported and checked out from a repository, NetBeans is able to read the repository information that is contained in the `.svn` directories located across the project structure and use this data to identify code changes on the fly during code editing. It also has access to information on the repository address that is associated to the plugin files to send new and updated items to the correct location without asking us to specify where they should be sent.

Managing a MySQL database server from the NetBeans interface

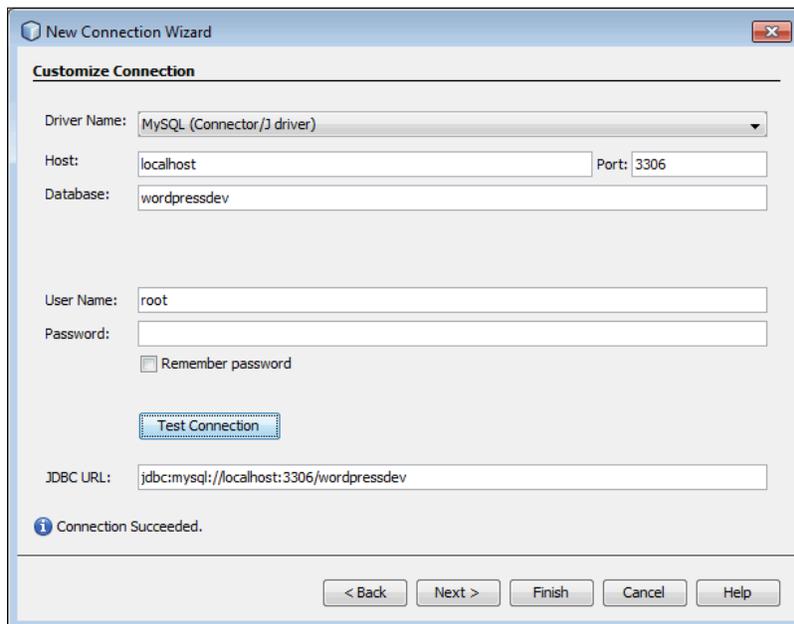
With its built-in MySQL interface, the NetBeans IDE allows us to perform database queries, data updates, and table structure modifications from the same interface that we will use to write our WordPress plugins. This recipe will show you how to get NetBeans to connect to your local MySQL database server and access data from some of the WordPress data tables.

Getting ready

You should have already installed a local web server on your computer, completed a local WordPress installation, set up the NetBeans IDE, and created a project pointing to your local WordPress development site.

How to do it...

1. Click on the **Services** tab on the left side of the NetBeans interface.
2. Expand the **Databases** item in the navigation tree.
3. Expand the **Drivers** item.
4. Right-click on the **MySQL (Connector/J driver)** item and select the **Connect Using...** menu item.



5. Leave the **Host** field with its default value (`localhost`).
6. Change the **Database** name to `wordpressdev`.
7. Leave all other fields as they are.
8. Click on the **Test Connection** button. This will try to connect to the MySQL database server and will display a message indicating if the connection was successful.
9. Click on the **Finish** button to complete the connection setup.
10. Expand the new **jdbc:mysql://localhost:3306/wordpressdev [root on Default schema]** item that appeared below **Drivers**.
11. Expand the **wordpressdev** item.
12. Expand the **Tables** item.
13. Expand the **wpdev_posts** table element to see a list of all fields in the WordPress posts table.
14. Right-click on the **wpdev_posts** table and select the **View Data...** menu item to show all data records contained in this table. As you can see, posts and pages are actually stored in the same database table. In addition to displaying the requested data, NetBeans also shows the SQL query that it performed to retrieve the data. This can be useful to learn the basics of this query language.

How it works...

The NetBeans IDE has been compiled with the necessary client libraries to connect to a MySQL database. By configuring NetBeans to access our database, it is able to get information on the complete information structure and present us with an interface that is much more dynamic than the phpMyAdmin web-based interface that was shown in a previous recipe.

Being aware of the structure of a WordPress database and learning the syntax of SQL commands will help you greatly as we start developing plugins in the later recipes.

2

Plugin Framework Basics

In this chapter we will cover the following topics:

- ▶ Creating a plugin file and header
- ▶ Adding output content to page headers using plugin actions
- ▶ Using WordPress path utility functions to load external files and images
- ▶ Modifying the page title using plugin filters
- ▶ Adding text after each item's content using plugin filters
- ▶ Inserting link statistics tracking code in page body using plugin filters
- ▶ Troubleshooting coding errors and printing variable content
- ▶ Creating a new simple shortcode
- ▶ Creating a new shortcode with parameters
- ▶ Creating a new enclosing shortcode
- ▶ Loading a stylesheet to format plugin output
- ▶ Writing plugins using object-oriented PHP

Introduction

From its very first versions, WordPress was always designed as a very open platform. This openness was exemplified not only through its open source licensing and distribution model but also with its open plugin architecture, providing developers with the ability to deliver an even richer experience to its users.

While a basic WordPress installation provides a great amount of functionality that continues to expand from one release to the next, users often have a need to add one more feature to make it the perfect website management system. This is where the plugins come into play. They can fill this gap by augmenting or manipulating virtually any aspect of a WordPress website's display and administrative tasks.

Just like WordPress, plugins are written in the PHP programming language, which is structurally similar to more traditional languages such as C and C++. This code is stored in plain ASCII text files that are read and executed on the web server when pages are requested to be displayed. The secret ingredient that enables plugins to have such great power in WordPress is the inclusion of callback mechanisms called **hooks** throughout the application's source code. These hooks come in two flavors, called action and filter hooks, which allow plugins to add content to a site and modify data before it is displayed, respectively. Whether it's rendering a site's front page, a single article, or its administration pages, WordPress has hundreds of entry points where custom functions can be executed.

Beyond their ability to augment the WordPress functionality, a side benefit of plugins is that most functionality they add to a site is independent of the active theme. Therefore, users who like to change their theme frequently don't have to worry about manually adding back custom elements to their new themes when they make a switch.

This chapter explains the difference between action and filter hooks and shows how to use them to write a first set of plugins that will range in functionality from adding information to the page header to defining new custom shortcodes.

Creating a plugin file and header

The first step of creating a WordPress plugin is to create a PHP file and add the necessary information to have it recognized by the web-publishing platform. This first recipe shows you how to create a basic plugin file for WordPress and how to see and activate this new extension from the administration interface.

Getting ready

You should have access to a WordPress development environment, either on your local computer or a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-plugin-header`.
3. Navigate to this directory and create a new text file called `ch2-plugin-header.php`.
4. Open the new file in a text editor and add the following text:

```
<?php
/*
Plugin Name:      Chapter 2 - Plugin Header
Plugin URI:
Description:     Declares a plugin that will be visible in the
WordPress admin interface
Version:         1.0
Author:          Yannick Lefebvre
Author URI:      http://ylefebvre.ca
License:         GPLv2
*/
?>
```



While the `Description` text is shown on two separate lines in the code example, it should all be entered on a single line in your code to be completely displayed in the WordPress **Installed Plugins** list.

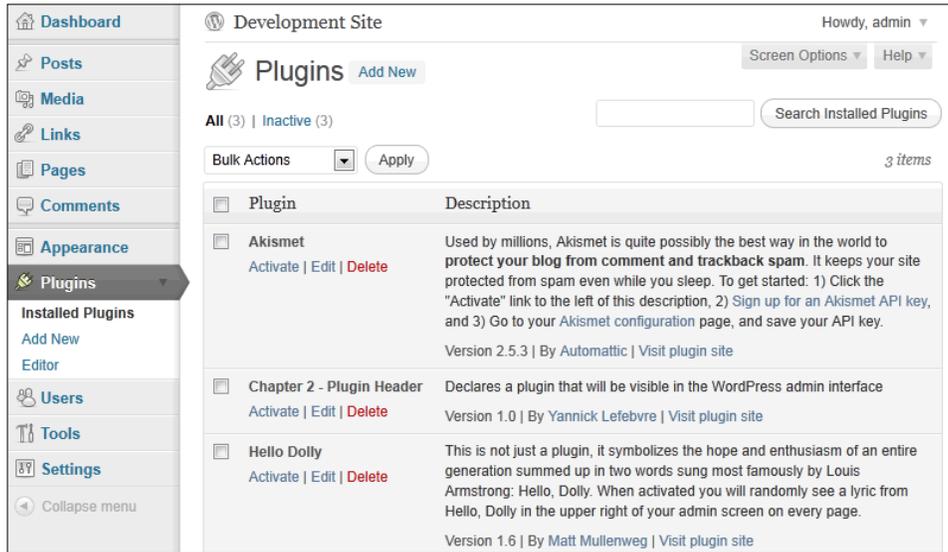


Downloading the example code

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

5. Save and close the new file.
6. Log in to the administration page of your development WordPress installation.

- Click on **Plugins** in the left-hand navigation menu to show a list of all installed plugins. You should see your new plugin listed next to the two default ones that come pre-packaged with WordPress.



- Enable the plugin by clicking on the **Activate** link under its name. You will see that the background color of your new plugin changes to indicate that it has been activated along with a message specifying that the activation was successful.

How it works...

Plugin files can either be located directly in the `wp-content/plugins` directory or in a subdirectory under this location. When you access the **Installed Plugins** list in the administration interface, WordPress scans all potential plugin locations looking for PHP files that contain comments following the format specified in this recipe. There can actually be one or more PHP files containing plugin header data in any of these directories and each of them will show up as an entry in the plugin list.

Taking a closer look at the code that we entered in the file, the first and last line of the plugin file are tags that identify the beginning and end of the PHP code that will be analyzed and executed by the PHP interpreter.



To ensure compatibility with most WordPress installations, it is important to use complete `<?php` open tag syntax in your plugin code instead of the `<?` short-hand version, since not all PHP installations are configured to support the short version and many users don't have access to change this type of configuration on their server.

The second and second-to-last lines indicate that the enclosed text should be considered as text comments. Finally, each line within the comment contains a specific label indicating the type of information that follows it. When this information is found, WordPress retrieves data about the plugin and adds it to the list.

When a plugin is activated, WordPress validates the file's content to be sure that it is valid PHP code. It will then execute this content every time any page is rendered on the site, whether that page is front-facing or a backend administration section. For this reason, it is preferable to activate plugins only when they are in use, to avoid site slowdowns.

Of course, at this point, our new plugin does not add or modify any functionality in our WordPress installation since it does not contain real code, but this is still an important first step.

There's more...

The creation of a plugin file and header can actually be automated if you opted to use the NetBeans IDE, as discussed in an earlier recipe.

Installing the WordPress plugin creation module in NetBeans

Similar to WordPress, NetBeans has its own plugin architecture that allows developers to add functionality to the IDE. To automate the creation of WordPress plugin files, you can download and install a small module that will quickly create these files:

1. Navigate to the following blog entry using your favorite web browser (the typo in the address is accurate to get you to the blog entry): http://blogs.oracle.com/netbeansphp/entry/my_fitst_wordpress_plugin_in.
2. Right-click on the link to download the module and save the `org-netbeans-modules-php-wordpress.nbm` file to your computer.
3. Start **NetBeans IDE**.
4. Select the **Tools | Plugins** menu item.
5. Select the **Downloaded** tab. Click on the **Add Plugins...** button.
6. Navigate to the location of the saved NetBeans module, select the file, and click on the **Open** button.

7. Back in the **Plugins** dialog, verify that the plugin is selected and click on the **Install** button located in the bottom-left corner. Click on the **Next** button, accept the terms of the license agreements, and click on **Install** to fully register the plugin with NetBeans. Click on **Continue** to accept installing the plugin even if it is not signed. Click on **Finish** once the installation is complete.
8. **Close** the **Plugins** dialog.
9. Select the **File | New File...** menu item to create a new file. You will see a new entry in the **File Types** section of the file creation dialog named **WordPress Plugin**.
10. Select the new entry and click on the **Next** button.
11. Set the **File Name** to `new-netbeans-plugin.php`. Click on **Finish** to create a new plugin file with a valid WordPress header.

See also

- ▶ *Installing a web server on your computer recipe in Chapter 1, Preparing a Local Development Environment*
- ▶ *Downloading and configuring a local WordPress installation recipe in Chapter 1, Preparing a Local Development Environment*
- ▶ *Installing and configuring the NetBeans Integrated Code Development Environment recipe in Chapter 1, Preparing a Local Development Environment*

Adding output content to page headers using plugin actions

A common action performed by plugins is to add extra content to the header of visitor-facing pages generated by WordPress. This recipe shows you how to register an action **hook** function to be able to add such additional content. To make this example more concrete, we will use the Google Analytics page header JavaScript code that so many people use to get good page view statistics for their site.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-page-header-output`.
3. Navigate to this directory and create a new text file called `ch2-page-header-output.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Page Header Output`.

5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code to register a function that will be called when WordPress renders the page header:

```
add_action( 'wp_head', 'ch2pho_page_header_output' );
```

7. Add the following code section to provide an implementation for the `ch2pho_page_header_output` function:

```
function ch2pho_page_header_output() { ?>

    <script type="text/javascript">

        var gaJsHost = ( ( "https:" == document.location.protocol ) ?
            "https://ssl." : "http://www." );

        document.write( unescape( "%3Cscript src='" + gaJsHost +
            "google-analytics.com/ga.js' \n\
            type='text/javascript'%3E%3C/script%3E" ) );

    </script>

    <script type="text/javascript">

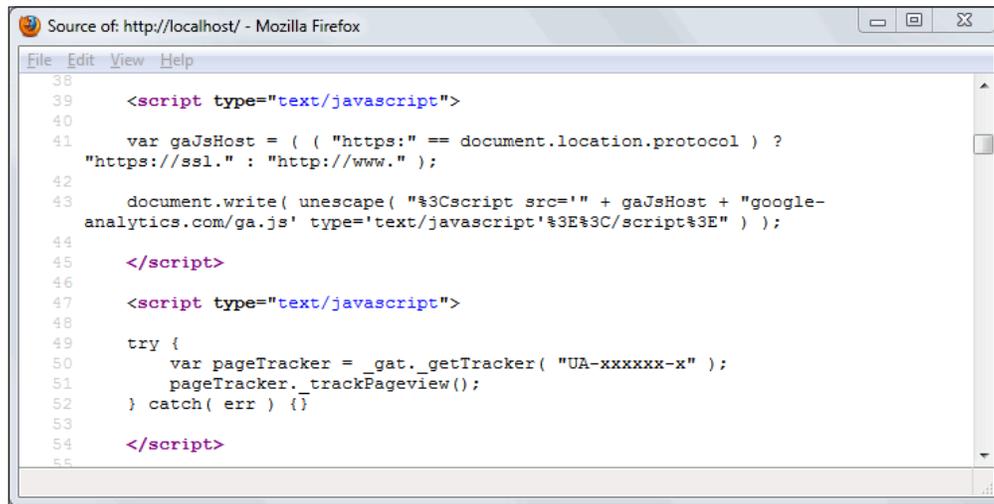
        try {
            var pageTracker = _gat._getTracker( "UA-xxxxxxx-x" );
            pageTracker._trackPageview();
        } catch( err ) {}

    </script>

    <?php }
```

8. Save and close the plugin file.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.

12. Navigate to your website's front page and use your browser's **View Page Source** function to see the HTML source code for the site. The exact name of this function will be slightly different based on which browser you are using. Reading through the page source code, all of the code contained between the two curled brackets of our new function will be visible on your website's header.



```
38
39 <script type="text/javascript">
40
41   var gaJsHost = ( ( "https:" == document.location.protocol ) ?
42   "https://ssl." : "http://www." );
43
44   document.write( unescape( "%3Cscript src='" + gaJsHost + "google-
45   analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E" ) );
46
47 </script>
48
49 <script type="text/javascript">
50
51   try {
52     var pageTracker = _gat._getTracker( "UA-xxxxxx-x" );
53     pageTracker._trackPageview();
54   } catch( err ) {}
55
56 </script>
```

How it works...

The `add_action` function is used to associate custom plugin code to one of the two types of WordPress hooks, the **action hook**. As mentioned briefly in this chapter's introduction, hooks are the enabling functionality that make plugins possible in WordPress. Action hooks enable the execution of additional code at specific points when either public-facing or administration pages are prepared to be displayed. This code usually adds content to a site or changes the way a given action is performed.

In this recipe, the first line of code that we wrote registered a function named `ch2pho_page_header_output` with an action hook called `wp_head`. This action is one among more than 500 action hooks that are available in current versions of WordPress and it allows any registered function to output additional content to the page header. Since all echoed content will be displayed, we can write our callback function very simply by placing `?>` and `<?php` tags around the Google Analytics code. This will tell PHP to display all content that is within that function's body as opposed to interpreting it.

As you may have noticed, the current code is not very flexible since you would need to hardcode your Google Analytics account number in the output for it to function properly. The creation of a configuration panel in *Chapter 3, User Settings and Administration Pages* will provide a way to configure such information to make our plugins more flexible.

Now, to fully understand its syntax, let's take a closer look at the complete `add_action` function:

```
add_action ( 'hook_name', 'your_function_name', [priority],  
            [accepted_args] );
```

The first parameter, the hook name, indicates the name of the WordPress hook that we want our custom function to be associated with. This name must be accurately spelled; otherwise our function will not be called and no error message will be displayed.

The second parameter is the name of the plugin function that will be called to perform an action. This function can have any name, with the only condition being that this name must be unique enough to avoid conflicting with functions from other plugins or from the WordPress code. In this recipe, the function name starts with an acronym representing the name of the plugin, making it much more unique.

The priority parameter is optional, as indicated by the square brackets, and has a default value of 10. It indicates the execution priority of this plugin relative to other plugin functions that hook into the same action, with a lower number indicating a higher priority.

Any plugin can register one or more functions with an action hook using the `add_action` function. As it is rendering web pages, WordPress keeps a queue of all entries and calls them at the appropriate moment. It is interesting to note that the hook mechanism is also used by WordPress itself as it regularly calls the `add_action` function in its own code to register functions to be called at the right time. If you realize that you need your function to be called before or after other plugins that are registering with the same hook, change the value of the priority parameter.

The last parameter of the `add_action` function, `accepted_args`, has a default value of 1 and should always be set to a number. It should also only be set to a different value for some particular hooks where more than one parameter should be passed to the registered function. Some of these hooks will be covered in later recipes.

There's more...

Finding the right hooks to register plugin functions is a large part of WordPress plugin development. Fortunately, there are a number of ways to get information on existing hooks and learn when they get called during the WordPress page generation process.

Action hooks online listings

The WordPress Codex is a Wiki-powered documentation site that contains a multitude of information that is useful to users and developers alike. When it comes to action hooks, the Codex contains information on the most commonly-used hooks, with basic descriptions indicating how they can be used available at http://codex.wordpress.org/Plugin_API/Action_Reference. That being said, this is not a complete listing.

There are many third-party sites that parse the WordPress source code and provide their own hook listings (for example, http://adambrown.info/p/wp_hooks/hook/actions). While hooks are not as eloquently documented in these types of raw listings, they do provide basic information on their names and where they are called as WordPress generates pages for visitors and administrators. These details can be enough to find a hook based on the functionality that you are trying to implement.

Searching for hooks in the WordPress source code

Since WordPress is open source, another way to find information about hooks is to search directly within its code. For every action hook that accepts user functions, you will see a call to the `do_action` function to execute all registered items. As can be seen, the function takes two or more arguments, with the second one(s) being optional:

```
do_action ( 'tag', [$arg] );
```

For the example shown in this recipe, a search for `do_action('wp_head')` reveals that it is the only function that is called when a theme makes a call to the `wp_head()` function in its header file:

```
do_action('wp_head');
```

See also

- ▶ *Creating a plugin file and header recipe*

Using WordPress path utility functions to load external files and images

On occasion, plugins need to refer to external files (for example, images, JavaScript, or jQuery script files) that are stored in the plugin directory. Since users are free to rename a plugin's folder or even install plugin files straight into the WordPress plugin directory, paths to any external files must be built dynamically based on the actual plugin location. Thankfully, a number of utility functions are present to simplify this task. In this recipe, we will write a simple plugin that will add a **favicon** meta tag to a website's header, pointing to an image file located in the plugin directory.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-favicon`.
3. Use a web service such as <http://getfavicon.org> to retrieve a website's **favicon** (for example, www.packtpub.com) and store it in the plugin directory with the name `favicon.png`.

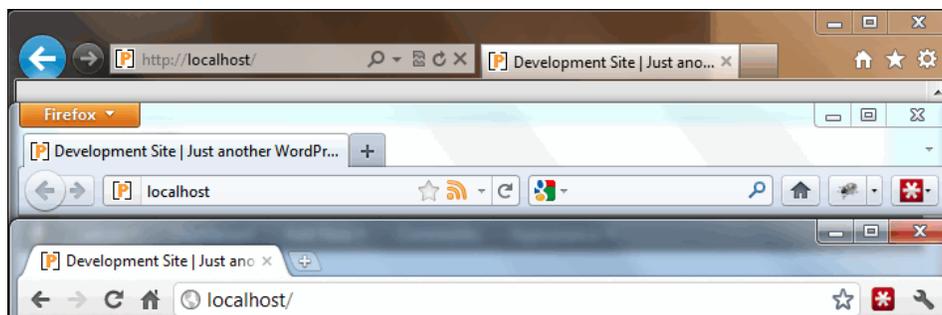
- Convert the PNG file to an ICO file using a web service such as <http://favicon-generator.org/> to work with a larger number of web browsers and make sure that the resulting file is called `favicon.ico`.
- Navigate to the plugin directory and create a new text file called `ch2-favicon.php`.
- Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Favicon`.
- Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the remaining PHP code.
- Add the following line of code to register a function that will be called when WordPress renders the page header:

```
add_action( 'wp_head', 'ch2fi_page_header_output' );
```

- Add the following code section to provide an implementation for the `ch2fi_page_header_output` function:

```
function ch2fi_page_header_output() {  
    $icon_url = plugins_url( 'favicon.ico', __FILE__ );  
  
    ?>  
  
    <link rel="shortcut icon" href="<?php echo $icon_url; ?>" />  
  
    <?php }  
}
```

- Save and close the plugin file.
- Log in to the administration page of your development WordPress installation.
- Click on **Plugins** in the left-hand navigation menu.
- Activate your new plugin.
- Navigate to your website's front page and refresh it to see that the icon file that you assigned through your plugin code now appears in your browser's address bar, title bar, or navigation tab, depending on your preferred browser. The following screenshot shows how the favicon file is rendered in Internet Explorer, Mozilla Firefox, and Google Chrome, from top to bottom:



How it works...

The `plugins_url` utility function, used in conjunction with the `__FILE__` PHP constant and the name of our favicon file, enables us to quickly get the URL of the directory where our plugin files are located and print out the appropriate HTML command to notify browsers of the location of this file:

```
plugins_url( $path, $plugin );
```

The `plugins_url` function can be called with or without parameters. In the first case, it builds a URL by appending the path found in the first parameter to the location of the file specified in the second argument. In the second situation, it simply returns the location of the plugin directory.

There's more...

The `plugins_url` function is one of the many functions that can be used in plugins to help find the location of files in a WordPress installation. Other useful functions include:

- ▶ `get_theme_root()`: Returns the address of the theme installation directory
- ▶ `get_template_directory_uri()`: Retrieves the URI to the current theme's files
- ▶ `admin_url()`: Provides the address of the WordPress administrative pages
- ▶ `content_url()`: Indicates where the `wp-content` directory can be found
- ▶ `site_url()` and `home_url()`: Return the site address
- ▶ `includes_url()`: Provides the location of WordPress include files
- ▶ `wp_upload_dir()`: Indicates the directory where user-uploaded files are stored

See also

- ▶ *Creating a plugin file and header recipe*
- ▶ *Adding output content to page headers using plugin actions recipe*

Modifying the page title using plugin filters

Beyond adding functionality or content to a site, the other major task commonly performed by plugins is to augment, modify, or reduce information before it is displayed on-screen. This is done by using WordPress filter hooks, which allow plugins to register a custom function through the WordPress API, to be called since content is prepared, before it is sent to the browser. In this recipe, we will learn how to implement our first filter callback function to add text to the page title, indicating the type of content that is currently displayed.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-title-filter`.
3. Navigate to this directory and create a new text file called `ch2-title-filter.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Title Filter`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section, to create space to add the PHP code.
6. Add the following line of code to register a function that will be called when WordPress is preparing data to output the page title as part of the page header:

```
add_filter( 'wp_title', 'ch2tf_title_filter' );
```

7. Add the following code section to provide an implementation for the `ch2tf_title_filter` function:

```
function ch2tf_title_filter ( $title ) {  
  
    //Select new title based on item type  
    if ( is_front_page() )  
        $new_title = 'Front Page >> ';  
    elseif ( get_post_type() == 'page' )  
        $new_title = 'Page >> ';  
    elseif ( get_post_type() == 'post' )  
        $new_title = 'Post >> ';  
  
    // Append previous title to title prefix  
    if ( isset( $new_title ) ) {  
        $new_title .= $title;  
        // Return new complete title to be displayed  
        return $new_title;  
    } else {  
        return $title;  
    }  
}
```

8. Save and close the plugin file.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.

12. Use a web browser to visit your website. As you visit pages and single posts, you will see that the browser's title bar or navigation tabs display additional text before their name.



How it works...

The `add_filter` function is used to associate a custom plugin function to the second type of WordPress hooks, the **filter hook**. Filter hooks give plugins the chance to augment, modify, delete, or completely replace information while WordPress is executed. To enable this, filter functions are sent data that can be modified as a function parameter. They must return the resulting set of data back to WordPress once they have finished making the changes.

Unlike action hooks, filter functions must not output any text or HTML code since they are executed while output is being prepared and that will likely result in the output showing up in unexpected places in the site layout. Instead, they should return the filtered data.

Taking a closer look at the parameters of the `add_filter` function, we can see that it is very similar to the `add_action` function that we saw in the previous recipes:

```
add_filter( 'hook_name', 'your_function_name', [priority],  
           [accepted_args] );
```

The first parameter, the hook name, indicates the name of the WordPress hook that we want our custom function to be associated with. This name must be accurately spelled; otherwise our function will not be called and no error message will be displayed.

The second parameter is the name of the plugin function that will be called to filter data. This function can have any name, with the only condition being that this name must be unique enough to avoid conflicting with functions from other plugins or from the WordPress code.

The `priority` parameter is optional, as indicated by the square brackets, and has a default value of 10. It indicates the execution priority of this plugin relative to other plugins that are loaded by WordPress, with a lower number indicating a higher priority.

The last parameter of the function, `accepted_args`, has a default value of 1 and indicates how many parameters will be sent to your custom filter function. It should only be set to higher values when you are using filters that will send multiple parameters, as will be shown in the later recipes.

There's more...

Beyond demonstrating how to change a page's title, this plugin also shows how to use some of WordPress' conditional and query functions. We also take a look at resources to learn more about filter hooks.

is_front_page function

The `is_front_page()` function is very useful when you are looking at implementing functionality that will only be displayed if the WordPress site is displaying its front page. It will return a simple `true` or `false` Boolean value that can be used in a logical comparison test.

get_post_type function

Between posts, pages, and custom post types, WordPress can display a multitude of information on a site. To create plugins that will adapt to all of these types of content, the `get_post_type()` function can come in quite handy. In this recipe, we checked what type of content was being displayed before making changes to the incoming title data.

Filter hooks online listings and the apply_filters function

Similar to action hooks, information about commonly-used filter hooks can be found on the WordPress Codex (http://codex.wordpress.org/Plugin_API/Action_Reference) or on sites that provide raw function lists (for example, http://adambrown.info/p/wp_hooks/hook/filters).

It is also possible to learn about filter hooks by searching for occurrences of the `apply_filters` function in the WordPress code. As can be seen in the following code, this function has a variable number of arguments, with the first one being the name of the filter hook, the second representing the value that the registered function will be able to modify, and the remaining optional parameters containing additional data that may be useful in the implementation of the filter function:

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

For the example shown in this recipe, a search for `apply_filters('wp_title'` in the WordPress code reveals that it is called within the `wp_title` template function and actually sends two additional parameters, in addition to the title to be modified:

```
$title = apply_filters('wp_title', $title, $sep, $seplocation);
```

See also

- ▶ *Creating a plugin file and header recipe*

Adding text after each item's content using plugin filters

After making a number of changes to the page header, title, and favicon, the next recipe takes a more active role by adding a link to each post or page, allowing visitors to e-mail a link to the item that they are currently viewing. This functionality is implemented using a filter hook attached to the page and post content, allowing our custom function to append custom output code to all entries that get displayed on-screen.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-email-page-link`.
3. Navigate to this directory and create a new text file called `ch2-email-page-link.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Email Page Link`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Visit an icon download website such as <http://iconarchive.com> and download an e-mail icon in a small size (32 x 32 pixels) in PNG format to the new plugin's directory, giving it the name `mailicon.png`.
7. Add the following line of code to register a function that will be called when WordPress is preparing data to display the content of a post or page:

```
add_filter( 'the_content', 'ch2ep1_email_page_filter' );
```

8. Add the following code section to provide an implementation for the `ch2ep1_email_page_filter` function:

```
function ch2ep1_email_page_filter ( $the_content ) {  
  
    // build url to mail message icon downloaded  
    // from iconarchive.com  
    $mail_icon_url = plugins_url( 'mailicon.png', __FILE__ );  
  
    // Set initial value of $new_content variable to previous  
    // content  
    $new_content = $the_content;
```

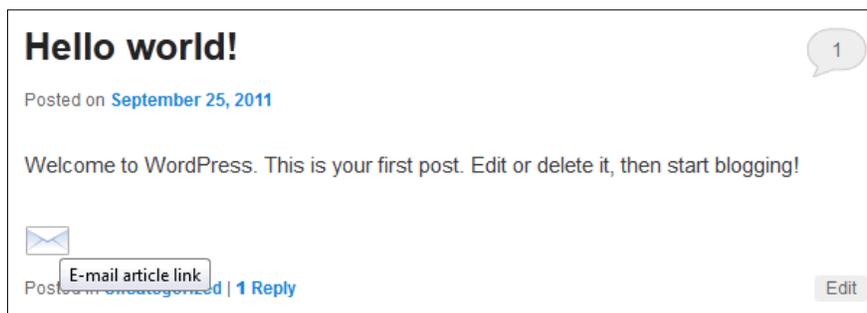
```

// Append image with mailto link after content, including
// the item title and permanent URL
$new_content .= "<a title='E-mail article link'
    href='mailto:someone@somewhere.com?subject=Check out
    this interesting article entitled ";
$new_content .= get_the_title();
$new_content .= "&body=Hi!%0A%0AI thought you would enjoy
    this article entitled ";
$new_content .= get_the_title();
$new_content .= ".%0A%0A";
$new_content .= get_permalink();
$new_content .= "%0A%0AEnjoy!'">
    <img alt='' src='' . $mail_icon_url. '' /></a>";

// Return filtered content for display on the site
return $new_content;
}

```

9. Save and close the plugin file.
10. Log in to the administration page of your development WordPress installation.
11. Click on **Plugins** in the left-hand navigation menu.
12. Activate your new plugin.
13. Visit your website to see the new mail icon at the end of each post and page.
14. Click on one of the mail links. Your mail client will come up with information about the item you were reading. The only information that needs to be updated is the recipient address and visitors can quickly send an e-mail.



How it works...

Similar to the previous recipe, this plugin uses the `add_filter` function to register a custom function to be called by WordPress as it prepares an item's content to be displayed on-screen. When the filter function is called, the first action that it performs is to create a URL to the e-mail icon that was downloaded in the recipe. It then goes on to modify the original content by appending the HTML code to display a `mailto` link. The same technique could be used to create links to popular social media and link sharing sites, with simple changes to the syntax of the link. Once the new content is ready, it is returned back to WordPress to be sent to any other registered filters and subsequently be displayed on the site.

There's more...

This recipe also introduces a pair of useful WordPress utility functions to get access to the current item's content.

get_the_title and get_permalink functions

While these two functions are mainly seen within theme template files, they can also be used by plugins to get easy access to the content of items that are currently being processed.

More specifically, the two utility functions that are used in this recipe are as follows:

- ▶ `get_the_title()`: This function gives us quick access to the item's title
- ▶ `get_permalink()`: A function that returns the item's permalink (a URL that is always associated with this post or page even after it is no longer featured on a website's front page)

See also

- ▶ *Creating a plugin file and header recipe*
- ▶ *Using WordPress path utility functions to load external files and images recipe*
- ▶ *Modifying the page title using plugin filters recipe*

Inserting link statistics tracking code in page body using plugin filters

After creating two filter functions that append text to the existing content, this recipe shows you how to modify the page content before it is displayed on-screen. More specifically, the following plugin will expand on the Google Analytics header plugin created earlier and add a JavaScript function to all links that are included in posts and pages to track when they are clicked by visitors.

Getting ready

You should have already followed the *Adding output content to page headers using plugin actions* recipe to have a starting point for this recipe. Alternatively, you can download the resulting code for that recipe from the Packt Publishing website (<http://www.packtpub.com/support>).

How to do it...

1. Navigate to the `ch2-page-header-output` folder in the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function that will be called when WordPress is preparing data to display a page or post's content:

```
add_filter( 'the_content', 'ch2lfa_link_filter_analytics' );
```

4. Add the following code section to provide an implementation for the `ch2lfa_link_filter_analytics` function:

```
function ch2lfa_link_filter_analytics ( $the_content ) {
    $new_content = str_replace( "href",
        "onClick=\"recordOutboundLink(this, 'Outbound Links', '" .
            home_url() . "' );return false;\" href", $the_content );

    return $new_content;
}
```

5. Add the following line of code to register a function that will be called when WordPress renders the page footer:

```
add_action( 'wp_footer', 'ch2lfa_footer_analytics_code' );
```

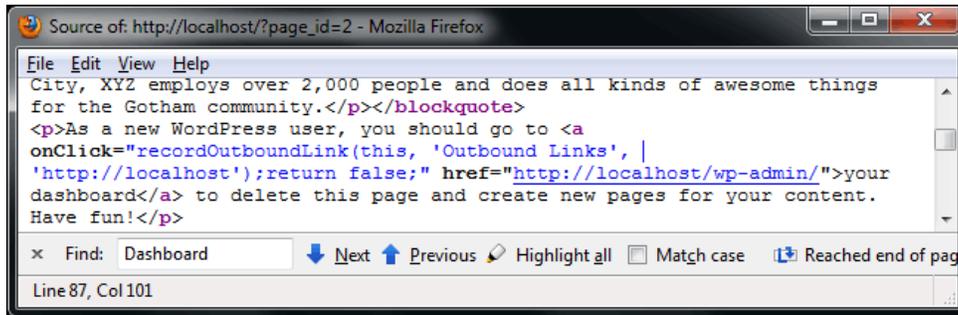
6. Add the following code section to provide an implementation for the `ch2lfa_footer_analytics_code` function:

```
function ch2lfa_footer_analytics_code() { ?>

<script type="text/javascript">
    function recordOutboundLink( link, category, action ) {
        _gat._getTrackerByName()._trackEvent( category, action );
        setTimeout( 'document.location = "' + link.href + "'", 100 );
    }
</script>

<?php }
```

7. Save and close the plugin file.
8. In your web browser, refresh your website and navigate to a page that contains one or more links in its content. The default Sample Page that is created in a new WordPress installation contains a link to the **Dashboard** page.
9. Open the Source View for the page and find the link to the Dashboard. You will see that the link tag has additional `onClick` Javascript code that will be called when visitors follow it:



10. Scroll to the bottom of the page to see the implementation of the `recordOutboundLink` Javascript function that was added to the page footer.

How it works...

The content filter function that is put in place by calling `add_filter` receives the entire content of all posts and pages—before they are rendered to the browser—and is allowed to make any number of changes to this information. In this case, we are using the PHP `str_replace` function to search for any occurrence of the string `href`, which indicates a link. When the string is found, it is replaced with a call to a JavaScript function as well as the original `href` tag.

To make this plugin complete, it also needs to provide an implementation for the JavaScript `recordOutboundLink` function. This is done by registering a custom function with the `wp_footer` hook that will output extra content with the function code in the website's footer.

The resulting plugin automates many of the tasks related to tracking usage data on a website using Google Analytics.

See also

- ▶ *Adding output content to page headers using plugin actions recipe*
- ▶ *Adding text after each item's content using plugin filters recipe*

Troubleshooting coding errors and printing variable content

As you transcribe code segments from the pages of this book or start writing your own plugins, there is a strong chance that you will have to troubleshoot problems with your code or have trouble working with data that your plugin is meant to manipulate. This recipe shows the basic techniques to identify and quickly resolve these errors while creating a plugin that will hide an item from the navigation menu for users who are not logged in to your site.

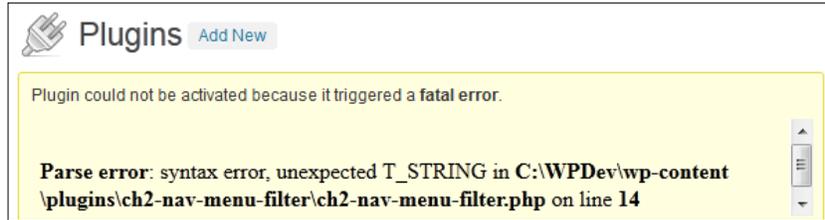
How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-nav-menu-filter`.
3. Navigate to this directory and create a new text file called `ch2-nav-menu-filter.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Nav Menu Filter`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code to register a function that will be called when WordPress is preparing data to display the site's navigation menu:

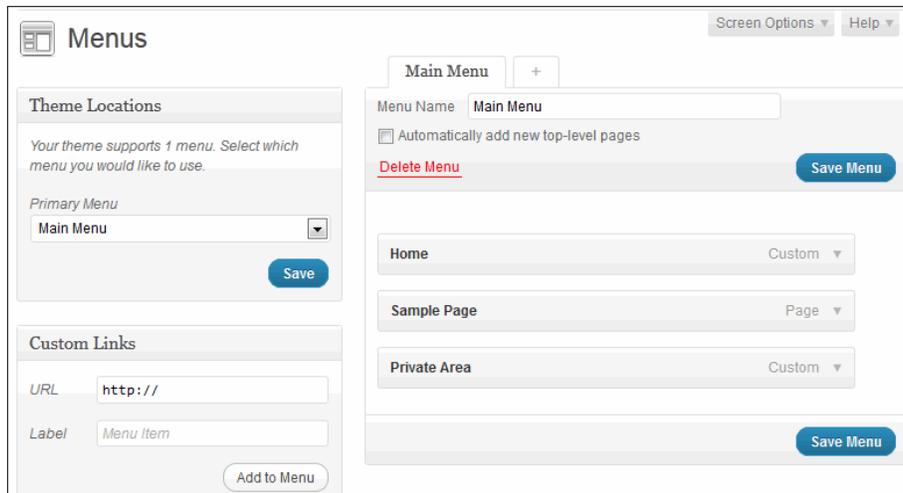
```
add_filter( 'wp_nav_menu_objects',  
           'ch2nmf_new_nav_menu_items', 10, 2 );
```
7. Add the following code section to provide an implementation for the `ch2nmf_new_nav_menu_items` function. Notice that the word `functio` is mistyped on purpose at the beginning of the first line:

```
functio ch2nmf_new_nav_menu_items( $sorted_menu_items, $args ) {  
    print_r( $sorted_menu_items );  
    return $sorted_menu_items;  
}
```
8. Save the plugin file and leave your code editor open.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.

- WordPress will display a fatal error message indicating that the plugin could not be activated since a syntax error was found. It also indicates the exact filename and line where the error occurred, helping to narrow down where the problem occurred:



- Go back to your code editor, correct the spelling of the word "function", and save the file.
- Activate the plugin a second time. It should now activate correctly.
- Back in the code editor, remove the last letter of the "function" word to reintroduce a syntax error.
- Refresh your website. You will now see that the entire site has disappeared and your browser only displays a blank page with an error message similar to the one we just saw.
- Correct the spelling error once again and your website will go back to normal.
- In the WordPress dashboard, navigate to the **Appearance | Menus** item and check if your current installation is using custom menus.
- If your site does not currently use a menu, create a new menu and associate it to a valid Theme Location.
- Add links to the **Home** and **Sample Page** pages in your new menu.
- Create a third item in your menu as a Custom Link called **Private Area** that points to the address `/privatearea`:



22. Click on the **Save Menu** button to store all of your updates.
23. Refresh your website and you will now see a lot of information printed before the navigation menu. This output is generated by the `print_r` function and is meant to help us understand how the data received by our filter function is organized. Once we have a good understanding of that data, we will be able to properly make changes to this information.
24. Replace the `print_r` function call inside of the filter function with the following code:

```
// Check if user is logged in, continue if not logged
if ( is_user_logged_in() == FALSE ) {
    // Loop through all menu items received
    // Place each item's key in $key variable
    foreach ( $sorted_menu_items as $key => $sorted_menu_item ) {
        // Check if menu item title matches search string
        if ( $sorted_menu_item->title == "Private Area" ) {
            // Remove item from menu array if found using
            // item key
            unset( $sorted_menu_items[ $key ] );
        }
    }
}
```

25. Refresh your website and you will see that the large array printout has disappeared. If you are logged in as the administrator, you will also notice the **Private Area** link in your menu. Log out to hide the menu item.

How it works...

As WordPress assembles a list of all available plugins to display them in the administration interface, it does not check to see if each plugin's PHP code is valid. This is actually done when a plugin is activated. At that time, any syntax error will be caught immediately and the newly-activated plugin will remain inactive, preventing a failure of the entire website.

That being said, once a plugin is activated, its code is evaluated every time WordPress renders a web page, and any subsequent code error that gets saved to the plugin file will cause the site to stop working correctly. For this reason, it is highly recommended to set up a local development environment, as shown in *Chapter 1, Preparing a Local Development Environment*, to avoid affecting a live site when an inevitable error creeps up in your plugin code. An alternative method would be to deactivate and reactivate plugins before making changes to them so that they are re-validated. With this method, the plugin's functionality won't be available on your site while you make changes, so this might not be optimal.

Once the code is working correctly, the second part of this recipe shows us how to visualize the information that is received by a custom filter function. While the WordPress Codex website provides great documentation about the purpose of most filters available, it does not go into details about the structure of the information that is sent to each filter function. Thankfully, the PHP `print_r` function comes in very handy since it can display the content of any variable on-screen, no matter what information is stored in the variable it receives as an argument.

Last but not least, the implementation of the custom filter function uses the WordPress API function `is_user_logged_in()` to see if the person viewing the site has provided login credentials, and then goes on to parse all menu items and remove the **Private Area** menu item if the visitor is not logged in.

There's more...

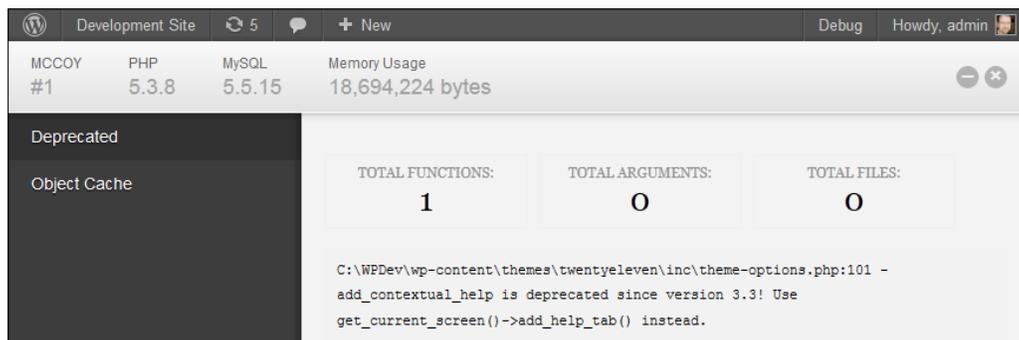
In addition to the debugging techniques used in this recipe, WordPress offers a number of built-in tools to facilitate plugin troubleshooting.

Built-in WordPress debugging features

While the `wp-config.php` file, located at the top of the WordPress file structure, is primarily used to store basic site configuration data, it can also be used to trigger a number of debugging features. The first of these is the debug mode, which will display all PHP errors, warnings, and notices at the top of site pages. For example, having this option active will show all undefined variables that you try to access in your code along with any deprecated WordPress function. To activate this tool, change the second parameter of the line defining the `WP_DEBUG` constant from `false` to `true` in `wp-config.php`:

```
define( 'WP_DEBUG', true );
```

To prevent debug messages from affecting the site's layout, you can download a useful plugin called Debug Bar (<http://wordpress.org/extend/plugins/debug-bar/>) to collect messages and display them in the admin bar:



MCOOY	PHP	MySQL	Memory Usage
#1	5.3.8	5.5.15	18,694,224 bytes

Deprecated

Object Cache

TOTAL FUNCTIONS:	TOTAL ARGUMENTS:	TOTAL FILES:
1	0	0

C:\WPDev\wp-content\themes\twentyeleven\inc\theme-options.php:101 -
add_contextual_help is deprecated since version 3.3! Use
get_current_screen()->add_help_tab() instead.

Other debugging features that can be activated from the `wp-config.php` file are as follows:

- ▶ `WP_DEBUG_LOG`: Stores all debug messages in a file named `debug.log` in the site's `wp-content` directory for later analysis
- ▶ `WP_DEBUG_DISPLAY`: Indicates whether or not error messages should be displayed on-screen
- ▶ `SAVEQUERIES`: Stores database queries in a variable that can be displayed in the page footer (see http://codex.wordpress.org/Editing_wp-config.php#Save_queries_for_analysis for more info)

See also

- ▶ *Modifying the page title using plugin filters* recipe

Creating a new simple shortcode

First introduced in WordPress 2.5, shortcodes became quite popular as a way to let users easily add content generated by plugins or themes to any page or post without needing to be familiar with PHP code and editing theme template files. As they are very simple to create, shortcodes can also be used to easily automate the output of content that repeatedly needs to be included in your site's content.

This recipe explains how to create a new custom shortcode that will be used to quickly add a link to a Twitter page in any post or page, automating a repetitive task.

How to do it...

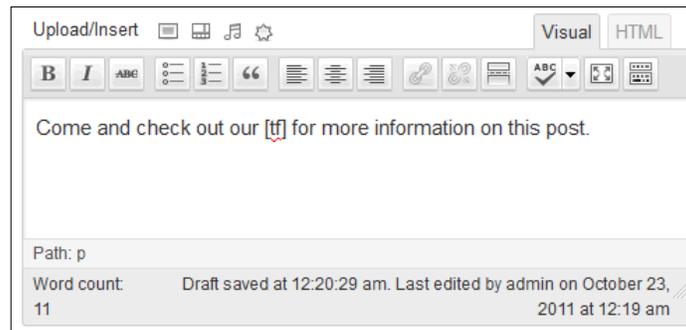
1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-twitter-shortcode`.
3. Navigate to this directory and create a new text file called `ch2-twitter-shortcode.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Twitter Shortcode`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code to declare a new shortcode, simply using the two characters `'tf'`, and specify the name of the function that should be called when the code is encountered in posts or pages:

```
add_shortcode( 'tf', 'ch2ts_twitter_feed_shortcode' );
```

7. Add the following code section to provide an implementation for the `ch2ts_twitter_feed_shortcode` function:

```
function ch2ts_twitter_feed_shortcode( $atts ) {
    $output = '<a href="http://twitter.com/ylefebvre">
        Twitter Feed</a>';

    return $output;
}
```
8. Save and close the plugin file.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.
12. Edit an existing post on your site and use the shortcode `[tf]` in the code editor.



13. Save and view the post to see that the shortcode was replaced by a link to a Twitter page attached to the words **Twitter Feed**.

How it works...

Shortcodes have similarities with both action hooks and filter hooks, since their associated custom function is called when it is time to perform a task just like an action hook, but it must return its output through a return value just like a filter hook. In terms of external data, the function associated with a shortcode will receive data in the case of some types of codes while it will only produce output in other cases.

When used in the text of a post or page, any shortcode surrounded by a pair of square brackets is identified by the WordPress engine, which then searches for functions registered for that specific code. If found, the associated function is called and the expected result is used to replace the original shortcode text in the item's content. Just like filter functions, shortcode functions must not output any text directly since it would likely appear in an unexpected place in the page layout, as WordPress calls all shortcode-processing functions before displaying the body of an item.

For simple shortcodes like we have in this recipe, the plugin function associated with it must return information but it does not receive any additional data through function parameters. That being said, it can rely on utility functions such as `get_the_ID`, `get_the_title`, and other WordPress utility functions to be able to produce the appropriate output. Other types of shortcodes seen in the later recipes will have more context and configuration options. It is also possible for shortcodes to access stored options data, which will be covered in *Chapter 3, User Settings and Administration Pages*.

See also

- ▶ *Creating a plugin file and header recipe*

Creating a new shortcode with parameters

While simple shortcodes already provide a lot of potential to output complex content to a page by entering a few characters in the post editor, shortcodes become even more useful when they are coupled with parameters that will be passed to their associated processing function. Using this technique, it becomes very easy to create a shortcode that accelerates the insertion of a media embed code in WordPress posts or pages by only needing to specify the shortcode and the unique identifier of the media element to be displayed.

We will illustrate this concept in this recipe by creating a shortcode that will be used to quickly add YouTube videos to posts or pages.

How to do it...

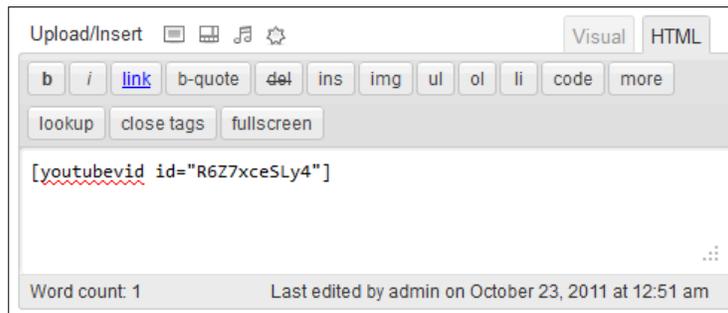
1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-youtube-embed`.
3. Navigate to this directory and create a new text file called `ch2-youtube-embed.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - YouTube Embed`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code to declare a new shortcode and specify the name of the function that should be called when the shortcode is found in posts or pages:

```
add_shortcode( 'youtubevid', 'ch2ye_youtube_embed_shortcode' );
```

7. Add the following code section to provide an implementation for the `ch2ye_youtube_embed_shortcode` function:

```
function ch2ye_youtube_embed_shortcode( $atts ) {  
  
    extract( shortcode_atts( array(  
        'id' => ''  
    ), $atts ) );  
  
    $output = '<iframe width="560" height="315"  
src="http://www.youtube.com/embed/' . $id . '"  
frameborder="0" allowfullscreen></iframe>';  
    return $output;  
}
```

8. Save and close the plugin file.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.
12. Create a new post and use the shortcode `[youtubevid id='R6Z7xceSLy4']` in the post editor, where `R6Z7xceSLy4` is the code of a video on YouTube.



13. Save and view the post to see that the shortcode was replaced by an embedded YouTube video on your site.

How it works...

When shortcodes are used with parameters, these extra pieces of data are sent to the associated processing function in the `$atts` parameter variable. By using a combination of the standard PHP `extract` and WordPress-specific `shortcode_atts` functions, our plugin is able to parse the data sent to the shortcode and create an array of identifiers and values that are subsequently transformed into PHP variables that we can use in the rest of our shortcode implementation function. In this specific example, we expect a single variable to be used, called `id`, which will be stored in a PHP variable called `$id`.

Once we have access to the video ID, we can put together the required HTML code that will embed a video player in our post and display the selected video file.

While this example only has one argument, it is possible to define multiple parameters for a shortcode.

See also

- ▶ [Creating a new simple shortcode recipe](#)

Creating a new enclosing shortcode

A different type of shortcode is available in WordPress that encloses content in posts and pages. Using a syntax similar to HTML tags, enclosing shortcodes can be used to identify parts of an item's content that need to be treated in a special way. For example, it is possible to use this type of shortcode to style a part of the post.

As an example of how to create enclosing shortcodes, this recipe shows you how to create a set of tags that will identify part of a post or page that should only be shown to visitors that are logged in to a site. In this way, the shortcode acts similarly to a filter hook, with the added bonus that you do not need to parse for instances of these tags as would normally need to be done in a filter.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch2-private-item-text`.
3. Navigate to this directory and create a new text file called `ch2-private-item-text.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 2 - Private Item Text`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code to declare a new shortcode and specify the name of the function that should be called when the shortcode is found in posts or pages:

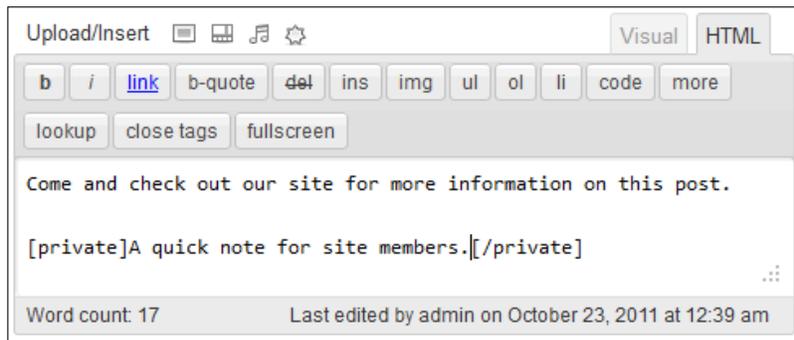
```
add_shortcode( 'private', 'ch2pit_private_shortcode' );
```

7. Add the following code section to provide an implementation for the `ch2pit_private_shortcode` function:

```
function ch2pit_private_shortcode( $atts, $content = null ) {
    if ( is_user_logged_in() )
        return '<div class="private">' . $content . '</div>';
}
```

```
else
    return '';
}
```

8. Save and close the plugin file.
9. Log in to the administration page of your development WordPress installation.
10. Click on **Plugins** in the left-hand navigation menu.
11. Activate your new plugin.
12. Create a new post and wrap some of the content with the **[private]** and **[/private]** tags:



13. Save and view the post to see that the text is visible while you are logged in to your site.
14. Log out and refresh the page to see that the enclosed text is not visible to regular visitors and is not found in the HTML source output either.

How it works...

Similar to a filter function, enclosing shortcodes receive a copy of the text that has been wrapped with the new tags. It is then possible to return this text with additional HTML code, or completely replace it with new content. In this specific case, we used the `is_user_logged_in` WordPress function to determine if the current visitor is logged in to the site. Based on the result of that query, the code determines if the original content should be displayed with some additional styling code or if it should not be displayed at all.

See also

- ▶ *Creating a new simple shortcode recipe*

Loading a stylesheet to format plugin output

When a plugin adds custom content or inserts styling tags to a post or page's existing content—as was done in the previous recipe showing how to create an enclosing shortcode—it usually needs to load a custom stylesheet to style these new elements. This recipe shows how to add a stylesheet in the WordPress style queue to format the private output created in the previous recipe. This queue is processed when the page header is rendered, listing all stylesheets that need to be loaded to display the site correctly.

Getting ready

You should have already followed the *Creating a new enclosing shortcode* recipe to have a starting point for this recipe. Alternatively, you can download the resulting code of that recipe from the Packt Publishing website (<http://www.packtpub.com/support>).

How to do it...

1. Navigate to the `ch2-private-item-text` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-private-item-text.php` file in a text editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function that will be called at the beginning of the WordPress page display process:

```
add_action( 'wp_enqueue_scripts', 'ch2pit_queue_stylesheet' );
```

4. Add the following code section to provide an implementation for the `ch2pit_queue_stylesheet` function:

```
function ch2pit_queue_stylesheet() {  
    wp_enqueue_style( 'privateshortcodestyle',  
                    plugins_url( 'stylesheet.css', __FILE__ ) );  
}
```
5. Save and close the plugin file.
6. Create a new text file in the plugin's directory called `stylesheet.css` and open it in a code editor.
7. Add the following content to the file:

```
.private {  
    color: #6E6A6B;  
}
```

8. Save and close the text file.
9. Navigate to your website, making sure you are logged in, and refresh the page containing the private text content. You should notice that the text is now displayed in gray.

How it works...

While it would have been possible to write straight HTML code to load the CSS file by registering a function with the `wp_head` action hook as we have done previously, WordPress has utility functions designed to help avoid loading duplicate stylesheets or scripts on a site. In this specific example, `wp_enqueue_script` is used to place the plugin's stylesheet file in a queue that will be processed when the plugin header is rendered, with the associated name `privateshortcodestyle`. Once WordPress has processed all plugins and boiled down all stylesheet requests to single instances, it will output the necessary HTML code to load all of them.

The content of the `stylesheet.css` file is normal CSS code that specifies that any text that is assigned the private class should be displayed in gray.

See also

- ▶ *Creating a new enclosing shortcode recipe*

Writing plugins using object-oriented PHP

So far, all plugin examples that have been covered in this chapter have been written using the procedural PHP programming style, with all functions declared directly in the main body of the plugin and the hook registration functions having direct access to these functions.

WordPress can also be written using an object-oriented PHP approach. This recipe explains how to convert the code from the previous recipe into a class-based version of the same functionality.

Getting ready

You should have already followed the *Loading a stylesheet to format plugin output* recipe to have a starting point for this recipe. Alternatively, you can download the resulting code from that recipe from the Packt Publishing website (<http://www.packtpub.com/support>).

How to do it...

1. Log in to the administration page of your WordPress installation.
2. Click on **Plugins** in the left-hand navigation menu.
3. Check if the Chapter 2 - Private Item Text plugin is currently active and deactivate it if it is.
4. Copy the entire contents of the `ch2-private-item-text` directory and name the copy `ch2-oo-private-item-text`.
5. Navigate to the newly renamed folder and rename the main PHP code file to `ch2-oo-private-item-text.php`.
6. Open the newly renamed plugin file in a code editor.
7. Update the plugin header to change the name of the plugin to `Chapter 2 - Object-Oriented - Private Item Text`.
8. Right after the plugin header, add the following text to declare a new class for our plugin and specify a constructor function for this class:

```
class CH2_OO_Private_Item_Text {  
    function __construct() {  
    }  
}  
  
$my_ch2_oo_private_item_text = new CH2_OO_Private_Item_Text();
```

9. Move the calls to the `add_shortcode` and `add_action` functions to be placed inside of the class constructor.
10. Modify the second argument of the `add_shortcode` and `add_action` functions as follows:

```
add_shortcode( 'private', array( $this,  
                                'ch2pit_private_shortcode' ) );  
  
add_action( 'wp_enqueue_scripts', array( $this,  
                                          'ch2pit_queue_stylesheet' ) );
```
11. Move the complete `ch2pit_private_shortcode` and `ch2pit_queue_stylesheet` functions inside of the class body.
12. Save and close the modified file.

13. Log in to the administration page of your development WordPress installation.
14. Click on **Plugins** in the left-hand navigation menu.
15. Activate the new plugin.
16. Visit your site to see that the private item content functionality is still in place and works as it did before.

How it works...

The code changes that we applied to the plugin first declares a class for all of our plugin's functionality and also contains a constructor function for that class. The constructor function is called once as soon as the class is instantiated by the last line in the plugin's code and can be used to associate custom functions with all action hooks, filter hooks, and shortcodes.

The main benefit of using an object-oriented approach is that you don't have to be as careful when naming your hook callbacks and all other functions, since these names are local to the class and can be the same as function names declared in any other classes or in procedural PHP code.

See also

- ▶ *Creating a new enclosing shortcode recipe*

3

User Settings and Administration Pages

This chapter is focused on setting up pages that enable users to configure plugin settings. It covers the following topics:

- ▶ Creating default user settings on plugin initialization
- ▶ Storing user settings using arrays
- ▶ Removing plugin data on deletion
- ▶ Creating an administration page menu item in the Settings menu
- ▶ Creating a multi-level administration menu
- ▶ Hiding items which users should not access from the default menu
- ▶ Rendering the admin page contents using HTML
- ▶ Processing and storing plugin configuration data
- ▶ Displaying a confirmation message when options are saved
- ▶ Adding custom help pages
- ▶ Rendering the admin page contents using the Settings API
- ▶ Accessing user settings from action and filter hooks
- ▶ Formatting admin pages using meta boxes
- ▶ Splitting admin code from the main plugin file to optimize site performance
- ▶ Storing stylesheet data in user settings
- ▶ Managing multiple sets of user settings from a single admin page

Introduction

As we saw in the previous chapter, it is very easy for a plugin to register custom functions with action and filter hooks to change or augment the way WordPress renders web pages. That being said, some of the examples covered in that chapter have limitations when it comes to dealing with custom user information such as the inability to specify a Google Analytics account number or the restricted filename and location for the favicon file to be associated with a website.

To make plugins easy to use for a wide audience, it is usually important to create one or more administration pages where users will be able to provide details that are specific to their installation, enter information on external accounts, and customize some of the aspects of the plugin's functionality. As an example, the Akismet plugin, provided in default WordPress installations, offers a configuration page that can be found under the **Plugins | Akismet** configuration menu. Thankfully, WordPress has a rich set of functions that allows plugin developers to easily put together configuration pages that will seamlessly blend with the rest of the administrative panels.

This chapter covers how to use the WordPress Options **Application Programming Interface (API)** functions to store and access user options in the site database. It then goes on to explain how to create custom dialogs to provide users with complete control over the configuration of the plugins that you create.

Creating default user settings on plugin initialization

A typical first step of most user-configurable plugins is to create a default set of values for all options when the plugin is activated. These default options will subsequently be used to populate the plugin's settings page when it is visited by the site administrator. This recipe shows how to register a function that is called when a plugin is activated and how to store option data in the site database.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch3-individual-options`.
3. Navigate to this directory and create a new text file called `ch3-individual-options.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 3 - Individual Options`.

5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the remainder of the PHP code.
6. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function that will be executed when the plugin is activated, after its initial installation or following an upgrade:

```
register_activation_hook( __FILE__,
                        'ch3io_set_default_options' );
```

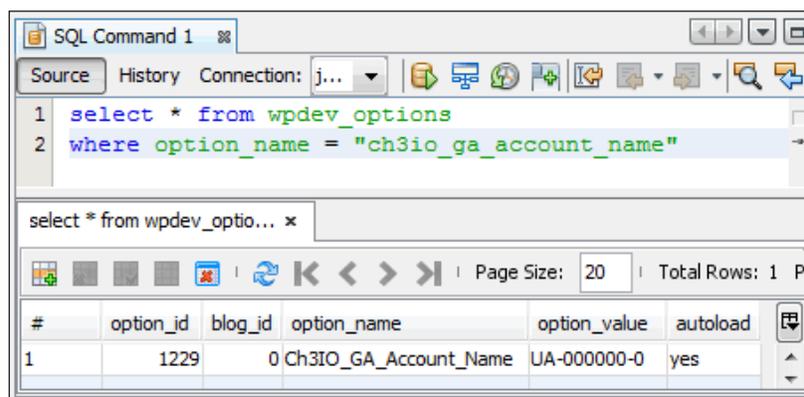
7. Add the following code section to provide an implementation for the `ch3io_set_default_options` function:

```
function ch3io_set_default_options() {
    if ( get_option( 'ch3io_ga_account_name' ) === false ) {
        add_option( 'ch3io_ga_account_name', "UA-000000-0" );
    }
}
```

8. Save and close the plugin file. Execute the activation function that was just added by clicking on the **Activate** option of the **Chapter 3 – Individual Options** plugin.
9. Using your web server's MySQL database administration tool or the **Services** section of the NetBeans development interface, query the **wpdev_options** table of your WordPress installation for an option named **ch3io_ga_account_name**.

```
select * from wpdev_options
where option_name = "ch3io_ga_account_name"
```

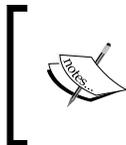
10. Your query should return a single row with the default value assigned to the new option, as shown in the following screenshot:



How it works...

The `register_activation_hook` function is used to indicate to WordPress the name of the function that should be called when it activates the plugin. Unlike other hooks, this function requires the name of the main plugin code file to be sent as its first argument, along with the name of the associated function. To do this easily, we can leverage the PHP `__FILE__` constant as the first argument, which will resolve to the filename.

When the callback function is called, we can use the Options API to create, update, or delete settings in the options table of the site's MySQL database. In this specific example, we are using the `add_option` function to easily create an option called `ch3io_ga_account_name` with a default value of `UA-000000-0`.



Just like function names, you should be careful when naming plugin options to avoid conflicts with other plugins. A good practice is to add a unique prefix to the beginning of each variable name.

Before making a call to create the new option, the activation function checks if the option is present in the WordPress options table using the `get_option` function. If the return value is false, indicating that the option was not found, a new default option can be created. Any other result would show that the plugin has been activated on the site previously and that options may have been changed from their default values. It is important to keep in mind when writing this code that plugins get deactivated and reactivated each time they are updated using the WordPress update tool, resulting in a call to their activation function. It is also possible that a user might have deactivated a plugin temporarily to debug site issues and brought it back at a later time, also resulting in the activation function getting called.

Finally, it should be noted that it is possible to call the `add_option` function multiple times if more than one option is needed to implement a plugin's desired functionality. That being said, it is not necessary to verify the presence of all options as checking for a single one would indicate that they were all previously set.

There's more...

Beyond the creation of default values for a plugin, the activation hook also plays a role in the plugin upgrade process and in more advanced concepts such as custom database tables. In contrast, the similar deactivation function hook does not have a real use within the creation of most plugins.

Adding new options when upgrading plugins

While the example code in this recipe creates a single new plugin option, it does not do anything if the option already exists. When creating new versions of a plugin, you might need to create new options and to initialize these new parameters to default values. This can be addressed with a few simple changes to the activation function to store a plugin version number when the plugin is first installed and create new options when the plugin is re-activated as part of the upgrade process.

```
function ch3io_set_default_options() {
    if ( get_option( 'ch3io_version' ) === false ) {
        add_option( 'ch3io_ga_account_name', 'UA-000000-0' );
        add_option( 'ch3io_track_outgoing_links', 'false' );
        add_option( 'ch3io_version', '1.1' );
    } elseif ( get_option( 'ch3io_version' ) < 1.1 ) {
        add_option( 'ch3io_track_outgoing_links', 'false' );
        update_option( 'ch3io_version', '1.1' );
    }
}
```

In the previous example, users of any version older than 1.1 will have a new option added to their installation when they upgrade, while new users will have all options created when they first activate. This will ensure that users performing an upgrade will not lose any changes that they made to the plugin options. As part of the upgrade process, it is important to update the version number to the most recent version, in addition to creating new options. This will avoid re-creating these new options each time the activation function is executed.

Creating new tables and initializing custom post type data

While entries in the `options` table are the most common elements that get created in the initialization function, it is also possible to perform more advanced tasks such as creation of custom database tables or the initialization of data related to custom post types. These advanced operations will be described in the later chapters.

Deactivation function

Similar to the activation function that we used in this recipe, WordPress provides a way to register a deactivation function (using `register_deactivation_hook`). While it may be tempting to use this function to remove options created by the plugin, it is not possible to know why the activation function was called. The three situations that could trigger this call are a plugin upgrade, a temporary deactivation to debug a site problem, or just before the plugin gets deleted. Since it is best to keep user options in the first two situations, any clean up and data removal code should be placed in a plugin's uninstallation file instead, as described in a later recipe.

See also

- ▶ [Removing plugin data on deletion recipe](#)

Storing user settings using arrays

While the previous recipe worked quite well in creating entries in the site's options table for each individual plugin option, another way to manage user settings is to store them as arrays in the database.

This recipe creates the same options as the previous one but uses an array instead of individual options to store them. It also incorporates the upgrade code strategy discussed in the *There's more...* section of the previous recipe.

Getting ready

You should have already followed the recipe entitled *Inserting link statistics tracking code in page body using plugin filters* in *Chapter 2, Plugin Framework Basics*, to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch2-page-header-output\ch2-page-header-output-v2.php`) from the code bundle downloaded from the Packt Publishing website (<http://www.packtpub.com/support>) and rename the file to `ch2-page-header-output.php`.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the file `ch2-page-header-output.php` in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when the plugin gets activated:

```
register_activation_hook( __FILE__,  
                        'ch2pho_set_default_options_array' );
```

4. Add the following code section to provide an implementation for the `ch2pho_set_default_options_array` function:

```
function ch2pho_set_default_options_array() {  
    if ( get_option( 'ch2pho_options' ) === false ) {  
        $new_options['ga_account_name'] = "UA-000000-0";  
        $new_options['track_outgoing_links'] = false;  
        $new_options['version'] = "1.1";  
        add_option( 'ch2pho_options', $new_options );  
    }  
}
```

```

    } else {
        $existing_options = get_option( 'ch2pho_options' );
        if ( $existing_options['version'] < 1.1 ) {
            $existing_options['track_outgoing_links'] = false;
            $existing_options['version'] = "1.1";
            update_option( 'ch2pho_options', $existing_options );
        }
    }
}

```

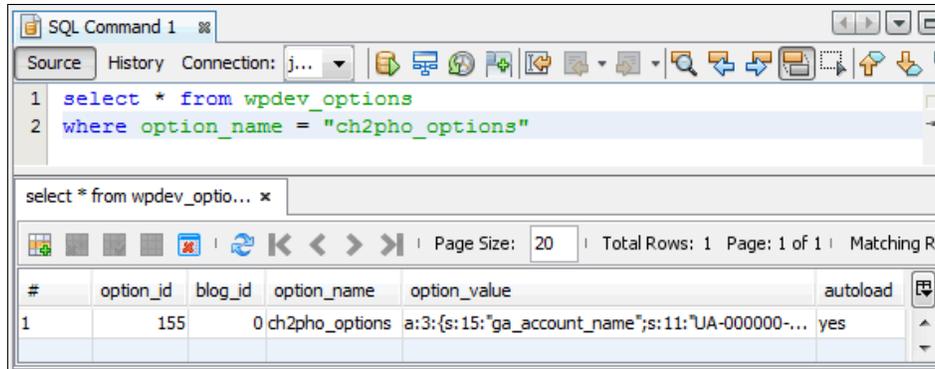
5. Save and close the plugin file.
6. Go to the **Plugins** section of the administration interface.
7. Click on the **Deactivate** link for the **Chapter 2 - Page Header Output** plugin, followed by a click on the **Activate** link to execute the activation function that was just added.
8. Using your web server's MySQL database administration tool or the **Services** section of the NetBeans development interface, query the `wpdev_options` table of your WordPress installation for an option with the name `ch2pho_options`.

```

select * from wpdev_options
where option_name = "ch2pho_options"

```

9. Your query should return a single row with a serialized set of data representing all of the fields in the array.



How it works...

The Options API functions accept values as single variables or arrays of data. When given an array, they transform the information received to a serialized array that gets stored in the site database. The main advantage of using arrays over multiple options is that all of the information can be retrieved with a single function call, optimizing the access to the MySQL database. This is especially important when your plugin options need to be queried every time a site page needs to be rendered.

Of course, this advantage is only true if you need to use most plugin options at the same time. Otherwise, your code will be managing large amounts of data for no reason.

Another benefit of this method is that the names of each option can be much shorter and simpler since you only need to worry about avoiding naming conflicts at the top option name level as opposed to each key in the array. Finally, having all options stored in a single array makes the bulk removal of these options much easier than if they were all stored separately, as we will see in the next recipe.

Similar to the previous recipe, this example handles the creation of all options when the plugin is first activated and the addition of new options for users that are upgrading from a previous version.

See also

- ▶ *Adding output content to page headers using plugin actions* recipe in *Chapter 2, Plugin Framework Basics*
- ▶ *Inserting link statistics tracking code in page body using plugin filters* recipe in *Chapter 2, Plugin Framework Basics*
- ▶ *Removing plugin data on deletion* recipe

Removing plugin data on deletion

As with any piece of software, it is quite possible that users might decide to remove a plugin from their WordPress installation if they no longer require the functionality that it provides or they have found an alternative that they prefer.

When this happens, the plugin author must decide if all of the configuration data stored in the site's database should be left in place, making it easier to re-install the plugin down the road, or to remove all of this information, leaving a clean database behind.

This recipe shows how to create a de-installation function that will remove options data from a site's database.

Getting ready

You should have already followed the *Storing user settings using arrays* recipe to have options data ready for deletion. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v3.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Create a new file called `uninstall.php`.
3. Open the new file in a text editor and add the following code to it:

```
<?php

    // Check that code was called from WordPress with
    // uninstallation constant declared

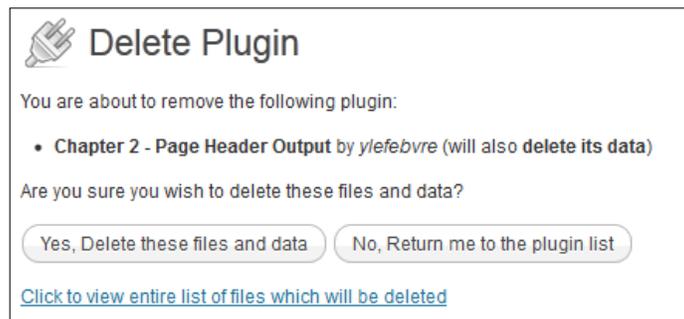
    if ( !defined( 'WP_UNINSTALL_PLUGIN' ) )
        exit;

    // Check if options exist and delete them if present

    if ( get_option( 'ch2pho_options' ) != false ) {
        delete_option( 'ch2pho_options' );
    }

?>
```

4. Save and close the plugin file.
5. Navigate to the administration page of your development WordPress installation.
6. Click on **Plugins** in the left-hand navigation menu.
7. **Deactivate** the **Chapter 2 - Page Header Output** plugin.
8. Make a copy of your plugin and uninstallation files to avoid losing them upon deletion of the plugin in the following steps. The copy should be moved outside of the `plugins` folder to avoid WordPress seeing two copies of the plugin.
9. Click the **Delete** link under the **Chapter 2 - Page Header Output** plugin.
10. Click on the **Yes, Delete these files and data** button to delete all plugin files.



11. Using your web server's MySQL database administration tool or the **Services** section of the NetBeans development interface, query the **wpdev_options** table of your WordPress installation for an option with the name **ch2pho_options** to see that the option has been deleted.

```
select * from wpdev_options
where option_name = "ch2pho_options"
```

How it works...

When a plugin is inactive and a site administrator clicks on its deletion link, WordPress checks for the presence of a file called `uninstall.php` in the plugin directory. If the file exists, it understands that it contains code designed to remove plugin data and settings and displays a message asking the user if they wish to delete the plugin files and data. If it is not found, it only asks the user to delete the plugin files.

Upon acknowledgment of the user, WordPress proceeds with the deletion of all plugin files and executes the contents of the `uninstall.php` file. This file should contain straight PHP code that deletes all plugin options and any other content created by the plugin's code. Once executed, the `uninstall` script will be deleted with the rest of the files.

Looking at the content of the `uninstall` script, the first few lines of code check for the presence of a constant that WordPress should have set before calling the script. If it is not present, the script will abort immediately for security purposes. This ensures that an external visitor knowing that a certain plugin is installed won't be able to try to delete it. Once the intent has been verified, the rest of the code checks for the existence of the `ch2pho_options` array that was created in the previous recipe and deletes it. If you created more than one option to store your configuration data, you will need to delete each option with individual calls to the `delete_option` function.

See also

- ▶ [Storing user settings using arrays recipe](#)

Creating an administration page menu item in the Settings menu

After defining default values for plugin configuration options, the next step is to create a place where users will be able to view and change these values. By using the WordPress API, we are able to create new items in the administration menu that will later allow us to create custom plugin configuration pages. This recipe shows how to create a new menu item that will appear under the **Settings** subsection of the administration menu.

Getting ready

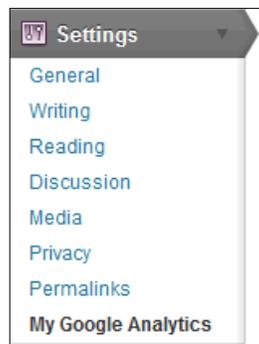
You should have already followed the *Storing user settings using arrays* recipe to have options data available to manage. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v3.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when WordPress is building the administration pages menu:


```
add_action( 'admin_menu', 'ch2pho_settings_menu' );
```
4. Add the following code section to provide an implementation for the `ch2pho_settings_menu` function:


```
function ch2pho_settings_menu() {
    add_options_page( 'My Google Analytics Configuration',
        'My Google Analytics', 'manage_options',
        'ch2pho-my-google-analytics', 'ch2pho_config_page' );
}
```
5. Save and close the plugin file.
6. Navigate to the administration page of your development WordPress installation.
7. Activate the **Chapter 2 - Page Header Output** plugin if you left it deactivated after following the previous recipe.
8. Click on the **Settings** section in the left-hand navigation menu to expand it. You will see a new menu item called **My Google Analytics** in the tree, created from the code that was just added to the plugin.



9. Click on the **My Google Analytics** menu item. You will see an error message displayed since WordPress cannot find the function intended to populate the configuration page. This error will go away once you perform the next recipe.

How it works...

The first line of code of this recipe registers a function to be called when WordPress is building the administration menu. When it is executed, the custom function that we created makes a call to the `add_options_page` function to add an item to the **Settings** menu. This function has a number of parameters that we will look at as follows:

```
add_options_page( $page_title, $menu_title, $capability,  
                 $menu_slug, $function );
```

The first two parameters are text strings that will be visible to site administrators, with the first one appearing in the browser title bar or tab title, and the second being the text of the submenu item that will appear under the **Settings** menu.

The third parameter is a bit more complicated and refers to the **user capability** required to be able to see and access this menu item. When creating users in a WordPress installation, each user is assigned one of the five default user roles (Subscriber, Administrator, Editor, Author, or Contributor). Each of these roles is mapped to a number of permissions that determine the actions that users with this role can perform. For a full list of roles and their associated capabilities, please refer to the WordPress Codex page on the topic (http://codex.wordpress.org/Roles_and_Capabilities). In this example, we used the user capability `manage_options`, which is assigned to users who have administrative rights on the site and to super admins, when working in a network WordPress installation.

The fourth menu item, the `menu_slug`, is a text string that will be used internally by WordPress to identify the menu item. This string should be unique to avoid conflicts with other plugins.



The `menu_slug` name should be all lowercase to ensure that more advanced functionalities, such as WordPress meta boxes, work correctly.

Finally, the last parameter specifies the name of the function to be called to display the contents of the configuration page when the submenu item is clicked.

The **Settings** menu is a perfect location for plugins that only require a single configuration page, as you may have seen when installing other plugins, while more complex plugins that require multiple menu sections should use the technique shown in the next recipe.

There's more...

While new items will always be located under the default **Settings** menu items created by WordPress (General, Writing, Reading, and so on), plugin developers do have some control over the location of their plugin in the list.

Settings hook priority to determine menu order

As mentioned in the previous chapter, when action hooks were first introduced, the `add_action` function's third parameter is used to indicate the priority of a registered callback over other functions registered for the same hook (in this case, the `admin_menu` hook). To ensure that the newly created menu item is as high as possible in the menu, the priority of the registered function can be set to a value of 1.

```
add_action( 'admin_menu', 'ch2pho_settings_menu', 1 );
```



It should be noted that other plugins can also set their callback to this priority. In such cases, alphabetical priority and activation sequence are other factors to determine which menu item will be displayed first after **Permalinks**.

See also

- ▶ [Storing user settings using arrays recipe](#)

Creating a multi-level administration menu

When plugins grow in complexity, their configuration options often grow in numbers, giving users a high level of flexibility in choosing how the plugin behaves on their site. While it is possible to display all plugin options on a single lengthy configuration page, creating a new top-level menu item with multiple sections can help organize parameters in logical groupings that will allow users to find what they are looking for more quickly.

This recipe shows how to create a new top-level menu item in the administration menu, with an accompanying submenu item.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch3-multi-level-menu`.
3. Navigate to this directory and create a new text file called `ch3-multi-level-menu.php`.

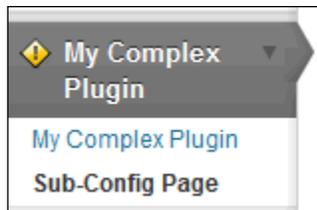
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 3 - Multi-Level menu`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add PHP code.
6. Add the following line of code to register a function that will be called when WordPress is preparing data to display the site's administration menu:

```
add_action( 'admin_menu', 'ch3mlm_admin_menu' );
```

7. Add the following code section to provide an implementation for the `ch3mlm_admin_menu` function:

```
function ch3mlm_admin_menu() {  
    // Create top-level menu item  
    add_menu_page( 'My Complex Plugin Configuration Page',  
        'My Complex Plugin', 'manage_options',  
        'ch3mlm-main-menu', 'ch3mlm_my_complex_main',  
        plugins_url( 'myplugin.png', __FILE__ ) );  
  
    // Create a sub-menu under the top-level menu  
    add_submenu_page( 'ch3mlm-main-menu',  
        'My Complex Menu Sub-Config Page', 'Sub-Config Page',  
        'manage_options', 'ch3mlm-sub-menu',  
        'ch3mlm_my_complex_submenu' );  
}
```

8. Save and close the plugin file.
9. Find and download a PNG format 16 x 16 pixel icon from a site such as IconArchive (<http://www.iconarchive.com>) and save it as `myplugin.png` in the plugin directory.
10. Navigate to the **Plugins** section of your site's administration area.
11. Activate your new plugin.
12. You will now see a new menu item in the administration menu.



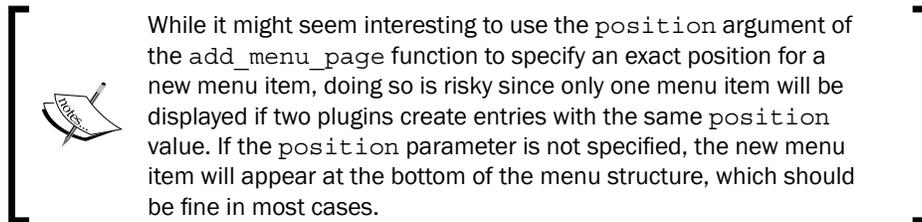
13. Expand the top-level new menu item to see the submenu item.

How it works...

The `add_menu_page` function is very similar to the `add_options_page` function seen in the previous recipe, with its first five parameters being identical.

```
add_menu_page( $page_title, $menu_title, $capability, $menu_slug,
    $function, $icon_url, $position );
```

The last two items are specific to this function, with the first allowing us to display a custom icon in the menu next to our new top-level item, and the second specifying where the new menu should be positioned within the administration menu.



Once the first menu item has been created, the `add_submenu_page` function can be used to attach a submenu item. The following are its parameters, which are virtually identical to the `add_options_page` function, except for the first parameter that should be the unique string identifier of the top-level menu item to which the submenu should be attached:

```
add_submenu_page( $parent_slug, $page_title, $menu_title,
    $capability, $menu_slug, $function );
```

While it is possible to use this technique to create top-level menu items for plugins with a single configuration page, these simpler extensions should create a single entry under the **Settings** menu, as shown in the previous recipe.

See also

- ▶ *Creating an administration page menu item in the Settings menu* recipe

Hiding items which users should not access from the default menu

Many users praise WordPress for its ease of use and streamlined administration interface. That being said, almost everyone who has deployed it to new users has instructed them to avoid certain menu items as they do not need to enter these sections and could potentially introduce site malfunctions if they modified settings in these areas.

A better solution than prevention through training is to use a few simple API functions to hide the undesired menu items. This recipe shows how to use these functions, introduced in version 3.1, to remove the Links editor and Permalinks settings menu items.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch3-hide-menu-item`.
3. Navigate to this directory and create a new text file called `ch3-hide-menu-item.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 3 - Hide Menu Item`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add PHP code.
6. Add the following line of code to register a function that will be called when WordPress is preparing data to display the site's navigation menu:

```
add_action( 'admin_menu', 'ch3hmi_hide_menu_item' );
```
7. Add the following code section to provide an implementation for the `ch3hmi_hide_menu_item` function, hiding the **Links** menu item:

```
function ch3hmi_hide_menu_item() {  
    remove_menu_page( 'link-manager.php' );  
}
```
8. Add an extra function call to the `ch3hmi_hide_menu_item` function to hide the **Permalinks** submenu item, found under the **Settings** menu:

```
remove_submenu_page( 'options-general.php',  
                    'options-permalink.php' );
```
9. Save and close the plugin file.
10. Navigate to the **Plugins** section of the administration interface.
11. Activate your new plugin.
12. Look at the administration menu to see that the **Links** menu is no longer visible.



13. Expand the **Settings** menu to see that the **Permalinks** submenu item is not visible either.

How it works...

The default WordPress administration menu uses the names of the PHP code files used to render each section as their unique identifiers. One way to quickly find out the identifier for a menu item is to hover the mouse cursor over it in a web browser and to look at the address that the link points to. In the case of the **Links** menu item, the URL is `http://localhost/wp-admin/link-manager.php`, thus the use of `link-manager.php` in the call to `remove_menu_page`.



A similar technique was used to determine the arguments to pass to the `remove_submenu_page` function, identifying that the Settings section has a URL of `http://localhost/wp-admin/options-general.php` while the **Permalinks** section has the address `http://localhost/wp-admin/options-permalink.php`.

Rendering the admin page contents using HTML

Once a custom menu item has been created, WordPress will call the function associated with it when it gets visited. The assigned function's main purpose is to render a configuration page containing a form with all options available to the user and to send the captured data back to WordPress for processing.

There are two main methods that can be used to render plugin configuration pages: straight HTML and the Settings API. This recipe explores the use of HTML to create a configuration panel while a later recipe will show how to use the Settings API to prepare the page output.

Getting ready

You should have already followed the *Creating an administration page menu item in the Settings menu* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v4.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.

3. Add the following lines of code after the existing functions and before the closing `?>` PHP command at the end of the file to implement the rendering code for the plugin options page:

```
function ch2pho_config_page() {
    // Retrieve plugin configuration options from database
    $options = get_option( 'ch2pho_options' );
    ?>

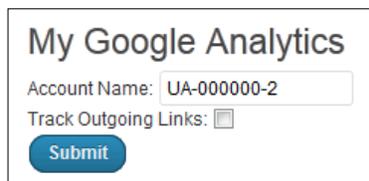
    <div id="ch2pho-general" class="wrap">
    <h2>My Google Analytics</h2>

    <form method="post" action="admin-post.php">

    <input type="hidden" name="action"
        value="save_ch2pho_options" />

    <!-- Adding security through hidden referrer field -->
    <?php wp_nonce_field( 'ch2pho' ); ?>
    Account Name: <input type="text" name="ga_account_name"
    value="<?php echo esc_html( $options['ga_account_name'] );
    ?>"/><br />
    Track Outgoing Links: <input type="checkbox"
    name="track_outgoing_links" <?php if (
    $options['track_outgoing_links'] ) echo ' checked="checked" ';
    ?>/><br />
    <input type="submit" value="Submit"
    class="button-primary"/>
    </form>
    </div>
    <?php }
```

4. Save and close the plugin file.
5. Click on the **Settings** section in the administration pages.
6. Click on the **My Google Analytics** menu item to display the plugin configuration page.



My Google Analytics

Account Name:

Track Outgoing Links:

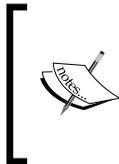
How it works...

Any output generated within the configuration page implementation function will be sent to the browser, enclosed within the WordPress administration interface layout. In this recipe's code, we first start by using the `get_option` function to retrieve all options for the plugin, conveniently organized in an array that we can store in a single variable.

We then use a closing PHP bracket to be able to write direct HTML code for the rest of the function's body, sending this content directly to the browser. The HTML code takes care of creating a standard form, rendering a text field to display and accept new values for the Google Analytics Account Number, and a checkbox for the user to specify whether or not outgoing links should be tracked. Finally, the HTML code adds a **Submit** button to allow users to submit any changes made to the plugin's configuration.

Taking a closer look at the code, it also contains small snippets of PHP code that display the current configuration values when the **Options** page is displayed.

The biggest advantage of using straight HTML to render a plugin's configuration page is that it allows for the creation of intricate layouts to present all of the options to the end user. This is in sharp contrast to using the Settings API, as we will see in a later recipe. HTML is also easier to understand for many web designers than working with intricate functions.



It should be noted that any changes submitted from this form in its current state won't be saved since we have not implemented the code necessary to process the submitted data and store it back in the `options` database table. This will be covered in the next recipe.

There's more...

As soon as user submission processing comes into play, it is important to think about security. The form that was created in this recipe is no exception.

wp_nonce_field

The `wp_nonce_field` function that was used in this recipe is part of a security measure to ensure that the data being sent for submission comes from the WordPress administration pages and not an external source. By adding this function call, a hidden text field is added to the plugin configuration form with information that will be checked when the post data is received.

While it is optional, the first argument of the function is a unique identifier that should always be set to ensure better security. If it is not set, default values will be used, facilitating security breaches. The function also has a number of other optional parameters as follows:

```
wp_nonce_field( $action, $name, $referer, $echo );
```

The other three arguments are used to specify a name for the nonce, which would need to be matched on the receiving end, a Boolean variable to indicate if the referer field should be set for validation and another Boolean parameter to determine if the hidden form field should be displayed or returned.

See also

- ▶ *Creating an administration page menu item in the Settings menu* recipe

Processing and storing plugin configuration data

With the configuration page in place, plugin users will be able to modify configuration options and submit them to be stored in the WordPress database. The missing link at this time is the creation of a data processing function that will receive the data posted by the user and store it in the site's `options` table.

This recipe describes how to implement a data processing function to validate that the information being sent for storage is legitimate and to store the information in an options array.

Getting ready

You should have already followed the *Rendering the admin page contents using HTML* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v5.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when WordPress first identifies that the requested page is an administration page:

```
add_action( 'admin_init', 'ch2pho_admin_init' );
```

4. Add the following code section to provide an implementation for the `ch2pho_admin_init` function:

```
function ch2pho_admin_init() {
    add_action( 'admin_post_save_ch2pho_options',
                'process_ch2pho_options' );
}
```

5. Add the following code section to provide an implementation for the `process_ch2pho_options` function that was declared in the previous step:

```
function process_ch2pho_options() {

    // Check that user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );

    // Check that nonce field created in configuration form
    // is present
    check_admin_referer( 'ch2pho' );

    // Retrieve original plugin options array
    $options = get_option( 'ch2pho_options' );

    // Cycle through all text form fields and store their values
    // in the options array
    foreach ( array( 'ga_account_name' ) as $option_name ) {
        if ( isset( $_POST[$option_name] ) ) {
            $options[$option_name] =
                sanitize_text_field( $_POST[$option_name] );
        }
    }

    // Cycle through all check box form fields and set the options
    // array to true or false values based on presence of
    // variables
    foreach ( array( 'track_outgoing_links' ) as $option_name ) {
        if ( isset( $_POST[$option_name] ) ) {
            $options[$option_name] = true;
        } else {
            $options[$option_name] = false;
        }
    }

    // Store updated options array to database
```

```
update_option( 'ch2pho_options', $options );

// Redirect the page to the configuration form that was
// processed

wp_redirect( add_query_arg( 'page',
                            'ch2pho-my-google-analytics',
                            admin_url( 'options-general.php' ) ) );

exit;
}
```

6. Save and close the plugin file.
7. Click on the **Settings** section of the administration menu.
8. Click on the **My Google Analytics** menu item to display the configuration page.
9. Change the value of one of the fields and click on the **Submit** button.
10. When the page refreshes, you will see that the values displayed reflect the values submitted.

How it works...

This recipe is the first to introduce an action hook that has a variable name. Instead of writing a specific action hook name when calling `add_action`, this hook name starts with the words `admin_post_` and is followed by the name of an action that it expects to match with a hidden form field. In this case, the action name is `save_ch2pho_options`. Going back to the previous recipe, you can see that this text is the same as the one that was placed in the hidden form field called `action`:

```
<input type="hidden" name="action"
value="save_ch2pho_options" />
```

When the configuration page form is submitted, it sends all data to the `admin-post.php` script, which checks for an `action` field and then sends the data that it received to the associated function, if present.

Once the processing function is executed, the calls to `current_user_can` and `check_admin_referer` are security measures where we check to see if the user who is currently logged in has administrative rights and if the nonce field that was part of the form is present. An error in these permission checks will result in a specific error message letting the user know that he does not have the rights to perform this action while the nonce check will display a vague error message to throw off potential hackers.



The rest of the function focuses on retrieving the current set of plugin options using the `get_option` function, processing the posted fields, and storing the updated values back in the site database. While using `foreach` loops might seem to be overkill to store two simple data fields, this approach can easily scale up to support large amounts of configuration fields.

The final step is a call to the `wp_redirect` function to send the browser back to the plugin options page after all data has been stored.

See also

- ▶ *Rendering the admin page contents using HTML recipe*

Displaying a confirmation message when options are saved

An important usability aspect of any user interface is to display an acknowledgement message when users have completed a task successfully. As you may have noticed in the previous recipe, WordPress does not provide any user feedback by default after configuration data has been saved to the options table.

This recipe explains how to display an acknowledgement message on the configuration page after the user has updated the plugin's configuration options.

Getting ready

You should have already followed the *Processing and storing plugin configuration data* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v6.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Modify the call to `wp_redirect` at the end of the `process_ch2pho_options` function from:

```
wp_redirect( add_query_arg( 'page',  
                            'ch2pho-my-google-analytics',  
                            admin_url( 'options-general.php' ) ) );
```

to:

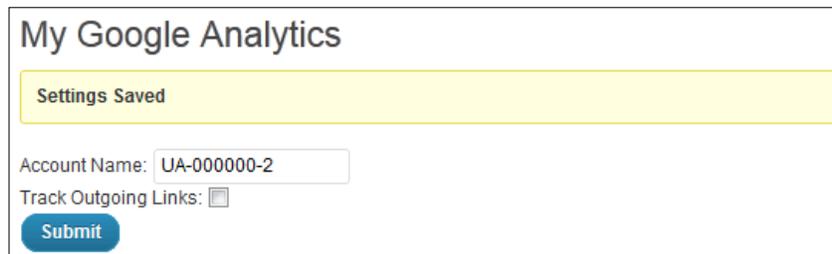
```
wp_redirect( add_query_arg(  
    array( 'page' => 'ch2pho-my-google-analytics',  
          'message' => '1' ),  
    admin_url( 'options-general.php' ) ) );
```

4. Add the following code (in bold) after the configuration page title, within the `ch2pho_config_page` function:

```
<h2>My Google Analytics</h2>
```

```
<?php if ( isset( $_GET['message'] )  
        && $_GET['message'] == '1' ) { ?>  
    <div id='message' class='updated fade'><p><strong>Settings  
    Saved</strong></p></div>  
<?php } ?>
```

5. Save and close the plugin file.
6. Click on the **Settings** section of the administration menu.
7. Click on the **My Google Analytics** menu item.
8. Change the value of one of the fields and click on the **Submit** button to see the newly created message indicating that the settings have been saved.



My Google Analytics

Settings Saved

Account Name:

Track Outgoing Links:

How it works...

When a redirection call is made, user-submitted fields and PHP variables do not carry forward to the target page. Therefore, we need to use another method, query arguments, to determine that a confirmation message should be displayed.

The first part of the recipe modifies the existing call to `wp_redirect` slightly to add a new query variable called `message`, set to a value of 1.

Once it receives this variable, the code responsible to render the options page can display a message, following the standard WordPress styling.

The same mechanism could be used to display different messages based on the outcome of the options storage. For example, if some fields need to receive data formatted a certain way, the `process_ch2pho_options` function could set the message value differently depending on the success or failure of the data processing operation.

See also

- ▶ *Processing and storing plugin configuration data recipe*

Adding custom help pages

As descriptive as field labels can be, a good plugin always needs to be accompanied by a set of documentation to allow users to quickly understand how to activate the plugin and perform the right steps to get the expected results. While a ReadMe file is often what developers first think to produce, users almost never read an external file or instructions on the official WordPress plugin page. They just install the plugin and try to figure it out by themselves.

To address this concern, version 3.3 of WordPress introduces the ability to create elaborate multi-section help pages right in the plugin's administration pages to allow users to quickly get answers to their questions. This recipe shows you how to register the appropriate callback function to add a help section to your plugin configuration page, containing multiple tabs of information.

Getting ready

You should have already followed the *Displaying a confirmation message when options are saved* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v7.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Find the `ch2pho_settings_menu` function in the existing code.
4. Modify the code to store the return value of the `add_options_page` function call to a variable:

```
$options_page = add_options_page(
    'My Google Analytics Configuration', 'My Google Analytics',
    'manage_options', 'ch2pho-my-google-analytics',
    'ch2pho_config_page' );
```

5. Add the following block of code to the `ch2pho_settings_menu` function to register an action that will be called when the plugin's options page is loaded:

```
if ( $options_page )
    add_action( 'load-' . $options_page,
        'ch2pho_help_tabs' );
```

6. Add the following code to implement the newly declared `ch2pho_help_tabs` function:

```
function ch2pho_help_tabs() {
    $screen = get_current_screen();
    $screen->add_help_tab( array(
        'id'          => 'ch2pho-plugin-help-instructions',
        'title'       => 'Instructions',
        'callback'    => 'ch2pho_plugin_help_instructions'
    ) );

    $screen->add_help_tab( array(
        'id'          => 'ch2pho-plugin-help-faq',
        'title'       => 'FAQ',
        'callback'    => 'ch2pho_plugin_help_faq',
    ) );
}
```

```

    $screen->set_help_sidebar( '<p>This is the sidebar
        content</p>' );
}

```

7. Add the following code section to provide an implementation for the `ch2pho_plugin_help_instructions` function:


```

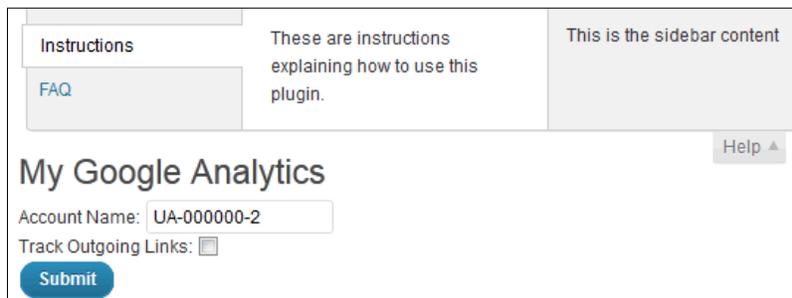
function ch2pho_plugin_help_instructions() { ?>
    <p>These are instructions explaining how to use this
        plugin.</p>
<?php }

```
8. Add the following code section to provide an implementation for the `ch2pho_plugin_help_faq` function:


```

function ch2pho_plugin_help_faq() { ?>
    <p>These are the most frequently asked questions on the use of
        this plugin.</p>
<?php }

```
9. Save and close the plugin file.
10. Click on the **Settings** section of the administration menu.
11. Click on the **My Google Analytics** menu to display the plugin configuration page. You will now see a **Help** tab appear in the top-right corner of the page.
12. Click on the **Help** tab to see all of the help content that was added to the plugin.



How it works...

As first discussed in the *Processing and storing plugin configuration data* recipe, some WordPress action hooks have names that contain a variable element that allows the plugin developer to get code executed when a specific page is rendered or when data from a specific form is submitted. In this example, the `load-<pagename>` hook is used to register a function that gets executed when a specific administration page is accessed by the user.

Once the callback occurs, the function's code retrieves a reference to the WordPress screen object, which contains data about the screen that is currently displayed along with a number of utility functions to manipulate and add content to the page. The code from the recipe then proceeds to register functions to render the content of two sections in the **Help** tab using the `add_help_tab` function.

The `add_help_tab` function is a little different from the functions that we have seen before, expecting a single array of options as its parameter. These options indicate a unique identifier for the menu section, which becomes the label displayed on each tab, and the name of the function that will render the tab contents. It is also possible to replace the callback argument with a parameter called `content`, which would directly contain the HTML code intended to be displayed in the **Help** tab. With this information, WordPress is able to integrate the provided HTML code when rendering the options page interface, including all of the necessary wrapper code to make the **Help** tab open and close, as well as allow the user to switch between the different sections.

The other function used in this recipe, `set_help_sidebar`, is even simpler than `add_help_tab`, with a single argument indicating the HTML content to be displayed on the right-hand side of the help section.

See also

- ▶ *Rendering the admin page contents using HTML recipe*

Rendering the admin page contents using the Settings API

Starting from version 2.7, WordPress introduced a set of functions referred to as the Settings API that can be used to automate the creation of complex configuration pages. While the work required to put this rendering technique in place is a bit overkill for plugins that only have a handful of options, it can definitely be useful if you are dealing with tens or hundreds of configuration fields, simplifying the task of writing out HTML code for every single item to a single function call. It also automates the task of processing and storing plugin configuration data.

This recipe explains how to specify the contents of a configuration page using the Settings API and how to provide rendering functions for the most commonly used types of form field used in configuration pages. It uses the same set of configuration options as other recipes in this chapter to show how the two techniques compare.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch3-settings-api`.
3. Navigate to this directory and create a new text file called `ch3-settings-api.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 3 - Settings API`.
5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add PHP code.
6. Add a PHP constant to specify an internal version number that will be used throughout the plugin code:

```
define( "VERSION", "1.1" );
```

7. Add the following line of code to register a function that will be called when WordPress activates the plugin:

```
register_activation_hook( __FILE__,
                        'ch3sapi_set_default_options' );
```

8. Add the following code section to provide an implementation for the `ch3sapi_set_default_options` function to set default plugin options:

```
function ch3sapi_set_default_options() {
    if ( get_option( 'ch3sapi_options' ) === false ) {
        $new_options['ga_account_name'] = "UA-000000-0";
        $new_options['track_outgoing_links'] = false;
        $new_options['version'] = VERSION;
        add_option( 'ch3sapi_options', $new_options );
    }
}
```

9. Add the following registration function to associate a callback with the `admin_init` action hook:

```
add_action( 'admin_init', 'ch3sapi_admin_init' );
```

10. Add an implementation for the `ch3sapi_admin_init` function, creating the settings group for the plugin and defining its contents:

```
function ch3sapi_admin_init() {
    // Register a setting group with a validation function
    // so that post data handling is done automatically for us

    register_setting( 'ch3sapi_settings',
                    'ch3sapi_options', 'ch3sapi_validate_options' );

    // Add a new settings section within the group
```

```

    add_settings_section( 'ch3sapi_main_section',
        'Main Settings',
        'ch3sapi_main_setting_section_callback',
        'ch3sapi_settings_section' );

    // Add each field with its name and function to use for
    // our new settings, put them in our new section

    add_settings_field( 'ga_account_name', 'Account Name',
        'ch3sapi_display_text_field',
        'ch3sapi_settings_section',
        'ch3sapi_main_section',
        array( 'name' => 'ga_account_name' ) );

    add_settings_field( 'track_outgoing_links',
        'Track Outgoing Links',
        'ch3sapi_display_check_box',
        'ch3sapi_settings_section',
        'ch3sapi_main_section',
        array( 'name' => 'track_outgoing_links' ) );
}

```

11. Declare a body for the `ch3sapi_validate_options` function, which was declared when registering the settings in the previous section, to return the user input with one additional piece of information indicating the plugin version:

```

function ch3sapi_validate_options( $input ) {
    $input['version'] = VERSION;
    return $input;
}

```

12. Declare a body for the `ch3sapi_main_setting_section_callback` function, declared when the settings section was created:

```

function ch3sapi_main_setting_section_callback() { ?>
    <p>This is the main configuration section.</p>
<?php }

```

13. Provide an implementation for the `ch3sapi_display_text_field` function, declared when a text field was added to the settings section:

```

function ch3sapi_display_text_field( $data = array() ) {
    extract( $data );
    $options = get_option( 'ch3sapi_options' );
    ?>
    <input type="text" name="ch3sapi_options[<?php echo $name;
    ?>]" value="<?php echo esc_html( $options[$name] );
    ?>" /><br />
<?php }

```

14. Declare and define the `ch3sapi_display_check_box` function, declared when a checkbox was added to the settings section:

```
function ch3sapi_display_check_box( $data = array() ) {
    extract ( $data );
    $options = get_option( 'ch3sapi_options' );
    ?>
    <input type="checkbox"
        name="ch3sapi_options[<?php echo $name; ?>]"
        <?php if ( $options[$name] ) echo ' checked="checked"' ;
    ?>/>
<?php }
```

15. Add the following line of code to register a function that will be called when WordPress is preparing data to display the site's administration menu:

```
add_action( 'admin_menu', 'ch3sapi_settings_menu' );
```

16. Provide code for the implementation of the `ch3sapi_settings_menu` function:

```
function ch3sapi_settings_menu() {
    add_options_page( 'My Google Analytics Configuration',
        'My Google Analytics - Settings API', 'manage_options',
        'ch3sapi-my-google-analytics',
        'ch3sapi_config_page' );
}
```

17. Add a definition for the `ch3sapi_config_page` function, defined when the new options page was declared:

```
function ch3sapi_config_page() { ?>
    <div id="ch3sapi-general" class="wrap">
        <h2>My Google Analytics - Settings API</h2>

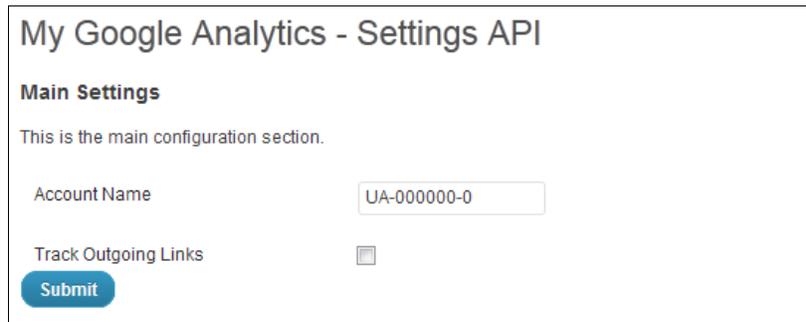
        <form name="ch3sapi_options_form_settings_api" method="post"
            action="options.php">

            <?php settings_fields( 'ch3sapi_settings' ); ?>
            <?php do_settings_sections( 'ch3sapi_settings_section' ); ?>

            <input type="submit" value="Submit" class="button-primary" />
        </form>
    </div>
<?php }
```

18. Save and close the plugin file.
19. Navigate to the **Plugins** menu of the administration area.

20. Activate your new plugin.
21. Navigate to the **Settings** menu and click on the **My Google Analytics – Settings API** menu item to see the configuration page for this plugin.



22. Make a change to the options and submit them to see that they are automatically handled by WordPress.

How it works...

The Settings API is an intricate series of callbacks that allow plugin developers to streamline the creation of administration pages and to automatically store user options to the site database without needing to write any code.

This self-contained plugin recipe starts with the creation of a new set of default options, to avoid inadvertently deleting options from previous recipes.

The code continues with registering a function to be called whenever admin pages are prepared for display using the `admin_init` action hook. Upon getting called, the callback function takes care of registering a new setting group, a setting section belonging to this group and two fields that will display the desired options within the section. As can be seen throughout this code, additional functions are registered to validate the user-submitted data, to display custom text at the beginning of the section and to display the two different types of fields required to capture and display user input.

Taking a closer look at each of the functions that were just used, the first function has three parameters that are as follows:

```
register_setting( $option_group, $option_name,  
                $sanitize_callback );
```

Within these parameters, the first option is a unique identifier for the settings group, the second is the name of the options array that will be used to store configuration data in the site database, while the third is the name of a callback function that will receive user input for validation.

Moving on to the second function used in this example, `add_settings_section`, the four parameters that it requires respectively indicate a unique identifier for the section, the title string that will be displayed when the section is rendered, a callback function that will be used to display a description for the section, and finally a page identifier that will be used to render all similar functions later within the plugin code.

```
add_settings_section( $id, $title, $callback, $page );
```

The third function of the Settings API that is used in this recipe, `add_settings_field`, is called multiple times to define the fields that make up each section.

```
add_settings_field( $id, $title, $callback, $page, $section,  
    $args );
```

Similar to the other functions, the first parameter is a unique identifier for the field, the second parameter is a label that will be displayed next to the field, and the third parameter is a callback function that will be executed to output the necessary HTML code to display the field. The next three parameters indicate the page that the field belongs to, the section that it is contained in, and an optional array of additional data to be sent to the callback function. As can be seen in the rest of this recipe, we are leveraging this optional additional data argument to send data to the field processing function to make them more generic.

When the configuration page is visited, the top-level form is created using regular HTML code, setting the action to `admin_post.php`. This script is responsible for automating the processing of user data. The rest of the form is quite simple since it gets generated by the `settings_fields` and `do_settings_sections` functions. When they are called, the setting group created earlier is rendered, followed by calls to the functions designed to draw all sections that it contains and all registered fields within these sections.

While the Settings API provides full control over the layout of the form fields themselves, its use dictates the general layout of the configuration page, creating a two-column table that contains the labels for each field in the first column and the code produced by the plugin's callback functions in the second one. As the functions for each type of field are called, they receive the array data that was associated with each of them and use it to retrieve current field values and to specify the name of each field to be stored back upon user input.

The last piece of the puzzle is the validation function that was registered when the setting group was first created. The purpose of this function is to allow the plugin developers to perform data type or content validation as user data is submitted through the form. In this specific example, we simply return the submitted data array, augmented with one piece of data containing the plugin version number, so that WordPress stores it automatically in the site database with the rest of the configuration data. This validation function can be used to create complex data type checks and validation rules, such as verifying date formats.

There's more...

While this recipe shows how to create rendering functions for two types of data fields, you may require other types of options for your plugin. The following are code examples which show how to handle most typical data types used in plugin options.

Rendering a drop-down list settings field

The first step to rendering a drop-down list is to provide the list of all possible options, along with the option name, in the optional field data array. Here is an example of the `add_settings_field` function call with such a list:

```
add_settings_field( 'Select_List', 'Select List',
    'ch3sapi_select_list',
    'ch3sapi_settings_section', 'ch3sapi_main_section',
    array( 'name' => 'Select_List',
        'choices' => array( 'First', 'Second', 'Third' ) ) );
```

With this information, we can provide an implementation for the `ch3sapi_select_list` function that will be able to render an HTML select element, using the `choices` array to populate it.

```
function ch3sapi_select_list( $data = array() ) {
    extract ( $data );
    $options = get_option( 'ch3sapi_options' );
    ?>
    <select name="ch3sapi_options[<?php echo $name; ?>]">
        <?php foreach( $choices as $item ) { ?>
            <option value="<?php echo $item; ?>"
                <?php selected( $options[$name] == $item ); ?>>
                <?php echo $item; ?></option>;
        <?php } ?>
    </select>
    <?php }
```

Rendering a text area settings field

Another common field type used in configuration pages is a multi-line text area. For this HTML construct, the `add_settings_field` function is identical to the text and checkbox examples shown in the recipe, while the field rendering code is as follows:

```
function ch3sapi_display_text_area( $data = array() ) {
    extract ( $data );
    $options = get_option( 'ch3sapi_options' );
    ?>
```

```

<textarea type="text"
  name="ch3sapi_options[<?php echo $name; ?>]"
  rows="5" cols="30">
  <?php echo esc_html ( $options[$name] ); ?></textarea>
<?php }

```

See also

- ▶ *Rendering the admin page contents using HTML recipe*

Accessing user settings from action and filter hooks

After creating a default set of values for our plugin's configuration and creating an interface to allow users to modify and update these values, we are now ready to start using these options when pages are rendered using our additional plugin functionality. Going back to the Google Analytics example created in the previous chapter, this recipe shows how to access the plugin options data using a familiar function to make the existing code much more flexible.

Getting ready

You should have already followed the *Adding custom help pages* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v8.php` to `ch2-page-header-output.php` before starting this recipe.

How to do it...

1. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-page-header-output.php` file in a text editor.
3. Modify the implementation of the `ch2pho_page_header_output` function to retrieve the plugin options array and use the stored value for the account number to embed it in the page header code. The new code sections are identified in bold:

```

function ch2pho_page_header_output() {
  $options = get_option( 'ch2pho_options' );
  ?>

  <script type="text/javascript">

  var gaJsHost = ( ( "https:" == document.location.protocol) ?
  "https://ssl." : "http://www." );

```

```

document.write( unescape( "%3Cscript src='" + gaJsHost +
    "google-analytics.com/ga.js'
    type='text/javascript'%3E%3C/script%3E" ) );

</script>

<script type="text/javascript">

try {
    var pageTracker = _gat._getTracker( "<?php echo
        $options['ga_account_name']; ?>" );
    pageTracker._trackPageview();
} catch( err ) {}

</script>

<?php }

```

4. Add code to check if outgoing code tracking should be done before registering an action hook to filter all post and page content, with the changes made identified in bold:

```

$options = get_option( 'ch2pho_options' );

if ( $options['track_outgoing_links'] == true ) {
    add_filter( 'the_content', 'ch2lfa_link_filter_analytics' );
}

```

5. Use the same check to determine if page footer code should be added to provide the JavaScript necessary for outgoing link tracking to occur, with the changes made identified in bold:

```

if ( $options['track_outgoing_links'] == true ) {
    add_action( 'wp_footer', 'ch2lfa_footer_analytics_code' );
}

```

6. Save and close the plugin file.
7. Visit the site and look at the page source to see that the previous UA-xxxxxxx-x has been replaced by UA-000000-0.

How it works...

As we saw earlier in this chapter when creating administrative pages, the `get_option` function can query the site's database and return the plugin configuration data that it contains. This data can be in the form of a single variable or an array of information. In this case, following the *Storing user settings using arrays* recipe found earlier in this chapter, an array was used and is accessed to use its values in the page output when header and footer action hooks are called and when page content is being filtered.

See also

- ▶ *Storing user settings using arrays* recipe

Formatting admin pages using meta boxes

As a plugin's administration page becomes longer and more complex, it becomes very important to divide its contents into multiple sections. While standard HTML headers or fieldset tags could be used for this task, they lack the usefulness and nice visual appearance of meta boxes. Meta boxes are the containers that show up in most default WordPress content editors, as well as on the main administration **Dashboard** page.

Beyond visually organizing content, meta boxes are very powerful since they allow site administrators to collapse configuration sections that they don't use, re-order sections based on their needs, and even hide elements that they don't use.

This recipe explains how to convert the HTML-based configuration page that was created earlier in this chapter to use the built-in meta box system.

Getting ready

You should have already followed the *Accessing user settings from action and filter hooks* recipe. Alternatively, you can get the resulting code from the downloaded code bundle. You should rename the file `ch2-page-header-output\ch2-page-header-output-v9.php` to `ch2-page-header-output.php` before starting the recipe.

How to do it...

1. Browse to the **Plugins** section of the administration section of your site and deactivate the **Chapter 2 – Page Header Output** plugin.
2. Navigate to the `ch2-page-header-output` folder of the WordPress plugin directory of your development installation.
3. Copy the file `ch2-page-header-output.php` to `ch2-page-header-output-metaboxes.php`.
4. Open the `ch2-page-header-output-metaboxes.php` file in a text editor.
5. Change the plugin name in the header from `Chapter 2 - Page Header Output` to `Chapter 2 - Page Header Output Meta Boxes`.

6. Right under the top plugin header comment, add a line of code to declare a global variable to hold the identifier for the options page:

```
global $options_page;
```

7. Find the `ch2pho_settings_menu` function in the existing code.
8. Add a line at the top of the function to point to the global options page variable:

```
global $options_page;
```

9. Find the `ch2pho_help_tabs` function within the plugin code.
10. Add the following block of code at the end of the function body to queue up scripts to be loaded when rendering the configuration page and to create meta boxes to be drawn on-screen:

```
global $options_page;

add_meta_box('ch2pho_general_meta_box',
    'General Settings', 'ch2pho_plugin_meta_box',
    $options_page, 'normal', 'core');

add_meta_box('ch2pho_second_meta_box',
    'Second Settings Section', 'ch2pho_second_meta_box',
    $options_page, 'normal', 'core');
```

11. Add a line of code at the end of the plugin code file to register a function to be called when administration page styles are placed in a queue:

```
add_action( 'admin_enqueue_scripts',
    'ch2pho_load_admin_scripts' );
```

12. Insert the following code segment to provide an implementation for the `ch2pho_load_admin_scripts` function:

```
function ch2pho_load_admin_scripts() {
    global $current_screen;
    global $options_page;

    if ( $current_screen->id == $options_page ) {
        wp_enqueue_script( 'common' );
        wp_enqueue_script( 'wp-lists' );
        wp_enqueue_script( 'postbox' );
    }
}
```

13. Create a new function to implement the `ch2pho_plugin_meta_box` function that was declared a few steps back. Notice that the body of the function is a direct copy and paste of the previous form code that was used to render the **Account Name** and **Track Outgoing Links** field.

```
function ch2pho_plugin_meta_box( $options ) { ?>
    Account Name: <input type="text" name="ga_account_name"
```

```

        value="<?php echo esc_html( $options['ga_account_name'] );
?>"/><br />

```

```

Track Outgoing Links <input type="checkbox"
name="track_outgoing_links" <?php if (
$options['track_outgoing_links'] ) echo '
checked="checked" ';
?>/><br />

```

```
<?php }
```

14. Add the following code to provide an implementation for the `ch2pho_second_meta_box` function, to display a second meta box. This second box will not have any real content. It will only be used to illustrate some of the meta box functionality.

```

function ch2pho_second_meta_box($options) { ?>
    <p>This is the content of the second metabox.</p>
<?php }

```

15. Find the code for the `ch2pho_config_page` function in your code and modify it as shown in the following code, where all new code segments are in bold. Delete the original code that rendered the `ga_account_name` and `track_outgoing_links` fields.

```

function ch2pho_config_page() {
    // Retrieve plugin configuration options from database
    $options = get_option( 'ch2pho_options' );
    global $options_page;
    ?>

    <div id="ch2pho-general" class="wrap">
    <h2>My Google Analytics</h2>

    <?php if (isset( $_GET['message'] )
        && $_GET['message'] == '1') { ?>
        <div id='message' class='updated fade'>
        <p><strong>Settings Saved</strong></p>
        </div>
    <?php } ?>

    <form action="admin-post.php" method="post">
    <input type="hidden" name="action"
        value="save_ch2pho_options" />

    <!-- Adding security through hidden referrer field -->
    <?php wp_nonce_field( 'ch2pho' ); ?>

    <!-- Security fields for meta box save processing -->

```

```
<?php wp_nonce_field( 'closedpostboxes',
    'closedpostboxesnonce', false ); ?>
<?php wp_nonce_field( 'meta-box-order',
    'meta-box-order-nonce', false ); ?>

<div id="poststuff" class="metabox-holder">
    <div id="post-body">
        <div id="post-body-content">
            <?php do_meta_boxes( $options_page, 'normal', $options) ;
            ?>
            <input type="submit" value="Submit"
                class="button-primary"/>
        </div>
    </div>
    <br class="clear"/>
</div>
</form>
</div>

<script type="text/javascript">

    //

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

        // close postboxes that should be closed
        $( '.if-js-closed' ) .removeClass( 'if-js-closed' ).
            addClass( 'closed' );

        // postboxes setup
        postboxes.add_postbox_toggles
            ( '&lt;?php echo $options_page; ?&gt;' );

    });

    //]]&gt;

&lt;/script&gt;

&lt;?php }</pre></div><div data-bbox="201 701 753 758" data-label="List-Group"><ol style="list-style-type: none;"><li>16. Save and close the plugin file.</li><li>17. Activate your new plugin.</li><li>18. Click on the <b>Settings</b> section in the left-hand navigation menu to expand it.</li></ol></div><div data-bbox="187 840 224 855" data-label="Page-Footer"><hr/><p>112</p></div>
```

19. Click on the **My Google Analytics** in the tree to display the re-designed administration page.

20. Drag-and-drop one of the meta boxes to re-order them.
21. Move the mouse cursor over the meta box to display its expansion controls in the top-right corner. Click on the arrow in the top-right corner to minimize the box.
22. Click on the **Screen Options** menu on the top-right corner to open a menu to control the visibility of all meta boxes.
23. Move to another section of the administration menu and come back to the **My Google Analytics** section to see that all changes made to the layout of the configuration page are retained.

How it works...

The setup of the meta box functionality is done in the `load-<pagename>` callback function by calling the `add_meta_box` function multiple times based on the desired number of boxes to be displayed on-screen.

The function takes a number of arguments, as shown:

```
add_meta_box( $id, $title, $callback, $page, $context, $priority,
    $callback_args );
```

Going over the parameters in this function, the first is a unique identifier for the meta box, while the second is the string that will be displayed as the title of the box itself and is also the name that will show up in the **Screen Options** configuration tab. The third parameter is the name of the function to be called to render the contents of the meta box. The fourth argument identifies the page where the meta boxes will be rendered. In this case, we use the value of the global variable `$options_page` for this parameter, to be sure that it will be assigned the correct page identifier.

The fifth parameter is an arbitrary name that indicates the name of a section where the box should be displayed. This name will be used when making a request to WordPress to render all meta boxes belonging to a specific section. The only requirement for this to work correctly is to use the same name when calling the `do_meta_boxes` function.

The sixth argument indicates the priority of the registered meta box within the section it belongs to, relative to other meta boxes. If all boxes have the same priority, the order in which the calls to the `add_meta_box` function were made will determine their original drawing order. Of course, as was seen in this recipe, this order can be overridden by the user through a simple drag-and-drop operation. The final parameter is optional and can be used to send information to the function that will render the meta box contents.

 While it is actually possible to call `add_meta_box` from other action hook callbacks, only meta boxes registered during a `load-<pagename>` callback will show up in the **Screen Options** list to allow the user to control their visibility. This may be the desired functionality in some cases to be sure that important boxes are always shown. It also provides a standard user experience for all users.

In addition to the calls to `add_meta_box`, we must make multiple calls to the `wp_enqueue_script` in the page load function to request for three JavaScript scripts to be loaded when our configuration page is rendered. These scripts provide the drag-and-drop, minimize, and hiding functionalities that were demonstrated at the end of the recipe, with only a few initialization calls needed to be done from our code through JavaScript functions.

Once the meta boxes have been created, the bulk of the work is done within the options page rendering function. As we can see in the modified code, the first thing that is done is to create new nonce fields. These unique numbers will be generated as hidden data in the page and will be used for authentication to save layout changes within the configuration page. Next, we create a number of `div` sections with specific `id` names that contain a nested call to the `do_meta_boxes` function. These `div` tags are used to ensure that the meta boxes are styled using the WordPress administration pages stylesheet.

Once called, the `do_meta_boxes` function takes care of drawing all of the meta boxes that were created for the given page (specified in the first argument) and given section (second argument). It also passes along any data specified in the third function argument to the functions associated with each box.

The remaining changes to the page rendering function is a block of JavaScript code that takes care of closing down any meta box section that was closed by the user during a previous visit to the page. It also assigns jQuery callbacks to the meta boxes so that any user interaction with them is saved to the site database by sending AJAX requests to the web server.

Last but not least, the meta box rendering functions are responsible for rendering the content inside each meta box. They can do this by outputting straight HTML. By passing along the complete options array to these functions, the code that is contained within them can be exactly the same as before to render the various options fields.

See also

- ▶ *Rendering the admin page contents using HTML* recipe

Splitting admin code from the main plugin file to optimize site performance

As mentioned in the previous chapter, the entire content of the main code file of a WordPress plugin gets evaluated every time any page is rendered on the site, whether it's a visitor-facing page or a backend administration page. This means that large amounts of PHP code can potentially be parsed on every iteration, wasting processing power on the site's server, even though some of this code will never be active when regular visitors are browsing the site.

A prime example of this waste is all of the code samples that we have been building in this chapter. While this code is extremely useful for site administrators, there is no sense in having the web server parse and validate that code when regular pages are displayed. For this reason, it is better to isolate this code in a separate file which will only be loaded and parsed when someone is visiting the site's dashboard. The following recipe shows how to isolate the less-frequently required code to a separate file and only load it when a user is visiting the site administration section.

Getting ready

You should have already followed the *Hiding items which users should not access from the default menu* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch3-hide-menu-item\ch3-hide-menu-item.php`) from the downloaded code bundle.

How to do it...

1. Navigate to the `ch3-hide-menu-item` folder of the WordPress plugin directory of your development installation.
2. Open the `ch3-hide-menu-item.php` file in a text editor.
3. Create a new PHP code file called `ch3-hide-menu-item-admin-functions.php` in the same directory.
4. Move the calls to the `add_action` function and the definition of the `ch3hmi_hide_menu_item` function to the new file, surrounded by the standard PHP open and close tags: `<?php` and `?>`.
5. Back in the main plugin code file (`ch3-hide-menu-item.php`), add code that will check if the current page being rendered is an administration page and proceed to load the administration functions if it is:

```
if ( is_admin() ) {  
    require plugin_dir_path( __FILE__ ) .  
        'ch3-hide-menu-item-admin-functions.php';  
}
```

6. Save and close the plugin file.
7. While the plugin will continue to work as it did before, the action hook registration code will only be processed when an administration page is displayed.

How it works...

As we briefly saw in the previous recipes, the `is_admin` function is used to quickly tell if the page currently being rendered is an administration page. If it is, our plugin code uses the standard PHP `include` function to load and execute the content of a separate file. In this case, the file is a second PHP file located in the plugin directory. To be flexible with regards to the location of the plugin files, we build a path to the file containing the administration functions using the WordPress `plugin_dir_path` function.

While the benefit of placing so little code in a separate file is minimal, this technique has a larger impact on performance when dealing with larger administration panels. In addition to not having to register an action hook on every page load, the PHP interpreter does not have to make sure that the syntax for the entire contents of that second file is valid when rendering front-facing pages.

See also

- ▶ *Hiding items which users should not access from the default menu* recipe

Storing stylesheet data in user settings

While most common plugin options are typically presented to users as simple textboxes, checkboxes, or drop-down lists, there are instances where more text needs to be stored for user settings. A good example of this are plugin-specific stylesheets, which allow users to change the visual appearance of plugin output. While loading a separate stylesheet file worked well in the *Loading a stylesheet to format plugin output* recipe in *Chapter 2, Plugin Framework Basics*, this approach did not give users a lot of liberty in changing these styling rules to work better with their site design since any changes that users make to the stylesheet will get overwritten when the plugin is updated using the WordPress automatic plugin upgrade process.

A solution to this problem is to store stylesheet data with the rest of the configuration options in the site database. This way, the information will remain intact when upgrades are performed. This recipe shows how to change the plugin created in the previous chapter to initialize the plugin options using an external file, how to create an administration panel to allow users to modify or reset the stylesheet, and how to use the new data to output the style information to the page header. Many of the lessons learned in this chapter will be put to use to create the final result.

Getting ready

You should have already followed the *Loading a stylesheet to format plugin output* recipe in the previous chapter, to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch2-private-item-text.php\ch2-private-item-text.php`) from the downloaded code bundle.

How to do it...

1. Navigate to the `ch2-private-item-text` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-private-item-text.php` file in a text editor.
3. Add the following lines of code after the existing functions and before the closing `?>` PHP command at the end of the file to implement an activation callback to initialize plugin options when it is installed or upgraded:

```
register_activation_hook( __FILE__,
                        'ch2pit_set_default_options_array' );

function ch2pit_set_default_options_array() {
    if ( get_option( 'ch2pit_options' ) === false ) {
        $stylesheet_location =
            plugin_dir_path( __FILE__ ) . 'stylesheet.css';
```

```

        $options['stylesheet'] =
            file_get_contents( $stylesheet_location );

        update_option( 'ch2pit_options', $options );
    }
}

```

4. Add the following code segment to register a function to be called when the menu is built to add an additional item under the **Settings** menu:

```

add_action( 'admin_menu', 'ch2pit_settings_menu' );

function ch2pit_settings_menu() {
    add_options_page('Private Item Text Configuration',
        'Private Item Text', 'manage_options',
        'ch2pit-private-item-text', 'ch2pit_config_page');
}

```

5. Insert the following code to provide an HTML implementation to render the options page:

```

function ch2pit_config_page() {
    // Retrieve plugin configuration options from database
    $options = get_option( 'ch2pit_options' );
    ?>

    <div id="ch2pit-general" class="wrap">
    <h2>Private Item Text</h2>

    <!-- Code to display confirmation messages when settings are
    saved or reset -->
    <?php if ( isset( $_GET['message'] )
        && $_GET['message'] == '1' ) { ?>
        <div id='message' class='updated fade'><p><strong>Settings
        Saved</strong></p></div>
    <?php } elseif ( isset( $_GET['message'] )
        && $_GET['message'] == '2' ) { ?>
        <div id='message' class='updated
        fade'><p><strong>Stylesheet reverted to
        original</strong></p></div>
    <?php } ?>

    <form name="ch2pit_options_form" method="post"
        action="admin-post.php">

    <input type="hidden" name="action"
        value="save_ch2pit_options" />
    <?php wp_nonce_field('ch2pit'); ?>

```

```

Stylesheet<br />
<textarea name="stylesheet" rows="10" cols="40" style="font-
    family:Consolas,Monaco,monospace"><?php echo esc_html
    ( $options['stylesheet'] ); ?></textarea><br />
<input type="submit" value="Submit" class="button-primary" />
<input type="submit" value="Reset" name="resetstyle"
class="button-primary" />
</form>
</div>
<?php }

```

6. Add the following block of code to register a function to be called when user options are saved, to provide an implementation for this function, and to provide a utility function to clean up redirection URLs:

```

add_action( 'admin_init', 'ch2pit_admin_init' );

function ch2pit_admin_init() {
    add_action( 'admin_post_save_ch2pit_options',
        'process_ch2pit_options' );
}

function process_ch2pit_options() {
    // Check that user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );

    // Check that nonce field created in configuration form
    // is present
    check_admin_referer( 'ch2pit' );

    // Retrieve original plugin options array
    $options = get_option( 'ch2pit_options' );

    if ( isset( $_POST['resetstyle'] ) ) {
        $stylesheet_location =
            plugins_dir_path( __FILE__ ) . 'stylesheet.css';
        $options['stylesheet'] =
            file_get_contents( $stylesheet_location );

        $message = 2;
    } else {
        // Cycle through all fields and store their values
        // in the options array
        foreach ( array( 'stylesheet' ) as $option_name ) {
            if ( isset( $_POST[$option_name] ) ) {
                $options[$option_name] = $_POST[$option_name];
            }
        }
    }
}

```

```
    }  
    $message = 1;  
  }  
  
  // Store updated options array to database  
  update_option( 'ch2pit_options', $options );  
  
  // Redirect the page to the configuration form that was  
  // processed  
  wp_redirect( add_query_arg( array(  
    'page' => 'ch2pit-private-item-text',  
    'message' => $message ),  
    admin_url( 'options-general.php' ) ) );  
  exit;  
}
```

7. Delete the call to the `add_action` function which associated the function `ch2pit_queue_stylesheet` with the `wp_enqueue_scripts` action hook, along with the `ch2pit_queue_stylesheet` function itself.
8. Add the following code to add the user-modifiable stylesheet code to the page header:

```
add_action( 'wp_head', 'ch2pit_page_header_output' );
```

```
function ch2pit_page_header_output() { ?>  
  <style type='text/css'>  
  <?php  
    $options = get_option( 'ch2pit_options' );  
    echo $options['stylesheet'];  
  ?>  
  </style>  
<?php }
```

9. Save and close the plugin file.
10. Deactivate, then activate the **Chapter 2 – Private Item Text** plugin from the administration interface.

11. Navigate to the **Settings** menu and select the **Private Item Text** submenu item to see the newly-created configuration panel, with options to submit changes to the stylesheet or reset it to its initial state:



12. Visit the website and look at the page source to see that the stylesheet data entered in the configuration page shows up in the HTML header:

```
<style type='text/css'>
.private {
  color: #CCCCCC;
}
</style>
```

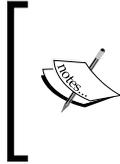
How it works...

Re-using many of the elements covered in this chapter, this recipe creates a simple yet effective configuration interface to allow users to make changes to the color that is used to highlight private text in posts, instead of this color being hardcoded in a plugin file.

That being said, this recipe does introduce two new concepts. The first is the initialization of the plugin options by reading data from a file instead of having all of that information stored in the PHP code. This technique is useful when dealing with an option that has a lot of content, such as a stylesheet.

The next element of interest is within the data processing function, where the code checks to see which button was pressed between the one to reset the stylesheet and the one to submit user changes to be stored in the site database. Based on the result, the processing code will either read back the initial stylesheet from the file or use the user posted data to update the configuration data.

Beyond these two new concepts, the other main change is to the code that was outputting header code referencing an external stylesheet file. In this new version, a change was made to echo the content of the stylesheet that is stored in the options table directly to the browser.



It should be noted that this recipe does not check to see if the user enters valid CSS code in the field before adding it to the page header, since verifying this would be too complex for now. A library such as CSSTidy (<http://csstidy.sourceforge.net/>) could be used to perform this task as desired.

See also

- ▶ *Creating a new enclosing shortcode recipe in Chapter 2, Plugin Framework Basics*
- ▶ *Loading a stylesheet to format plugin output recipe in Chapter 2, Plugin Framework Basics*

Managing multiple sets of user settings from a single admin page

Throughout this chapter, we have learned how to create configuration pages to manage single sets of configuration options for our plugins. In some cases, only being able to specify a single set of options will not be enough. For example, looking back at the YouTube Embed shortcode plugin that was created in the previous chapter, a single configuration panel would only allow users to specify one set of options, such as the desired video dimensions for the clips that will be displayed. A more flexible solution would be to allow users to specify multiple sets of configuration options, which could then be called up by using an extra shortcode parameter (for example, `[youtubevid id="R6Z7xceSLy4" optionid="2"]`).

While the first thought that might cross your mind to configure such a plugin is to create a multi-level menu item with submenus to store a number of different settings, this method would produce a very awkward interface for users to navigate. A better way is to use a single panel but give the user a way to select between multiple sets of options to be modified.

In this recipe, we will learn how to enhance the previously created YouTube video shortcode plugin to be able to control the embedded player size from the plugin configuration and to give the user the ability to specify multiple display sizes.

Getting ready

You should have already followed the *Creating a new shortcode with parameters* recipe in the previous chapter to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch2-youtube-embed\ch2-youtube-embed.php`) from the downloaded code bundle.

How to do it...

1. Navigate to the `ch2-youtube-embed` folder of the WordPress plugin directory of your development installation.
2. Open the `ch2-youtube-embed.php` file in a text editor.
3. Add the following lines of code after the existing functions and before the closing `?>` PHP command at the end of the file to implement an activation callback to initialize plugin options when it is installed or upgraded:

```
register_activation_hook( __FILE__,
    'ch2ye_set_default_options_array' );

function ch2ye_set_default_options_array() {
    if ( get_option( 'ch2ye_options_1' ) === false ) {
        ch2ye_create_setting( 1 );
    }
}

function ch2ye_create_setting( $option_id ) {
    $options['setting_name'] = 'Default';
    $options['width'] = 560;
    $options['height'] = 315;
    $options['show_suggestions'] = false;

    $option_name = 'ch2ye_options_' . $option_id;
    update_option( $option_name, $options );
}
```

4. Insert the following code segment to register a function to be called when the administration menu is put together. When this happens, the callback function adds an item to the **Settings** menu and specifies the function to be called to render the configuration page:

```
// Assign function to be called when admin menu is constructed
add_action( 'admin_menu', 'ch2ye_settings_menu' );

// Function to add item to Settings menu and
```

```
// specify function to display options page content
function ch2ye_settings_menu() {
    add_options_page( 'YouTube Embed Configuration',
        'YouTube Embed', 'manage_options',
        'ch2ye-youtube-embed',
        'ch2ye_config_page' );
}
```

5. Add the following code to implement the configuration page rendering function:

```
// Function to display options page content
function ch2ye_config_page() {

    // Retrieve plugin configuration options from database
    if ( isset( $_GET['option_id'] ) )
        $option_id = intval( $_GET['option_id'] );
    else
        $option_id = 1;

    $options = get_option( 'ch2ye_options_' . $option_id );

    if ( $options === false ) {
        ch2ye_create_setting( $option_id );
        $options = get_option( 'ch2ye_options_' . $option_id );
    }

    ?>

    <div id="ch2ye-general" class="wrap">
    <h2>YouTube Embed</h2>

    <!-- Display message when settings are saved -->
    <?php if ( isset( $_GET['message'] )
        && $_GET['message'] == '1' ) { ?>
    <div id='message' class='updated fade'><p><strong>Settings Saved
    </strong></p></div>
    <?php } ?>

    <!-- Option selector -->
    <div id="icon-themes" class="icon32"><br></div>
    <h2 class="nav-tab-wrapper">
    <?php for ( $counter = 1; $counter <= 5; $counter++ ) {
        $temp_option_name = "ch2ye_options_" . $counter;
        $temp_options = get_option( $temp_option_name );
        $class =
            ( $counter == $option_id ) ? ' nav-tab-active' : '';?>
```

```

    <a class="nav-tab<?php echo $class; ?>" href="<?php echo
add_query_arg( array( 'page' => 'ch2ye-youtube-embed', 'option_id'
=> $counter ), admin_url( 'options-general.php' ) ); ?>"><?php
echo $counter; ?><?php if ( $temp_options !== false ) echo ' ( ' .
$temp_options['setting_name'] . ')'; else echo ' (Empty)'; ?></a>
    <?php } ?>
</h2><br />

<!-- Main options form -->
<form name="ch2ye_options_form" method="post"
    action="admin-post.php">

    <input type="hidden" name="action" value="save_ch2ye_options"
/>
    <input type="hidden" name="option_id"
        value="<?php echo $option_id; ?>" />
    <?php wp_nonce_field( 'ch2ye' ); ?>

    <table>
        <tr>
            <td>Setting Name</td>
            <td><input type="text" name="setting_name"
value="<?php echo esc_html( $options['setting_name'] ); ?>"/></td>
        </tr>
        <tr>
            <td>Video Width</td>
            <td><input type="text" name="width" value="<?php echo
esc_html( $options['width'] ); ?>"/></td>
        </tr>
        <tr>
            <td>Video Height</td>
            <td><input type="text" name="height" value="<?php echo
esc_html( $options['height'] ); ?>"/></td>
        </tr>
        <tr>
            <td>Display suggestions after viewing</td>
            <td><input type="checkbox" name="show_suggestions"
<?php if ( $options['show_suggestions'] ) echo ' checked="checked"
'; ?>/></td>
        </tr>
    </table><br />
    <input type="submit" value="Submit" class="button-primary" />
</form>
</div>
<?php }

```

6. Add the following block of code to register a function that will process user options when submitted to the site:

```
add_action( 'admin_init', 'ch2ye_admin_init' );

function ch2ye_admin_init() {
    add_action( 'admin_post_save_ch2ye_options',
        'process_ch2ye_options' );
}
```

7. Add the following code to implement the `process_ch2ye_options` function, declared in the previous block of code, and to declare a utility function used to clean the redirection path:

```
// Function to process user data submission
function process_ch2ye_options() {
    // Check that user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );

    // Check that nonce field is present
    check_admin_referer( 'ch2ye' );

    // Check if option_id field was present
    if ( isset( $_POST['option_id'] ) )
        $option_id = $_POST['option_id'];
    else
        $option_id = 1;

    // Build option name and retrieve options
    $options_name = 'ch2ye_options_' . $option_id;
    $options = get_option( $options_name );

    // Cycle through all text fields and store their values
    foreach ( array( 'setting_name', 'width', 'height' ) as
        $param_name ) {
        if ( isset( $_POST[$param_name] ) ) {
            $options[$param_name] = $_POST[$param_name];
        }
    }

    // Cycle through all check box form fields and set
    // options array to true or false values
    foreach ( array( 'show_suggestions' ) as $param_name ) {
        if ( isset( $_POST[$param_name] ) ) {
            $options[$param_name] = true;
        }
    }
}
```

```

        } else {
            $options[$param_name] = false;
        }
    }

    // Store updated options array to database
    update_option( $options_name, $options );

    $cleanaddress =
        add_query_arg( array( 'message' => 1,
                               'option_id' => $option_id,
                               'page' => 'ch2ye-youtube-embed' ),
                    admin_url( 'options-general.php' ) );
    wp_redirect( $cleanaddress );
    exit;
}

```

8. Find the `ch2ye_youtube_embed_shortcode` function and modify it as follows to accept the new `option_id` parameter and load the plugin options to produce the desired output. The changes are identified in bold within the recipe:

```

function ch2ye_youtube_embed_shortcode( $atts ) {
    extract( shortcode_atts( array(
        'id' => '',
        'option_id' => ''
    ), $atts ) );

    if ( empty( $option_id ) ||
        intval( $option_id ) < 1 ||
        intval( $option_id ) > 5 ) {

        $option_id = 1;
    }

    $option_name = 'ch2ye_options_' . intval( $option_id );
    $options = get_option( $option_name );
    $output = '<iframe width="' . $options['width'];
    $output .= '" height="' . $options['height'];
    $output .= '" src="http://www.youtube.com/embed/' . $id;
    $output .= ( $options['show_suggestions'] == true
                ? " : "?rel=0" );
    $output .= '" frameborder="0" allowfullscreen></iframe>';
    return $output;
}

```

9. Save and close the plugin file.
10. **Deactivate** and **re-Activate** the **Chapter 2 – YouTube Embed** plugin from the administration interface to execute its activation function and create default settings.
11. Navigate to the **Settings** menu and select the **YouTube Embed** submenu item to see the newly-created configuration panel, with a first set of options being displayed and more sets of options accessible through the drop-down list shown at the top of the page.

YouTube Embed

1 (Default) 2 (Empty) 3 (Empty) 4 (Empty) 5 (Empty)

Setting Name

Video Width

Video Height

Display suggestions after viewing

Submit

12. To select the set of options to be used, add the parameter `optionid` to the **shortcode** used to display a YouTube video, as follows:

```
[youtubevid id="R6Z7xceSLy4" optionid="1"]
```

How it works...

This recipe shows how we can leverage `options` arrays to create multiple sets of options simply by creating the name of the `options` array on-the-fly. Instead of having a specific option name in the first parameter of the `get_option` function call, we create a string with an option ID. This ID is sent through as a URL parameter on the configuration page and as a hidden text field when processing the form data.

On initialization, the plugin only creates a single set of options, which is probably enough for most casual users of the plugin. Doing so will avoid cluttering the site database with useless options. When the user requests to view one of the empty option sets, the plugin creates a new set of options right before rendering the options page.

The rest of the code is very similar to the other examples that we saw in this chapter, since the way to access the array elements remains the same.

See also

- ▶ *Rendering the admin page contents using HTML recipe*

4

The Power of Custom Post Types

This chapter covers one of the most powerful features of WordPress, custom post types, through the following topics:

- ▶ Creating a custom post type
- ▶ Adding a new section to the custom post type editor
- ▶ Displaying single custom post type items using custom templates
- ▶ Creating an archive page for custom post types
- ▶ Displaying custom post type data in shortcodes
- ▶ Adding custom categories for custom post types
- ▶ Hiding the category editor from the custom post type editor
- ▶ Displaying additional columns in the custom post list page
- ▶ Adding filters for custom categories to the custom post list page
- ▶ Updating page title to include custom post data using plugin filters

Introduction

Building on its history of openness and ease-of-use, WordPress 3.0 reached new heights in customization with the introduction of custom post types.

Custom post types are new categories of items that are created by using the WordPress API and that appear in the WordPress administration interface as complete new sections, next to the default **Posts**, **Links**, and **Pages** sections. These custom items can be used to store any type of information, including events, bug reports, recipes, movie reviews, and many more.

When using custom post types to implement this kind of functionality, developers are able to take advantage of WordPress' internal content editing capabilities, including its powerful text editor and user-friendly media uploader. Custom post types also simplify data management for developers since all of the information related to these new entries is stored in the site database using the existing table structures. Finally, custom post types can leverage the established theme and template system to display the information that site administrators store in these new content types.

If you ever took a peek at the MySQL database behind a WordPress site, you know that posts, pages, attachments, revisions, and navigation menu items all share the same tables. In essence, all of these data elements are custom post types, with some of them using the standard text editor while others such as the navigation menus, have a custom management interface. Each of these types of items also has a different mechanism to be displayed on a site.

Using custom post types opens up endless possibilities to tailor the functionality of a WordPress installation and provide a very custom solution to end users without needing to invest a large amount of time re-inventing the wheel. This chapter covers all facets of creating custom post types through the creation of a **Book Review** system, including the creation of the new type of elements, displaying the newly stored information on the website, and customizing the environment to create an editor with unique capabilities.

Creating a custom post type

The initial creation of a custom post type is extremely easy. It only requires a single function to be called from an action hook callback. Once in place, a lot of functionality immediately becomes available to administrators and site visitors. This recipe shows how to create a new custom post type that will be used to store Book Reviews.

Getting ready

You should have access to a WordPress development environment, either on your local computer or a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch4-book-reviews`.
3. Navigate to this directory and create a new text file called `ch4-book-reviews.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 4 – Book Reviews`.

5. Add a few carriage returns before the `?>` characters that close the plugin header section to create space to add the PHP code.
6. Add the following line of code before the closing `?>` PHP command at the end of the file to register a function that will be executed during the initialization phase every time WordPress generates a page:

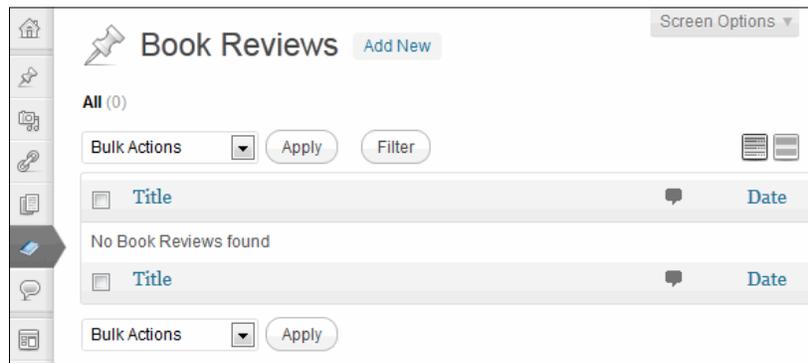
```
add_action( 'init', 'ch4_br_create_book_post_type' );
```

7. Add the following code block to provide an implementation for the `ch4_br_create_book_post_type` function:

```
function ch4_br_create_book_post_type() {
    register_post_type( 'book_reviews',
        array(
            'labels' => array(
                'name' => 'Book Reviews',
                'singular_name' => 'Book Review',
                'add_new' => 'Add New',
                'add_new_item' => 'Add New Book Review',
                'edit' => 'Edit',
                'edit_item' => 'Edit Book Review',
                'new_item' => 'New Book Review',
                'view' => 'View',
                'view_item' => 'View Book Review',
                'search_items' => 'Search Book Reviews',
                'not_found' => 'No Book Reviews found',
                'not_found_in_trash' =>
                    'No Book Reviews found in Trash',
                'parent' => 'Parent Book Review'
            ),
            'public' => true,
            'menu_position' => 20,
            'supports' =>
                array( 'title', 'editor', 'comments',
                    'thumbnail', 'custom-fields' ),
            'taxonomies' => array( '' ),
            'menu_icon' =>
                plugins_url( 'book-16x16.png', __FILE__ ),
            'has_archive' => true
        )
    );
}
```

8. Save and close the plugin file.

- Find and download a PNG format book icon measuring 16 x 16 pixels from a site such as IconArchive (<http://www.iconarchive.com>) and save it as `book-16x16.png` in the plugin directory.
- Navigate to the **Plugins** management page and activate the **Chapter 4 – Book Reviews** plugin.
- Click on the newly available **Book Reviews** menu item, located under the **Pages** section, to see the Book Review creation and management interface.



- Click on the **Add New** button, next to the section title, to display the Book Review editor featuring the complete WordPress text editor, the custom fields editor, comments control, publishing controls, and the featured image section.
- Fill in the new entry by specifying the Book Review title (for example, `WordPress Plugin Development Cookbook`) and a short description.
- Scroll to the **Custom Fields** section and type `book_author` as the **Name** of the field and `Yannick Lefebvre` as the **Value**. Click **Add Custom Field** to create a second field.

 If custom fields already exist in your WordPress installation (from previous recipes or other data entry), you will need to click on **Enter new** before being able to set the **Name** to `book_author`.

- Set the **Name** of the second field to `book_rating` and its **Value** to 5.
- Find and download a book cover image from websites such as Google Images (images.google.com) or Packt Publishing (www.packtpub.com).
- Click on the **Set featured image** link, located in the right-hand sidebar of the editing interface.
- Click on **Select Files** to pick the image that you downloaded to your computer and store it within the WordPress content folder.

19. Once the file is uploaded and WordPress displays information about it, scroll to the bottom of the media upload dialog and click on the **Use as featured image** link.
20. Click on the **Save all changes** button at the bottom of the page, then close the media uploader. You should see the image you uploaded appear in the **Featured Image** meta box. Click on the **Publish** button to save this first Book Review. Click the **View Book Review** button to see the newly-created content in your web browser.

How it works...

By making a call to the `register_post_type` function, the entire WordPress environment becomes aware of the existence of this new post type. This awareness includes the creation of a dedicated section to create and edit posts of this type and the ability to process web page requests for Book Reviews.

As mentioned in the beginning of this recipe, the function is quite simple to use and only requires two arguments:

```
register_post_type( $post_type, $args );
```

The first argument is a text string that indicates the name of the post type. Please note when choosing this name that it will be used as the default value for the permalinks of all items that use the new type, and that it should be unique enough to avoid potential conflicts with other plugins.

The second argument is an array of properties that specify the characteristics of the new post type and determine how this type will be edited.

In this specific example, the first element of the `properties` array is actually another array, which contains a number of labels. These labels indicate the text strings that should be displayed when managing items created under the new post type. For example, if we look at the screenshot in step 11, the message **No Book Reviews found** came directly from the definition of the `not_found` label in this array.

The second argument, named `public`, determines if the post type's administration interface should be shown to manage it and whether or not its contents should show up in search results. Next is the `menu_position` member of the configuration array, indicating the desired position of the new element in the administration menu. In this example, a value of 20 indicates that it should be displayed below the **Pages** menu item. Visit WordPress Codex (http://codex.wordpress.org/Function_Reference/register_post_type) for a full list of potential values for this parameter and their associated positions. The `supports` parameter is another array that indicates which parts of the content editor should be displayed for items that use the custom post type. In this case, we left out some sections such as `author`, `excerpt`, `trackbacks`, `revisions`, `page-attributes`, and `post-formats` as they were not desirable for Book Reviews.

The next few parameters in the configuration array indicate that we do not want to define custom taxonomies at this time and specify the path and name of the image file that should be displayed next to the post type's name in the administration menu. Finally, the last argument determines if WordPress should present an archive listing page for the new type when users visit the `/book-reviews` page on the site.

There are actually many other parameters that can be included in the configuration array to get more precise control over some aspects of the new custom post type. Please visit the WordPress Codex at the address mentioned previously to learn more about them.

There's more...

While the internal post type name is used by default to generate post permalinks, it can actually be overridden to create better-looking URLs.

Changing the custom post type permalinks slug

An optional member of the custom post type configuration is the `rewrite` parameter. It can be defined as follows:

```
'rewrite' => array( 'slug' => 'awesome_book_reviews' )
```

Although this may seem very simple, the permalinks won't change over immediately after making this change. To be more precise, the administration interface will show the new slug in the post editor, but trying to visit the page for the item with the new address will result in a 404 error (page not found). The solution to this problem is to rebuild the internal WordPress permalinks configuration using one of the following two techniques:

- ▶ Visit the **Permalinks** section of the **Settings** menu and click on the **Save Changes** button.
- ▶ Make calls to functions from the rewrite module to programmatically request for the configuration to be rebuilt. As this is not something that should be done every time WordPress displays a page, but would be too early to do when a plugin gets initialized or upgraded, a good place to call these functions would be within the plugin options storage function. You might even decide to give administrators the ability to specify their own slug. The code to reset the permalinks rules is as follows:

```
global $wp_rewrite;  
$wp_rewrite->flush_rules();
```

Adding a new section to the custom post type editor

While the custom post editor that has been put in place so far is functional, it is not the friendliest of user interfaces, especially with the custom fields section where users need to type or select the names of each field as they create new items. A cleaner approach is to create a custom interface using the **meta box** mechanism that we saw in the previous chapter to display all data associated with Book Reviews.

This recipe shows how to create a meta box that will be associated with a custom post type and how to save the information that is entered in that new interface.

Getting ready

You should have already followed the *Creating a custom post type* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v1.php`) from the code bundle downloaded from the Packt Publishing website (<http://www.packtpub.com/support>) and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when the administration interface is visited:

```
add_action( 'admin_init', 'ch4_br_admin_init' );
```

4. Add the following code section to provide an implementation for the `ch4_br_admin_init` function and register a meta box to be associated with the `book_reviews` post type:

```
function ch4_br_admin_init() {  
    add_meta_box( 'ch4_br_review_details_meta_box',  
                'Book Review Details',  
                'ch4_br_display_review_details_meta_box',  
                'book_reviews', 'normal', 'high' );  
}
```

5. Insert this function to implement the `ch4_br_display_review_details_meta_box` function and render the meta box contents:

```
function ch4_br_display_review_details_meta_box( $book_review ) {
    // Retrieve current author and rating based on review ID

    $book_author =
        esc_html( get_post_meta( $book_review->ID,
                                'book_author', true ) );

    $book_rating =
        intval( get_post_meta( $book_review->ID,
                               'book_rating', true ) );

    ?>
    <table>
        <tr>
            <td style="width: 100%">Book Author</td>
            <td><input type="text" size="80"
                name="book_review_author_name"
                value="<?php echo $book_author; ?>" /></td>
        </tr>
        <tr>
            <td style="width: 150px">Book Rating</td>
            <td>
                <select style="width: 100px"
                    name="book_review_rating">
                    <?php
                        // Generate all items of drop-down list
                        for ( $rating = 5; $rating >= 1; $rating -- ) {
                            ?>
                                <option value="<?php echo $rating; ?>"
                                    <?php echo selected( $rating,
                                                            $book_rating ); ?>>
                                    <?php echo $rating; ?> stars
                                <?php } ?>
                            </select>
                        </td>
                    </tr>
                </table>
            <?php } ?>

```

6. Add the following code segment to register a function that will be called when posts are saved to the database:

```
add_action( 'save_post',
            'ch4_br_add_book_review_fields', 10, 2 );
```

7. Add an implementation for the `ch4_br_add_book_review_fields` function, defined in the previous `add_action` call:

```
function ch4_br_add_book_review_fields( $book_review_id,
                                        $book_review ) {
    // Check post type for book reviews
    if ( $book_review->post_type == 'book_reviews' ) {
        // Store data in post meta table if present in post data
        if ( isset( $_POST['book_review_author_name'] ) &&
            $_POST['book_review_author_name'] != '' ) {
            update_post_meta( $book_review_id, 'book_author',
                              $_POST['book_review_author_name'] );
        }

        if ( isset( $_POST['book_review_rating'] ) &&
            $_POST['book_review_rating'] != '' ) {
            update_post_meta( $book_review_id, 'book_rating',
                              $_POST['book_review_rating'] );
        }
    }
}
```

8. Find the `ch4_br_create_book_post_type` function, where the new book type was originally created, and remove the `custom-fields` element from the `supports` array:


```
'supports' => array( 'title', 'editor', 'comments', 'thumbnail' ),
```
9. Save and close the plugin file.
10. Open the previously created Book Review to see the new **Book Review Details** meta box, containing a text field to specify the author and a drop-down list for the rating.

The screenshot shows a meta box titled "Book Review Details". It contains two input fields. The first is labeled "Book Author" and has a text input field with the value "Tessa Blakeley Silver". The second is labeled "Book Rating" and has a dropdown menu with "5 stars" selected.

How it works...

This recipe uses the WordPress built-in meta box system to create a clean interface that will allow users to manage fields specific to custom post types without having to use the cumbersome default **Custom Fields** editor. As we saw in *Chapter 3, User Settings and Administration Pages*, custom meta boxes can be created by using the `add_meta_box` function. In addition to declaring the meta box and associating it with the custom post type, `add_meta_box` defines a callback that is responsible for rendering the contents of the box.

The next section of the recipe implements the function that renders the meta box content. As we can see, this box receives an object variable that contains information about the Book Review that is being displayed in the post editor. Using this object, our code retrieves the post ID and uses it to query the site database for a book author and rating associated with the entry. Once the custom field data has been retrieved from the database, it can be used to render the author and rating fields on-screen. When new Book Reviews are created, both calls to `get_post_meta` will return an empty string, resulting in the display of an empty text field and the last entry in the drop-down list.

The last steps of this recipe take care of registering a function that will be called when posts of all types are saved or deleted by the site administrator. Since it will deal with all types of data, the saving callback must first check the type of the received post data. If it's a Book Review, the code proceeds to check if post data was received for the two Book Review meta box fields and stores the information in the post meta data table. In this recipe, the parameters for the `update_post_meta` function are similar to the `get_post_meta` function, except for the third argument which is used to specify the data to be stored.

One last detail that should be mentioned about this recipe is the use of the fourth parameter of the `add_action` function when associating a callback to the `save_post` action hook. This argument indicates that two arguments will be received by the registered callback. If this argument is not set, the callback function will never receive that second piece of data.

See also

- ▶ *Creating a custom post type recipe*
- ▶ *Formatting admin pages using meta boxes recipe in Chapter 3, User Settings and Administration Pages*

Displaying single custom post type items using custom templates

Custom post types straddle the line between plugin development and theme development since displaying any information that is specific to these new elements requires the creation of custom template files in the theme directory.

While it is not possible to create a solution that will work for all themes, a good practice that can be followed by plugin developers is to display the new data using the default WordPress theme. This way, users configuring the plugin on their site can see what needs to be done and adapt the code to their own theme.

This recipe shows how to create a template file to display all elements that we stored in the Book Review created in the previous recipe.

Getting ready

You should have already followed the *Adding to the custom post type editor* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v2.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when the administration interface is visited:

```
add_filter( 'template_include',
           'ch4_br_template_include', 1 );
```

4. Add the following code section to provide an implementation for the `ch4_br_template_include` function:

```
function ch4_br_template_include( $template_path ) {

    if ( get_post_type() == 'book_reviews' ) {
        if ( is_single() ) {
            // checks if the file exists in the theme first,
            // otherwise serve the file from the plugin
            if ( $theme_file = locate_template( array
                ( 'single-book_reviews.php' ) ) ) {
                $template_path = $theme_file;
            } else {
                $template_path = plugin_dir_path( __FILE__ ) .
                    '/single-book_reviews.php';
            }
        }
    }

    return $template_path;
}
```

5. Save and close the plugin file.
6. Create a new code file called `single-book_reviews.php` and open it in a code editor.

7. Add the following code in the file to create a template to display Book Reviews, including their custom fields:

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

<div id="primary">
    <div id="content" role="main">

        <!-- Cycle through all posts -->
        <?php while ( have_posts() ) : the_post(); ?>

        <article id="post-<?php the_ID(); ?>" <?php post_class(); ?>>

        <header class="entry-header">
        <!-- Display featured image in right-aligned floating div -->
        <div style="float: right; margin: 10px">
            <?php the_post_thumbnail( 'large' ); ?>
        </div>

        <!-- Display Title and Author Name -->
        <strong>Title: </strong><?php the_title(); ?><br />
        <strong>Author: </strong>
        <?php echo esc_html( get_post_meta( get_the_ID(),
            'book_author', true ) ); ?>
        <br />

        <!-- Display yellow stars based on rating -->
        <strong>Rating: </strong>
        <?php
            $nb_stars = intval( get_post_meta( get_the_ID(),
                'book_rating', true ) );
            for ( $star_counter = 1; $star_counter <= 5;
                $star_counter++ ) {
                if ( $star_counter <= $nb_stars ) {
                    echo '';
                } else {
                    echo '';
                }
            }
        ?>
        </header>

        <!-- Display book review contents -->
        <div class="entry-content"><?php the_content(); ?></div>
```

```

</article>

<!-- Display comment form -->
<?php comments_template( '', true ); ?>

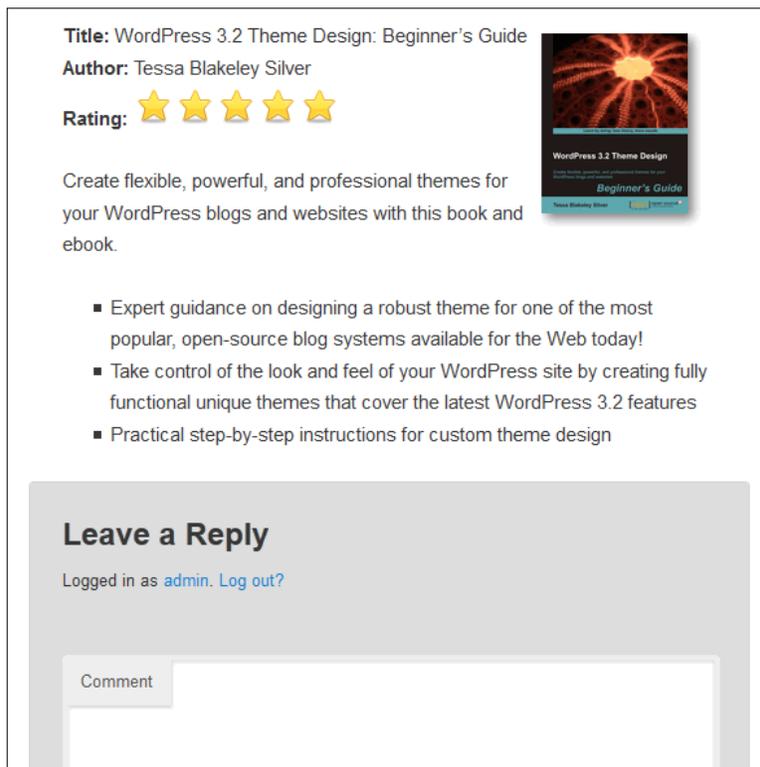
<?php endwhile; ?>

</div>
</div>

<?php get_footer(); ?>

```

8. Save and close the template file.
9. Find and download a PNG format pixel star icon measuring 32 x 32 pixels from a site such as IconArchive (<http://www.iconarchive.com>) and save it as `star-icon.png` in the plugin directory.
10. Create a black-and-white version of the star icon using any graphic processing tool and save it as `star-icon-grey.png`.
11. Go to the **Book Reviews** management page and click on the **View** link under the existing entry created in the previous recipe to see the content rendered using the new template.



How it works...

When rendering any web page, the default WordPress functionality is to search the current theme directory for an applicable template suitable for the content at hand. In the case of a single custom post type item such as a Book Review, it first looks for a single item template named `single-<post-type-name>.php`, where the latter part is the actual post type name. If it does not find this file, it defaults to the general single item template. In the first recipe of this chapter, the template that was used to show the Book Review was the default single item template, simply named `single.php`. To add better support for our new post type, this recipe associates a function with the `template_include` filter hook to change that behavior. More specifically, we use the `locate_template` function to check if the user provided a template for the `book_reviews` post type in the theme directory. If no template was found, we change the template path to load a template that we provide as part of the plugin files. This gives users the flexibility to use our template, which is more specific than the generic single item template, or to provide their own.

The rest of the recipe creates a new default template for Book Reviews, following standard theme mechanics, starting with the display of the site header and followed with a PHP `while` loop that cycles through all posts to be displayed (a single one in the case of a specific Book Review). This is followed by a number of WordPress template functions, such as `the_title()` and `the_content()`, to display various elements of the current post item.

To round out the page layout, the template contains code that displays the featured image along with the book title, its author, and its rating before showing the main content of the review and a comment form.

Out of these elements, the author and rating use a new function that we have not encountered yet called `get_post_meta`. This function is used to retrieve data that was stored in the custom fields section of the post editor and has three parameters:

```
get_post_meta( $post_id, $field_name, $single );
```

The first parameter is the post ID, which can easily be retrieved using the `get_the_ID()` template function. This ID is used to identify the post to which the custom information is associated. The second argument is the custom field name, which should match the name specified when it is created in the post editor. The third and final argument indicates if the return value should be a single value or an array of values. If set to `false`, it will produce an array containing a single element even if the custom field only contains a single value. In most cases, it should be set to `true` to receive a single value that can be accessed directly.

See also

- ▶ *Creating a custom post type recipe*

Creating an archive page for custom post types

Similar to the Book Review single item template file created in the previous recipe, WordPress allows users to create a special template to display archive listings for custom post types. This recipe shows how to create an archive template file to display a list of all Book Reviews in a table layout.

Getting ready

You should have already followed the *Displaying single custom post type items using custom templates* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v3.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Visit the `/book_reviews` page on your WordPress development site to see the default archive layout.
2. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
3. Open the `ch4-book-reviews.php` file in a code editor.
4. Find the `ch4_br_template_include` function and add the following highlighted code:

```
function ch4_br_template_include( $template_path ) {

    if ( get_post_type() == 'book_reviews' ) {
        if ( is_single() ) {
            // checks if the file exists in the theme first,
            // otherwise serve the file from the plugin
            if ( $theme_file = locate_template( array
                ( 'single-book_reviews.php' ) ) ) {
                $template_path = $theme_file;
            } else {
                $template_path = plugin_dir_path( __FILE__ ) .
                    '/single-book_reviews.php';
            }
        }
        elseif ( is_archive() ) {
            if ( $theme_file = locate_template( array
                ( 'archive-book_reviews.php' ) ) ) {
                $template_path = $theme_file;
            } else {
```

```
        $template_path = plugin_dir_path( __FILE__ ) .
                        '/archive-book_reviews.php';
    }
}

return $template_path;
}
```

5. Save and close the plugin file.
6. Create a new code file called `archive-book_reviews.php` and open it in a code editor.
7. Add the following code in the file to provide a template that will cycle through Book Reviews and display their title and author in a table:

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

<section id="primary">
    <div id="content" role="main" style="width: 80%">

        <?php if ( have_posts() ) : ?>

            <header class="page-header">
                <h1 class="page-title">Book Reviews</h1>
            </header>

            <table>
                <!-- Display table headers -->
                <tr>
                    <th style="width: 450px"><strong>Title</strong></th>
                    <th><strong>Author</strong></th>
                </tr>

                <!-- Start the Loop -->
                <?php while ( have_posts() ) : the_post(); ?>

                    <!-- Display review title and author -->
                    <tr>
                        <td><a href="<?php the_permalink(); ?>">
                            <?php the_title(); ?></a></td>
                        <td><?php echo esc_html( get_post_meta( get_the_ID(),
                            'book_author', true ) ); ?></td>
                    </tr>

                <?php endwhile; ?>

            </table>
```

```

<!-- Display page navigation -->
<?php global $wp_query;
    if ( isset( $wp_query->max_num_pages )
        && $wp_query->max_num_pages > 1 ) { ?>
        <nav id="<?php echo $nav_id; ?>">
        <div class="nav-previous"><?php next_posts_link(
            '<span class="meta-nav">&larr;</span>'
            Older reviews'); ?></div>
        <div class="nav-next"><?php previous_posts_link(
            'Newer reviews <span class="
            meta-nav">&rarr;</span>' ); ?></div>
        </nav>
    <?php };
    endif; ?>
</div>
</section>

<?php get_footer(); ?>

```

8. Save and close the template file.
9. Refresh the archive page to see a new table-based archive layout.

BOOK REVIEWS	
Title	Author
Mastering phpMyAdmin 3.3.x for Effective MySQL Management	Marc Delisle
Firebug 1.5: Editing, Debugging, and Monitoring Web Pages	Chandan Luthra, Deepak Mittal
PHP jQuery Cookbook	Vijay Joshi
Learning jQuery	Jonathan Chaffer, Karl Swedberg
WordPress 3 Search Engine Optimization	Michael David
← Older reviews	

How it works...

Similar to single posts, the default WordPress functionality is to search through the current theme directory looking for a template designed to render the current post type's archive page, before resorting to the default archive template. Extending the code from the previous recipe, we override this mechanism by checking to see if the user provided an archive template file for Book Reviews in the theme folder and provide our own file if no template is found. The rest of the recipe creates a new default archive template, where we cycle through the post entries using a while loop and display them using a table layout.

We also output a navigation menu if there are more reviews to display than the maximum number configured in the **Reading configuration** section of the **Settings administrative** section. This is done by first getting access to the `global wp_query` object, which contains data about the query that is currently being executed to render the page contents. If the maximum number of pages is higher than 1, navigation menus are displayed using the `next_post_links` and `previous_post_links` functions. Both of these functions are sent a single argument in the form of a string indicating the desired text to be displayed for the navigation links.

This recipe also uses the `get_post_meta` function that was covered in the previous recipe to retrieve custom field data that was created in the post editor.

See also

- ▶ [Displaying single custom post type items using custom templates recipe](#)

Displaying custom post type data in shortcodes

While displaying a list of custom post type items can be done using the archive page as shown in the previous recipe, it might be desirable to display a list of these elements in other places such as a page or post, or perhaps even on the front page as part of the basic template. In any of these cases, a good solution to display these items is to create a shortcode that will display one or more posts in any of these site areas.

This recipe shows how to create a shortcode that will retrieve and display five Book Reviews at a time, with accompanying navigation links.

Getting ready

You should have already followed the *Creating an archive page for custom post types* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v4.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.

3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function that declares the new shortcode:

```
add_shortcode( 'book-review-list', 'ch4_br_book_review_list' );
```

4. Add the following code section to provide an implementation for the `ch4_br_book_review_list` function:

```
function ch4_br_book_review_list() {
    // Preparation of query array to retrieve 5 book reviews
    $query_params = array( 'post_type' => 'book_reviews',
                          'post_status' => 'publish',
                          'posts_per_page' => 5 );

    // Execution of post query
    $book_review_query = new WP_Query;
    $book_review_query->query( $query_params );

    // Check if any posts were returned by the query
    if ( $book_review_query->have_posts() ) {
        // Display posts in table layout
        $output = '<table>';

        $output .= '<tr><th style="width: 350px"><strong>';
        $output .= 'Title</strong></th>';
        $output .= '<th><strong>Author</strong></th></tr>';

        // Cycle through all items retrieved
        while ( $book_review_query->have_posts() ) {
            $book_review_query->the_post();

            $output .= '<tr><td><a href="' . post_permalink();
            $output .= '">';
            $output .= get_the_title( get_the_ID() ) . '</a></td>';
            $output .= '<td>';
            $output .= esc_html( get_post_meta
                               ( get_the_ID(), 'book_author', true ) );
            $output .= '</td></tr>';
        }

        $output .= '</table>';

        // Display page navigation links
        if ( $book_review_query->max_num_pages > 1 ) {
            $output .= '<nav id="nav-below">';
            $output .= '<div class="nav-previous">';
```

```
$output .= get_next_posts_link
( '<span class="meta-nav">&larr;</span>
  Older reviews',
  $book_review_query->max_num_pages );
$output .= '</div>';
$output .= '<div class="nav-next">';
$output .= get_previous_posts_link
( 'Newer reviews <span class="meta-nav">
  &rarr;</span>',
  $book_review_query->max_num_pages );
$output .= '</div>';
$output .= '</nav>';
}

// Reset post data query
wp_reset_postdata();
}

return $output;
}
```

5. Save the plugin file.
6. Create a new page and insert the shortcode [book-review-list].
7. **Publish** and **View** the page to see that a list of Book Reviews will be displayed in place of the shortcode.

Book Reviews List	
TITLE	AUTHOR
Mastering phpMyAdmin 3.3.x for Effective MySQL Management	Marc Delisle
Firebug 1.5: Editing, Debugging, and Monitoring Web Pages	Chandan Luthra, Deepak Mittal
PHP jQuery Cookbook	Vijay Joshi
Learning jQuery	Jonathan Chaffer, Karl Swedberg
WordPress 3 Search Engine Optimization	Michael David
← Older reviews	

8. If more than five Book Reviews exist in the system, click on the navigation links that are displayed. You will see that the URL in the browser address bar changes but the list of entries show the same first five items as before.

9. Back in the `ch4-book-reviews.php` file, add the following highlighted code near the top of the `ch4_br_book_review_list`, right after the line initializing the value of the `$query_params` variable:

```
// Preparation of query string to retrieve 5 book reviews
$query_params = array( 'post_type' => 'book_reviews',
                      'post_status' => 'publish',
                      'posts_per_page' => 5 );

// Retrieve page query variable, if present
$page_num = ( get_query_var( 'paged' ) ?
             get_query_var( 'paged' ) : 1 );

// If page number is higher than 1, add to query array
if ( $page_num != 1 )
    $query_params['paged'] = $page_num;
```

10. Save and close the plugin file. Refresh the page using the shortcode and use the navigation links to see that the list of items now changes properly.

How it works...

As we have seen in *Chapter 3, User Settings and Administration Pages*, shortcodes are text elements that can be inserted in any page and post that will be replaced with content generated by the plugin when they are found. The registered callback function must prepare the output and send it back as a return value at the end of its execution.

The first part of the `ch4_br_book_review_list` function takes care of preparing a query array to be passed to a new instance of the `WP_Query` class. This class allows developers to easily extract information from the site database's post table. In this example, the parameters that are being set in the query are the internal post type name (`post_type`), the status of the items that we want to display (`post_status`), and the number of items that should be retrieved at a time (`posts_per_page`).

Once the query string is in place, we create a global variable called `book_review_query` and assign to it a new instance of a `WP_Query` object. Once created, we initialize it using the query string that was just assembled. If posts are found by the object, we output HTML code to create a table and use a `while` loop to cycle through all items found and display their title and author using code similar to the previous two recipes.

As part of this recipe we have seen that if more entries exist for the custom post type than the value specified with the `posts_per_page` query argument, navigation controls are added under the table of entries but will not work correctly since we manually created the query string. To rectify the situation, we use the `get_query_var` function to see if a page number was requested. If that is the case, and the page number is not 1, we add that number to our query parameters.

There's more...

As mentioned in the beginning of this recipe, there may be instances where a list of custom post type items needs to be displayed as part of a theme template. The following section shows how to get shortcode content to be displayed as part of a template file.

do_shortcode function

The `do_shortcode` function can be called from any theme template file, for the front page or any other section of the site, to render content associated with a shortcode. It takes a single argument, the shortcode string, including any parameters. To display the content created in this recipe, we would simply need to call the following:

```
<?php echo do_shortcode( '[book-review-list]' ); ?>
```

Adding custom categories for custom post types

To keep items organized in a site, administrators often use the built-in WordPress categories and terms to identify similar items. Looking back at the **Book Review System** that we have been putting in place so far in this chapter, a type of categorization that would be helpful is a book type (for example, Science-Fiction, Documentary, Fiction, Poetry, and so on).

This recipe shows how to create a new category (known as a **taxonomy** in the WordPress backend) and associate it with the Book Review custom post type.

Getting ready

You should have already followed the *Displaying custom post type data in shortcodes* and *Displaying single custom post type items using custom templates* recipes to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v5.php` and `ch4-book-reviews\single-book_reviews-v1.php`) from the downloaded code bundle. You should then rename them as `ch4-book-reviews.php` and `single-book_reviews.php`, respectively.

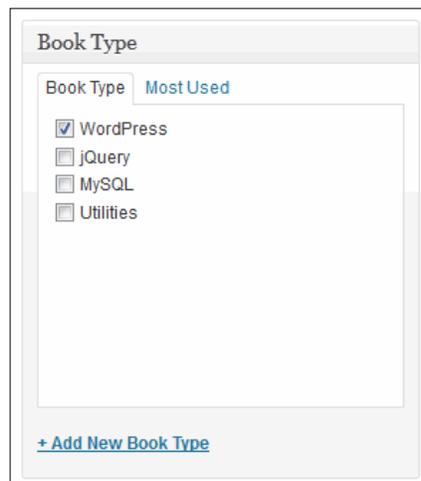
How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.

- Find the `ch4_br_create_book_post_type` function and add the following code after the existing call to `register_post_type` to create the new taxonomy:

```
register_taxonomy(  
    'book_reviews_book_type',  
    'book_reviews',  
    array(  
        'labels' => array(  
            'name' => 'Book Type',  
            'add_new_item' => 'Add New Book Type',  
            'new_item_name' => "New Book Type Name"  
        ),  
        'show_ui' => true,  
        'show_tagcloud' => false,  
        'hierarchical' => true  
    )  
);
```

- Save and close the plugin file.
- Open the previously created Book Reviews to see the newly added **Book Type** meta box on the right-hand side of the post editor.



- Click on the **+ Add New Book Type** link to create a new item and assign it as the current item's type. Click on the **Update** button on the top-right section of the post editor to save the review.

7. Look at the left-hand administration menu to see that a new menu item was added to manage book types, leading to an editor similar to the post and page category editor.



8. Open the `single-book_reviews.php` template file in a code editor.
9. Add the following code after the section displaying the author name to display the book type:

```
<strong>Type: </strong>
<?php
    $book_types =
        wp_get_post_terms( get_the_ID(),
                           'book_reviews_book_type' );

    if ( $book_types ) {
        $first_entry = true;
        for ( $i = 0; $i < count( $first_entry ); $i++ ) {
            if ( $first_entry == false )
                echo ', ';
            echo $book_types[$i]->name;
            $first_entry = false;
        }
    }
    else
        echo 'None Assigned';
?>
<br />
```

10. Save and close the template file.
11. Visit a Book Review page to see the book type displayed under the author's name.

How it works...

The `register_taxonomy` function is used to create a new type of category in WordPress and associate it to a post type. It has three parameters:

```
register_taxonomy( $taxonomy_name, $post_type, $options );
```

The first argument is a unique identifier for the taxonomy. The second parameter is the post type that it should be associated with, which should match the type declared with the `register_post_type` function. The third argument is an array of parameters that determine how the new taxonomy will behave.

In this example, we have set a few taxonomy options, including a first item called **labels** that contains an array of text strings that will be used in the interface when referring to the new taxonomy. We also specified a second element, called `show_ui`, that controls the display of the taxonomy meta box in the post editor and the presence of a link to access the taxonomy editor in the administration menu. Next is an option called `show_tagcloud`, which we set to `false` to avoid displaying a tag cloud of all taxonomy values. Finally, the last item in the options array is called `hierarchical`. When set to be `true`, taxonomy items will be able to have parent/child relationships and will be accessible as a list of checkboxes in the post editor. If set to `false`, all taxonomies are organized as a flat list and can be selected using an interface similar to the tag window in the post and page editor.

There are many more options available for the `register_taxonomy` function, as can be seen if you visit the WordPress Codex website (codex.wordpress.org), but the ones found here are the essential ones to define a basic taxonomy.

See also

- ▶ [Creating a custom post type recipe](#)

Hiding the category editor from the custom post type editor

As we have seen in the previous recipe, when we associated a new taxonomy with the Book Review custom post type, the `show_ui` option controls the visibility of the taxonomy assignment meta box and the admin menu link to the taxonomy editor. In some cases, it is desirable to give users access to the full taxonomy editor but only let editors choose from a controlled drop-down list when they create new entries in the custom post type editor.

This recipe shows how to hide the taxonomy interface and create a new menu item to still have access to the book type editor. It also shows how to update the custom post type meta box created in the previous recipe to assign a type to new Book Reviews and save this information in the site database.

Getting ready

You should have already followed the *Adding custom categories for custom post types* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v6.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Click on the **Book Reviews** top-level menu item in the administration menu.
2. Right-click on the **Book Type** submenu and select the browser option that will copy the link destination address (for example, `http://localhost/wp-admin/edit-tags.php?taxonomy=book_reviews_book_type&post_type=book_reviews` in a local development environment).
3. Paste the contents of the clipboard in a new text document for temporary storage.
4. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
5. Open the `ch4-book-reviews.php` file in a code editor.
6. Find the call to the `register_taxonomy` function within the `ch4_br_create_book_post_type` function and set the `show_ui` member of the configuration array to be false:

```
'show_ui' => false,
```

7. Save the plugin and refresh the administration page to see that the **Book Type** menu has disappeared.
8. Edit a Book Review to see that the **Book Type** taxonomy box is no longer displayed.
9. Add the following line of code after all the existing functions and before the closing `?>` PHP command at the end of the file to register a function that will be called when the administration section menu is being constructed:

```
add_action( 'admin_menu', 'ch4_br_add_book_type_item' );
```

10. Add the following code section to provide an implementation for the `ch4_br_add_book_type_item` function, using the previously stored Book Type editor address as the last item of the new submenu item array:

```
function ch4_br_add_book_type_item() {
    global $submenu;

    $submenu['edit.php?post_type=book_reviews'][501] =
        array( 'Book Type', 'manage_options',
            admin_url( '/edit-tags.php?
```

```

        taxonomy=book_reviews_book_type&
        post_type=book_reviews' ) );
    }

```

11. Save the plugin and refresh the administration page to see that the **Book Type** menu item is back.
12. Locate the `ch4_br_display_review_details_meta_box` function in the code and add the following code within the existing table rendering code to add a new row containing a drop-down selection box for the book type:

```

<tr>
    <td>Book Type</td>
    <td>
        <?php

            // Retrieve array of types assigned to post
            $assigned_types = wp_get_post_terms( $book_review->ID,
                                                'book_reviews_book_type' );

            // Retrieve array of all book types in system
            $book_types = get_terms( 'book_reviews_book_type',
                                    array( 'orderby' => 'name',
                                            'hide_empty' => 0 ) );

            if ( $book_types ) {
                echo '<select name="book_review_book_type"';
                echo ' style="width: 400px">';

                foreach ( $book_types as $book_type ) {
                    echo '<option value="' . $book_type->term_id;
                    echo '" ' . selected( $assigned_types[0]->term_id,
                                          $book_type->term_id ) . '>';
                    echo esc_html( $book_type->name );
                    echo '</option>';
                }

                echo '</select>';
            } ?>
        </td>
</tr>

```

- Find the `ch4_br_add_book_review_fields` function and add the following code segment, within the if statement checking to see if the post type is a Book Review, to save the selected book type to the site database upon submission of the post:

```
if ( isset( $_POST['book_review_book_type'] )
    && $_POST['book_review_book_type'] != '' ) {
    wp_set_post_terms( $book_review->ID,
                      $_POST['book_review_book_type'],
                      'book_reviews_book_type' );
}
```

- Save and close the plugin file.
- Open the previously created Book Review to see the updated **Book Review Details** meta box, containing a new drop-down list to specify the **Book Type**.



The screenshot shows a meta box titled "Book Review Details". It contains three input fields:

- Book Author:** A text input field containing "Tessa Blakeley Silver".
- Book Rating:** A dropdown menu with "5 stars" selected.
- Book Type:** A dropdown menu with "jQuery" selected.

How it works...

This recipe tricks WordPress permissions capabilities into giving users access to the book type editor by taking note of the original editor address and manually adding a link within the Book Reviews submenu that will take administrators to that page even if the link was hidden by the system. As can be seen in the recipe's code, this is done by using an action hook function attached to the `admin_menu` hook. When the function is executed, it adds an item to the submenu array for the **Book Reviews** menu in a slot with a high index to make sure that it does not conflict with other existing menu elements. The item contains the label to be displayed in the menu, the user capabilities required to view the item, and the URL of the taxonomy editor.

The recipe also makes use of three functions related to storing and retrieving taxonomy entries related to posts. The first, `wp_get_post_terms`, retrieves an array of terms associated with a post based on its ID and the name of the taxonomy. The second, `wp_set_post_terms`, assigns a term to a post based on its ID and the taxonomy name. Finally, `get_terms` retrieves an array of all terms in the taxonomy, ordered based on the query string found in the second argument.

See also

- ▶ [Adding custom categories for custom post types recipe](#)

Displaying additional columns in the custom post list page

After customizing the post editor to give content creators a tailored environment to create and edit custom post type entries, this recipe turns its efforts towards the Book Reviews management page where all entries for this type are listed. By default, custom post type listings are quite simple, only showing the title, publication date, and number of comments for each item. To make it easier to identify, sort, and find data in this management page, WordPress offers a number of customization capabilities, starting with the ability to change the columns that are displayed.

This recipe shows how to add and remove columns in the post management page, as well as make sorting in new columns possible.

Getting ready

You should have already followed the *Hiding the category editor from the custom post type editor* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v7.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when the Book Review listings page is being prepared:

```
add_filter( 'manage_edit-book_reviews_columns',  
           'ch4_br_add_columns' );
```

4. Add the following code section to provide an implementation for the `ch4_br_add_columns` function:

```
function ch4_br_add_columns( $columns ) {  
    $columns['book_reviews_author'] = 'Author';  
    $columns['book_reviews_rating'] = 'Rating';  
}
```

```
    $columns['book_reviews_type'] = 'Type';  
    unset( $columns['comments'] );  
    return $columns;  
}
```

5. Add the following line of code to assign a function to be called when columns data is getting retrieved for each row in the post listing:

```
add_action( 'manage_posts_custom_column',  
           'ch4_br_populate_columns' );
```

6. Insert the following code segment to provide an implementation for the `ch4_br_populate_columns` function:

```
function ch4_br_populate_columns( $column ) {  
    if ( 'book_reviews_author' == $column ) {  
        $book_author = esc_html(  
            get_post_meta( get_the_ID(),  
                          'book_author',  
                          true ) );  
        echo $book_author;  
    } elseif ( 'book_reviews_rating' == $column ) {  
        $book_rating = get_post_meta( get_the_ID(),  
                                      'book_rating', true );  
        echo $book_rating . ' stars';  
    } elseif ( 'book_reviews_type' == $column ) {  
        $book_types = wp_get_post_terms( get_the_ID(),  
                                         'book_reviews_book_type' );  
        if ( $book_types )  
            echo $book_types[0]->name;  
        else  
            echo 'None Assigned';  
    }  
}
```

7. Save the plugin file and navigate to the **Book Reviews** listing page to see that the list of columns has been altered and that data stored in the post custom fields is now displayed for each item in the list.
8. Back in the code editor, add the following code at the end of the plugin file, before the closing `?>` PHP tags, to register a function to be called when WordPress identifies columns that will be sortable for the Book Reviews custom post type:

```
add_filter( 'manage_edit-book_reviews_sortable_columns',  
           'ch4_br_author_column_sortable' );
```

9. Append the following code to provide an implementation for the `ch4_br_author_column_sortable` function:

```
function ch4_br_author_column_sortable( $columns ) {
    $columns['book_reviews_author'] = 'book_reviews_author';
    $columns['book_reviews_rating'] = 'book_reviews_rating';

    return $columns;
}
```

10. Add the following block of code to register a function that will be called when data is requested to display post lists:

```
add_filter( 'request', 'ch4_br_column_ordering' );
```

11. Insert the following code segment to implement the `ch4_br_column_ordering` function:

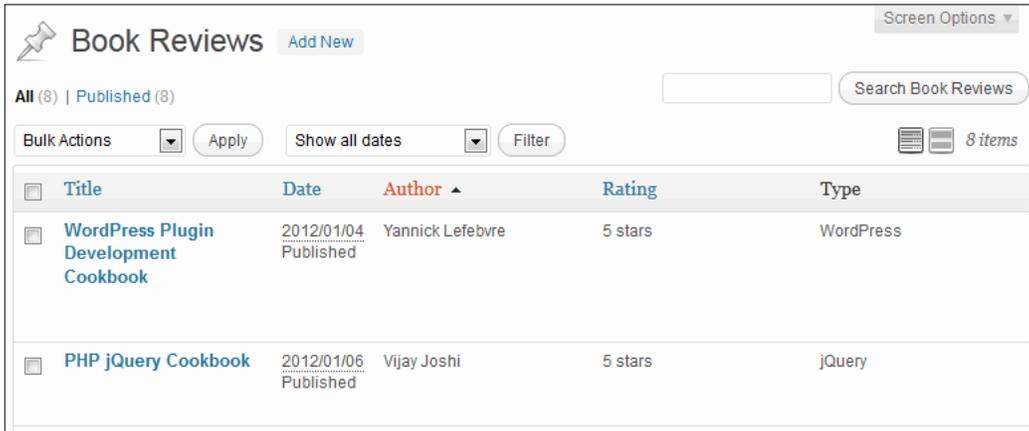
```
function ch4_br_column_ordering( $vars ) {
    if ( !is_admin() )
        return $vars;

    if ( isset( $vars['orderby'] ) &&
        'book_reviews_author' == $vars['orderby'] ) {
        $vars = array_merge( $vars, array(
            'meta_key' => 'book_author',
            'orderby' => 'meta_value' ) );
    } elseif ( isset( $vars['orderby'] ) &&
        'book_reviews_rating' == $vars['orderby'] ) {
        $vars = array_merge( $vars, array(
            'meta_key' => 'book_rating',
            'orderby' => 'meta_value_num' ) );
    }

    return $vars;
}
```

12. Save and close the plugin file.

- Refresh the **Book Reviews** listing to see that the **Author** and **Rating** column headers are links that can be clicked to sort these columns.



How it works...

Customizing the post listings page requires an intricate mix of action and filter hooks to achieve the final goal. The first function we registered is associated with the variable filter name `manage_edit-<post_type>_columns`, where `<post_type>` is replaced with the internal post type name. When the registered function is called, it receives the default column list that will be shown when listing Book Reviews entries as an argument. Using this data, it proceeds to add three columns for `author`, `rating`, and `type` and removes the `comments` column from the array. Once finished, it returns the modified array.

The second part of the recipe registers the function that will be responsible for populating the new columns. Since this function gets called when any custom post type column is rendered, the code checks which column is currently requested before echoing the requested data to the browser. The function makes calls to `get_the_ID()` to get the index of the currently displayed row and to be able to find its associated data using `get_post_meta` and `wp_get_post_terms`.

At this point in the recipe, the new columns are visible in the Book Reviews management page and data is displayed for each of them. The purpose of the rest of the recipe is to make the `author` and `rating` columns sortable. This is done by first registering a function with the variable filter name `manage_edit-<post_type>_sortable_columns`, where `<post_type>` is replaced with the post type name. When the function is executed, it adds two items to the array of columns that will be sorted. This takes care of making the column headers links that can be clicked for sorting, associated with the appropriate URLs.

The last function that gets registered is associated with the request filter and takes care of adding elements to the query array, based on the variables that came through in the query URL.

The final result allows administrators to easily reorder **Book Reviews** based on these two columns that can be sorted, as well as see information about each entry's type.

See also

- ▶ *Adding custom categories for custom post types* recipe

Adding filters for custom categories to the custom post list page

A second customization method for the custom post listings is to create a drop-down box that will allow administrators to only display items that belong to a single category at a time. This can help significantly reduce the number of entries that are shown to quickly find the desired entry.

This recipe shows how to add a filter mechanism based on the Book Review type to the listings page.

Getting ready

You should have already followed the *Displaying additional columns in the custom post list page* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v8.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when WordPress is preparing the filter drop-down boxes for the post listings:

```
add_action( 'restrict_manage_posts',  
            'ch4_br_book_type_filter_list' );
```

4. Add the following code section to provide an implementation for the `ch4_br_book_type_filter_list` function:

```
function ch4_br_book_type_filter_list() {
    $screen = get_current_screen();
    global $wp_query;
    if ( $screen->post_type == 'book_reviews' ) {
        wp_dropdown_categories(array(
            'show_option_all' => 'Show All Book Types',
            'taxonomy'        => 'book_reviews_book_type',
            'name'            => 'book_reviews_book_type',
            'orderby'        => 'name',
            'selected'        =>
            ( isset( $wp_query->query['book_reviews_book_type'] ) ?
              $wp_query->query['book_reviews_book_type'] : '' ),
            'hierarchical'   => false,
            'depth'          => 3,
            'show_count'     => false,
            'hide_empty'     => true,
        ));
    }
}
```

5. Insert the following line of code to register a function that will be called when the post display query is being prepared:

```
add_filter( 'parse_query',
            'ch4_br_perform_book_type_filtering' );
```

6. Implement the `ch4_br_perform_book_type_filtering` function with the following code segment:

```
function ch4_br_perform_book_type_filtering( $query ) {
    $qv = &$query->query_vars;

    if ( empty( $qv['book_reviews_book_type'] ) &&
          is_numeric( $qv['book_reviews_book_type'] ) ) {
        $term = get_term_by( 'id',
                            $qv['book_reviews_book_type'],
                            'book_reviews_book_type' );
        $qv['book_reviews_book_type'] = $term->slug;
    }
}
```

7. Save and close the plugin file.
8. Visit the **Book Reviews** listings to see the new drop-down to restrict what book types are displayed.



How it works...

This recipe starts by registering an action callback that will be executed when WordPress renders the various filter controls that are available for each post type listing. When the function is called, it retrieves a global variable to know the post type that is currently being displayed and determine if it should show the book type filter list. It also accesses the global post query variable to see if a book type filter is already in place and sets the correct drop-down list entry to be selected, if there is one.

The callback then proceeds to use the `wp_dropdown_categories` function to display a list of all of the taxonomy items registered for book types. This utility function expects to receive an array of parameters that determine which taxonomy list to display, the name of the drop-down list field name, and the label to be displayed for the option to show all types. This array should also contain a few parameters to determine the order in which the items should be displayed, specify the item to set as selected, indicate the maximum depth to show for hierarchical taxonomies, and determine whether or not item counts and empty items should be shown.

Once the new book type selection list is in place, selecting an entry and clicking on the **Filter** button triggers a refresh of the web page and leads to the second registered callback that was put in place being executed. The filter function receives the current WordPress post query object and starts by first getting a pointer to the query variables that are stored inside of the query object. With this in hand, it moves on to verify if a book type is part of the query variables and if it is numeric. If the result is positive, it replaces the numeric value with the textual name for the selected book type so that the query can take place.

Once all of this code is executed, users are able to quickly filter which book types should be displayed in the **Book Reviews** management page. They are also still able to use the column sorting mechanism implemented in the previous recipe.

See also

- ▶ [Adding custom categories for custom post types recipe](#)

Updating page title to include custom post data using plugin filters

A last customization touch that can be put in place to support our Book Reviews custom post type is to add custom information about the posts in the title bar when displaying them. For example, we could add the author's name next to the book title.

This recipe shows how to use the `wp_title` filter, first seen in *Chapter 2, Plugin Framework Basics*, to alter the post title for Book Reviews.

Getting ready

You should have already followed the *Adding filters for custom categories to the custom post list page* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v9.php`) from the downloaded code bundle and rename the file to `ch4-book-reviews.php`.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of the WordPress plugin directory of your development installation.
2. Open the `ch4-book-reviews.php` file in a code editor.
3. Add the following line of code after the existing functions and before the closing `?>` PHP command at the end of the file to register a function to be called when WordPress is preparing the text to be displayed in the browser's title bar:

```
add_filter( 'wp_title', 'ch4_br_format_book_review_title' );
```

4. Add the following code section to provide an implementation for the `ch4_br_format_book_review_title` function:

```
function ch4_br_format_book_review_title( $the_title ) {
    if ( get_post_type() == 'book_reviews' && is_single() ) {
        $book_author = esc_html( get_post_meta( get_the_ID(),
            'book_author', true ) );
        $the_title .= ' by ' . $book_author;
    }
    return $the_title;
}
```

5. Save and close the plugin file.
6. Visit a Book Review page. You will see that the book author is now displayed after the name in the title. However, you will also see that there is a vertical bar character between the title and the additional text elements.
7. Navigate to the `twentyeleven` folder of the WordPress themes directory of your development installation.
8. Open the `header.php` file in a code editor.
9. Replace the following code segment:

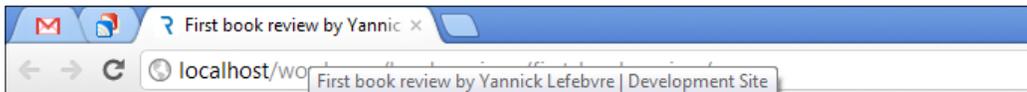
```
wp_title( '|', true, 'right' );

// Add the blog name.
bloginfo( 'name' );
```

with:

```
wp_title( '', true, 'right' );

// Add the blog name.
echo ' | ';
bloginfo( 'name' );
```
10. Save and close the template file.
11. Refresh the **Book Review** page to see the proper extended title displayed in the browser's title bar.



How it works...

As seen previously in *Chapter 2, Plugin Framework Basics*, the `wp_title` filter allows plugins to modify or completely replace the page title contents. In this case, the code of the function that we associated with the filter hook receives the title that WordPress intends to display as an argument. It then proceeds to check if the item that is being prepared for display is a Book Review and whether or not it is a single item. While the first condition is something obvious to check, the `is_single` verification is done to make sure that the code does not try to add a book author on the Book Reviews archive listing page.

While implementing this filter should have been enough to get the author to be displayed next to the book title in the title bar, we encounter a problem with the way that the `twentyeleven` theme puts together the page title. Namely, it uses the first parameter of the `wp_title` function to specify a separator to be displayed after the title. Unfortunately, this separator is added to the title before our filter function gets executed, even if we use the higher priority (1) when registering our filter hook callback. Therefore, we must modify the theme header file template to get the proper output. This may not be required for other themes based on how they put together the page title.

5

Customizing Post and Page Editors

In the previous chapters, we learned how to create custom plugin configuration panels and how to set up custom post types. With this knowledge in hand, we will now see how to customize the post and page editors as follows:

- ▶ Adding extra fields to the post editor using custom meta boxes
- ▶ Displaying custom post data in theme templates
- ▶ Hiding the Custom Field section in the post editor
- ▶ Extending the post editor to allow users to upload files directly

Introduction

Meta boxes are a very useful tool in the creation of WordPress plugins, as we saw in the previous two chapters. They were first used to organize large administration panels into manageable sections in *Chapter 3, User Settings and Administration Pages*, and then continued to be a key element in the creation of tailored interfaces to edit custom post types in *Chapter 4, The Power of Custom Post Types*.

This chapter explores how meta boxes can be used to augment the default post and page editors' capabilities. While WordPress posts and pages already offer a lot of functionality in a default installation, custom data entry fields go a long way in crafting a user experience that is much smoother than using the custom fields editor. These extra fields can be used to store anything. For example, they could be used to specify alternative language links for blog entries, or to assign a pop-up dialog to specific articles on a site.

Adding extra fields to the post editor using custom meta boxes

The WordPress post and page editors are organized in a series of collapsible sections with headers called meta boxes. While WordPress is mainly responsible for populating these containers with all of the appropriate elements, plugin developers can insert their own sections by registering user meta boxes.

To demonstrate this capability, this recipe shows how to add a custom meta box that will be used to display and capture information about the name and web address of the source materials used when writing a new post or page entry.

Getting ready

You should have access to a WordPress development environment, either on your local computer or a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch5-post-source-link`.
3. Navigate to the directory and create a text file called `ch5-post-source-link.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 5 - Post Source Link`.
5. Add the following line of code before the closing `?>` PHP command at the end of the file to register a function that will be executed when WordPress is preparing a list of meta boxes for all administration areas:

```
add_action( 'add_meta_boxes',  
            'ch5_psl_register_meta_box' );
```

6. Add the following code segment to provide an implementation for the `ch5_psl_register_meta_box` function:

```
function ch5_psl_register_meta_box() {  
    add_meta_box( 'ch5_psl_source_meta_box',  
                 'Post/Page Source', 'ch5_psl_source_meta_box',  
                 'post', 'normal' );  
  
    add_meta_box( 'ch5_psl_source_meta_box',  
                 'Post/Page Source', 'ch5_psl_source_meta_box',  
                 'page', 'normal' );  
}
```

7. Insert this code to provide an implementation for the `ch5_psl_source_meta_box` function, responsible for rendering the meta box contents:

```
function ch5_psl_source_meta_box( $post ) {

    // Retrieve current source name and address based on post ID
    $post_source_name = esc_html( get_post_meta( $post->ID,
                                                'post_source_name', true ) );
    $post_source_address = esc_html( get_post_meta( $post->ID,
                                                'post_source_address',
                                                true ) );

    ?>

    <!-- Display fields to enter source name and address -->
    <table>
        <tr>
            <td style="width: 100px">Source Name</td>
            <td>
                <input type="text" size="40"
                name="post_source_name" value="<?php echo
                $post_source_name; ?>" />
            </td>
        </tr>
        <tr>
            <td>Source Address</td>
            <td>
                <input type="text" size="40"
                name="post_source_address" value="<?php echo
                $post_source_address; ?>" />
            </td>
        </tr>
    </table>
<?php }
```

8. Insert the following block of code to register a function that will be called when any type of post is saved:

```
add_action( 'save_post', 'ch5_psl_save_source_data', 10, 2 );
```

9. Append the following code section to provide an implementation for the `ch5_psl_save_source_data` function:

```
function ch5_psl_save_source_data( $post_id = false,
                                   $post = false ) {
    // Check post type for posts or pages
```

```
if ( $post->post_type == 'post' ||
    $post->post_type == 'page' ) {

    // Store data in post meta table if present in post data
    if ( !empty( $_POST['post_source_name'] ) )
        update_post_meta( $post_id, 'post_source_name',
                          $_POST['post_source_name'] );

    if ( !empty( $_POST['post_source_address'] ) )
        update_post_meta( $post_id, 'post_source_address',
                          $_POST['post_source_address'] );

}
}
```

10. Save and close the plugin file.
11. Navigate to the **Plugins** management page and activate the **Chapter 5 – Post Source Link** plugin.
12. Go to the **Posts** section of the administration and click on one of the entries to open the post editor and see the new **Post/Page Source** meta box.



The image shows a screenshot of a meta box titled "Post/Page Source". It contains two text input fields. The first field is labeled "Source Name" and the second is labeled "Source Address". The meta box has a light gray border and a title bar.

How it works...

Every time an administrator or content manager visits the platform's backend, WordPress creates a number of meta boxes for all of its internal editors (post, pages, links, and so on) using the `add_meta_box` function that we have seen in the previous two chapters.

In this recipe, we use the same `add_meta_box` function twice to associate a new box to the page and post editors, with both calls registering the same callback function since we want the same functionality in both places. As WordPress stores posts and pages in the same database table, it will automatically make sure that all entries have unique IDs between both types of entries.

The other function that we have seen before is `get_post_meta`, which is used to retrieve custom metadata associated with post entries.

The content of the meta box itself is displayed using standard HTML. As this box will be part of a larger editor, there is no need to worry about declaring a form in this box.

Once the new dialog section is created, the next task is to put code in place to store data from the additional fields upon submission, through the use of the `save_post` action. Functions associated with this hook get called when posts of any type get saved. When executed, the associated function receives two parameters from WordPress that contain the ID of the post being saved and a copy of all data that has been processed to be saved so far. The callback also has access to all server post data received from the user.

As indicated in the previous chapter, it is important to set the fourth parameter of the `add_action` call—that is, `accepted_args`—for actions and filters that receive more than one argument. If it is not specified, these additional arguments will not be available to the receiving hook function.

Working with the assumption that the meta box was only added to the post and page editors, the code contained in the `ch5_psl_save_source_data` function first checks to see if the post type is set as a post or page. If it is one of these two types, it moves on to check for the presence of post data for the source name and address fields. If found, two calls to `update_post_meta` are made to store the new information in the site database, associated with the posts that it belongs to. Making a call to the update function will actually update the post custom field data if it exists or create it in the case of new post or page entries.

There's more...

While this recipe specifically adds a new section to the post and page editors, it may be desirable to make the new fields available to all post types, including custom ones.

Adding a new meta box to all post types (including custom ones)

This recipe made two function calls to register meta boxes with the post and page editors. This concept does not expand well to register a custom section with all post types since custom types created by other plugins are not known. Thankfully, there is an easy way to get an array of all post types that can be used to associate the new meta box to all post editors using a quick `foreach` loop.

The following code shows how the `ch5_psl_register_meta_box` function can be re-written to associate the new box with all post types:

```
function ch5_psl_register_meta_box() {
    $post_types = get_post_types( array(), 'objects' );
    foreach ( $post_types as $post_type ) {
        add_meta_box( 'ch5_psl_post_source_meta_box',
                    'Post/Page Source',
                    'ch5_psl_source_meta_box',
                    $post_type->name, 'normal' );
    }
}
```

Displaying custom post data in theme templates

If you have ever taken a look inside a theme's template files, you should be familiar with the functions that WordPress offers to weave content elements in the page layout. With custom field data associated to posts and pages, two options present themselves to users to display this information.

The first of these is to make calls to the `get_post_meta` function and display the resulting information right from the template code. This method assumes that the user understands all of the parameters to the function and knows what the names of all custom fields are. A cleaner alternative is to create a dedicated utility function that will display the new content with a single call from the template file.

This recipe explains how to create a function to display the source data associated to a post or page item as a clean link.

Getting ready

You should have already followed the *Adding extra fields to the post editor using custom meta boxes* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch5-post-source-link\ch5-post-source-link-v1.php`) from the code bundle you downloaded from the Packt Publishing website (<http://www.packtpub.com/support>) and rename the file to `ch5-post-source-link.php`.

How to do it...

1. Navigate to the `ch5-post-source-link` folder of the WordPress plugin directory of your development installation.
2. Open the `ch5-post-source-link.php` file in a code editor.
3. Add the following line of code before the closing `?>` PHP command at the end of the file to declare a function to be used to display a source link in a template:

```
function ch5_psl_display_source_link ( $before_link = '',
                                       $source_intro_text = '',
                                       $after_link = '',
                                       $post_id = '' ) {

    $post_id = ( !empty( $post_id ) ? $post_id :
                                                         get_the_ID() );

    $before_link = ( !empty( $before_link ) ? $before_link :
```

```

        '<div class="PostSource">' );
    $source_intro_text = ( !empty( $source_intro_text ) ?
        $source_intro_text : '<strong>Source:</strong>' );
    $after_link = ( !empty( $after_link ) ? $after_link :
        '</div>' );

    // Retrieve current source name and address based on post ID
    $post_source_name =
        get_post_meta( $post_id, 'post_source_name', true );
    $post_source_address =
        get_post_meta( $post_id, 'post_source_address', true );

    // Output information to browser
    if ( !empty( $post_source_name ) &&
        !empty( $post_source_address ) ) {
        echo $before_link;
        echo $source_intro_text;
        echo '<a href="' . esc_url( $post_source_address );
        echo '>' . $post_source_name;
        echo '</a>' . $after_link;
    }
}

```

4. Save and close the plugin file.
5. Navigate to the `twentyeleven` folder of the WordPress themes directory of your development installation.
6. Open the template file `content-single.php` in a code editor.
7. Add a call to the newly created `ch5_psl_display_source_link` function after the call to `the_content` and before the call to `wp_link_pages`:

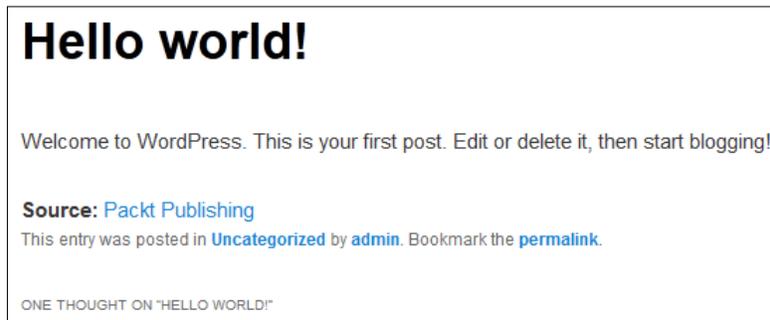
```
<?php the_content(); ?>
```

```
<?php ch5_psl_display_source_link(); ?>
```

```
<?php wp_link_pages( array( 'before' => '<div class="page-
link"><span>' . __( 'Pages:', 'twentyeleven' ) . '</span>',
'after' => '</div>' ) ); ?>
```

8. Save and close the template file.

9. Add source data to one of your site's posts and view it to see the new **Source** link displayed on the page.



How it works...

This recipe takes advantage of the fact that any function declared in a plugin code file can be accessed from any theme template file to create a data display utility function. Once in place, it allows users to easily display the captured article source data on any post or page template.

All parameters of this function are optional:

```
ch5_psl_display_source_link( $before_link,  
                             $source_intro_text,  
                             $after_link, $post_id );
```

The first three parameters are used for styling, allowing for HTML code to be displayed before and after the link, as well as displaying text in front of the link itself. The last parameter can be used to specify a post ID, in situations where this function is called outside a WordPress content processing loop. Unlike filter functions, this custom display function displays text directly to the browser as it is executed by using the echo function.

The `esc_url` function that is used in this code is a sanitization function that will make sure that only allowed HTML code is displayed when displaying the source link.

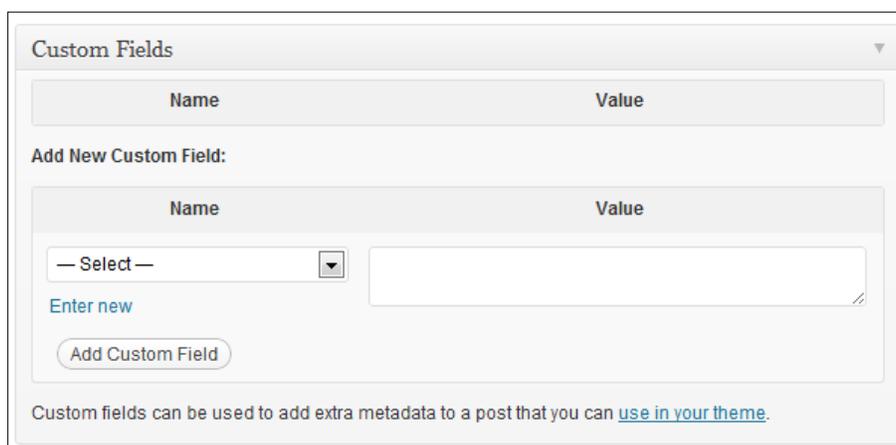
See also

- ▶ *Adding extra fields to the post editor using custom meta boxes recipe*

Hiding the Custom Field section in the post editor

After having full control over which meta boxes are shown when creating custom post type editor controls and putting together plugin configuration pages, things are a little different when it comes to altering the basic post and page editors. More specifically, instead of choosing which meta boxes to display, the editor sections created by WordPress need to be removed to tailor the user experience.

This recipe shows how to remove the **Custom Fields** meta boxes from the post and page editors.



Getting ready

You should have access to a WordPress development environment, either on your local computer or a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch5-hide-custom-fields`.
3. Navigate to this directory and create a new text file called `ch5-hide-custom-fields.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 5 - Hide Custom Fields`.

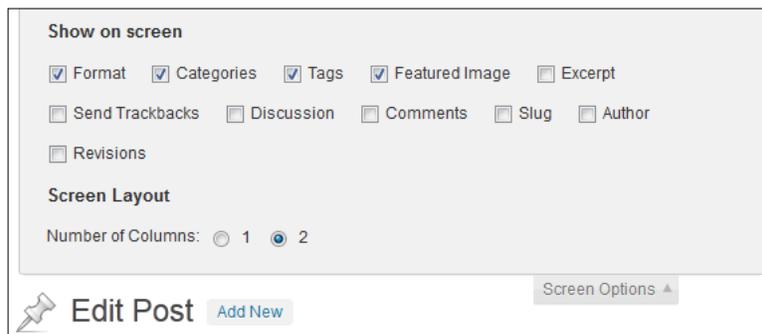
5. Add the following line of code before the closing `?>` PHP command at the end of the file to register a function that will be executed when WordPress is preparing a list of meta boxes for all administration areas:

```
add_action( 'add_meta_boxes',  
           'ch5_hcf_remove_custom_fields_metabox' );
```

6. Add the following code section to provide an implementation for the `ch5_hcf_remove_custom_fields_metabox` function:

```
function ch5_hcf_remove_custom_fields_metabox() {  
    remove_meta_box( 'postcustom', 'post', 'normal' );  
    remove_meta_box( 'postcustom', 'page', 'normal' );  
}
```

7. Save and close the plugin file.
8. Navigate to the **Plugins** management page and activate the **Chapter 5 – Hide Custom Fields** plugin.
9. Go to the **Posts** section of the administration and click on one of the entries to open the post editor. You will see that the **Custom Fields** section is no longer visible in the editor and does not show up in the **Screen Options** configuration tab either.



How it works...

This short recipe contains only a few lines of code, which register a function to be called when WordPress is preparing meta boxes for all administration sections, followed by the implementation of this function. The function itself makes two calls to the `remove_meta_box` function to remove the custom fields section from the post and page editors. This function requires three parameters:

```
remove_meta_box( $id, $page, $context );
```

The first argument is the meta box identifier that was used when it was first created. While you may not know where the creation code for a given meta box is located within the WordPress source code, a quick look at the box's `div id` in the page source from a browser reveals its name. In this case, the `id` is `postcustom`. The other two arguments indicate the name of the editor from which the meta box is to be removed and the context of the meta box (`normal`, `advanced`, or `side`).

Once the plugin is activated, the designated box disappears from the interface immediately.

Extending the post editor to allow users to upload files directly

WordPress offers a very complete media upload dialog. However, some projects might require users to be able to attach files right from the post editor. This recipe shows how to modify the post editor form to be able to attach files to articles and how to store these files once they have been uploaded. While any type of file could be attached using this technique, the code will be written to accept only items with a PDF file extension.

Getting ready

You should have access to a WordPress development environment, either on your local computer or a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch5-custom-file-uploader`.
3. Navigate to this directory and create a new text file called `ch5-custom-file-uploader.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 5 - Custom File Uploader`.
5. Add the following line of code before the closing `<?>` PHP command at the end of the file to register a function that will be executed when WordPress is rendering the HTML code at the beginning of the post editor form:

```
add_action( 'post_edit_form_tag', 'ch5_cfu_form_add enctype' );
```

6. Add the following code section to provide an implementation for the `ch5_cfu_form_add enctype` function:

```
function ch5_cfu_form_add enctype() {
    echo ' enctype="multipart/form-data"';
}
```

7. Insert the following line of code to register a function to be called when WordPress is preparing the meta boxes for all administration sections:

```
add_action( 'add_meta_boxes', 'ch5_cfu_register_meta_box' );
```

8. Add the following block of code to implement the `ch5_cfu_register_meta_box` function:

```
function ch5_cfu_register_meta_box() {  
    add_meta_box( 'ch5_cfu_upload_file', 'Upload File',  
                 'ch5_cfu_upload_meta_box', 'post', 'normal' );  
    add_meta_box( 'ch5_cfu_upload_file', 'Upload File',  
                 'ch5_cfu_upload_meta_box', 'page', 'normal' );  
}
```

9. Implement the function responsible for rendering the meta box contents, `ch5_cfu_upload_meta_box`, with the following code:

```
function ch5_cfu_upload_meta_box( $post ) {  
    ?>  
    <table>  
        <tr>  
            <td style="width: 150px">PDF Attachment</td>  
            <td>  
                <?php  
                    // Retrieve attachment data for post  
                    $attachment_data = get_post_meta( $post->ID,  
                                                    'attachdata',  
                                                    true );  
  
                    // Display post link if data is present  
                    if ( empty ( $attachment_data ) ) {  
                        echo 'No Attachment Present';  
                    } else {  
                        echo '<a href='';  
                        echo esc_url( $attachment_data['url'] );  
                        echo '>' . 'Download Attachment</a>';  
                    }  
                ?>  
            </td>  
        </tr>  
        <tr>  
            <td>Upload File</td>  
            <td><input name="uploadpdf" type="file" /></td>  
        </tr>  
    </table>
```

```

        <td>Delete File</td>
        <td><input type="submit" name="deleteattachment"
            class="button-primary" id="deleteattachment"
            value="Delete Attachment" /></td>
    </tr>
</table>
<?php }

```

10. Add the following line of code, which calls the `add_function` to register a callback that will be executed when post data is processed to be saved:

```
add_action( 'save_post', 'ch5_cfu_save_uploaded_file', 10, 2 );
```

11. Implement the `ch5_cfu_save_uploaded_file` with the following code:

```

function ch5_cfu_save_uploaded_file( $post_id = false,
                                     $post = false ) {
    if ( isset($_POST['deleteattachment']) ) {
        $attach_data = get_post_meta( $post_id, "attachdata",
                                     true );

        if ( $attach_data != "" ) {
            unlink( $attach_data['file'] );
            delete_post_meta( $post_id, 'attachdata' );
        }
    } elseif ( $post->post_type == 'post' ||
               $post->post_type == 'page' ) {

        // Look to see if file has been uploaded by user
        if( array_key_exists( 'uploadpdf', $_FILES ) &&
            !$_FILES['uploadpdf']['error'] ) {

            // Retrieve file type and store lower-case version
            $file_type_array = wp_check_filetype( basename(
                $_FILES['uploadpdf']['name'] ) );
            $pdf_file_type = strtolower( $file_type_array['ext']
            );

            // Display error message if file is not a PDF
            if ( $pdf_file_type != 'pdf' ) {
                wp_die( 'Only files of PDF type are allowed.' );
                exit;
            } else {
                // Send uploaded file data to upload directory
                $upload_return = wp_upload_bits(
                    $_FILES['uploadpdf']['name'], null,
                    file_get_contents(
                        $_FILES['uploadpdf']['tmp_name'] ) );
            }
        }
    }
}

```

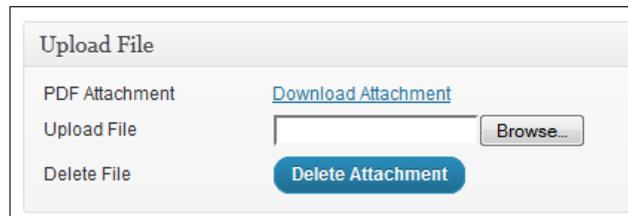


```

if ( empty( $attachment_data ) ) {
    echo 'No PDF Attachment Present';
} else {
    echo $before_link . '<a href="';
    echo esc_html( $attachment_data['url'] );
    echo '>' . $pdf_link_text;
    echo '</a>' . $after_link;
}
}

```

13. Save and close the plugin file.
14. Navigate to the **Plugins** management page and activate the **Chapter 5 – Custom File Uploader** plugin.
15. Edit any post on your site to see the new **Upload File** meta box and upload a PDF file to be associated with the item.



How it works...

The default WordPress post editor declares a simple form that does not have an encoding type defined and therefore can only accept regular text input. Fortunately, we have access to an action hook to register a callback function that will output additional code when the form gets created, allowing us to upload files. This callback is implemented in the first part of this recipe.

The next code section registers a meta box, as we have seen in many recipes so far, to display a new editor section that will display a link to an attached file if present, a file selection box to upload a new file, and a button to request for the attachment to be deleted.

Moving on to the function responsible for processing the post data, the recipe's code first checks if the user requested to delete a file associated with a post. If that is the case, it will proceed with the deletion of the file and remove the associated post meta data. In other circumstances, if the item's post type is a post or page, the plugin will proceed to search for a file uploaded by the user within the PHP global `$_FILES` array. This array contains information on any uploads that have been processed as part of a form's post data. If an entry is found, we use the `wp_check_filetype` function to retrieve information about the file type and proceed to convert the resulting extension to a lowercase string to make comparisons easier.

As this example only expects to receive PDF files, the code then checks to see if the file extension is correct to decide whether it will display an error message using the `wp_die` function or move the file from the web server's temp directory to the `wp-content/uploads` directory in WordPress using the `wp_upload_bits` function. In the latter case, it also stores the resulting file's path and URL in the post custom field table.

Once this is done, this recipe ends with the implementation of a utility function that can be used to easily display a link to the PDF attachment from any theme template file.

See also

- ▶ *Adding extra fields to the post editor using custom meta boxes recipe*

6

Accepting User Content Submissions

In this chapter, we will be focusing on giving visitors the ability to make submissions to a site. We will cover the following topics:

- ▶ Creating a client-side content submission form
- ▶ Saving user-submitted content in custom post types
- ▶ Sending e-mail notifications upon new submissions
- ▶ Implementing a captcha on user forms

Introduction

Giving users the ability to contribute content to a site is always a good way to engage the community and keep content fresh on a site. Going back to the book review system that was put in place in *Chapter 4, The Power of Custom Post Types*, this chapter explains how to allow visitors to add their own book reviews to a site.

Creating a client-side content submission form

The first step towards giving visitors the ability to contribute to a site is to present a form that they will be able to use to submit new content. This recipe shows how to create a shortcode that can easily be inserted on any WordPress page to render a simple form.

Getting ready

You should be running the final version of the Book Reviews plugin created in *Chapter 4, The Power of Custom Post Types*.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch6-book-review-user-submission` and open it.
3. Create a text file called `ch6-book-review-user-submission.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 6 – Book Review User Submission`.
5. Add the following line of code before the closing `?>` PHP command at the end of the file to declare a new shortcode and its associated function:

```
add_shortcode( 'submit-book-review',  
              'ch6_brus_book_review_form' );
```

6. Add the following code segment to provide an implementation for the `ch6_brus_book_review_form` function:

```
function ch6_brus_book_review_form() {  
    // make sure user is logged in  
    if ( !is_user_logged_in() ) {  
        echo '<p>You need to be a site member to be able to '  
        echo 'submit book reviews. Sign up to gain access!</p>';  
        return;  
    }  
    ?>  
  
<form method="post" id="addbookreview" action="">  
  
<!-- Nonce fields to verify visitor provenance -->  
<?php wp_nonce_field( 'add_review_form', 'br_user_form' ); ?>  
  
<table>  
    <tr>  
        <td>Book Title</td>  
        <td><input type="text" name="book_title" /></td>  
    </tr>  
    <tr>  
        <td>Book Author</td>  
        <td><input type="text" name="book_author" /></td>  
    </tr>  
    <tr>
```

```

        <td>Review</td>
        <td><textarea name="book_review_text"></textarea></td>
</tr>
<tr>
    <td>Rating</td>
    <td><select name="book_review_rating">
        <?php
        // Generate all rating items in drop-down list
        for ( $rating = 5; $rating >= 1; $rating-- ) { ?>
            <option value="<?php echo $rating; ?>">
                <?php echo $rating; ?> stars
        <?php } ?>
        </select>
    </td>
</tr>
<tr>
    <td>Book Type</td>
    <td>
        <?php
        // Retrieve array of all book types in system
        $book_types = get_terms( 'book_reviews_book_type',
            array( 'orderby' => 'name',
                'hide_empty' => 0 ) );

        // Check if book types were found
        if ( !is_wp_error( $book_types ) &&
            !empty( $book_types ) ) {

            echo '<select name="book_review_book_type">';

            // Display all book types
            foreach ( $book_types as $book_type ) {
                echo '<option value="' . $book_type->term_id;
                echo '>' . $book_type->name . '</option>';
            }

            echo '</select>';
        } ?>
    </td>
</tr>
</table>

<input type="submit" name="submit" value="Submit Review" />
</form>
<?php }

```

7. Save and close the plugin file.
8. Activate the **Chapter 6 – Book Review User Submission** plugin.
9. Create a new page and insert the newly created [submit-book-review] shortcode in the item's content.
10. Publish and view the page to see the form. Do not submit data as the associated processing script has not been implemented yet.



Submit your own Book Review!

Book Title

Book Author

Review

Rating

Book Type

How it works...

As seen in previous chapters, shortcodes are special blocks of text that can be easily added in any post or page to be replaced by content when they are found in pages visited by users. This recipe uses the `add_shortcode` function to declare a new shortcode that gets replaced with a review submission form.

The form itself is created using standard HTML and displays a number of text fields. It also uses a bit of PHP code to dynamically build the list of ratings and book types defined in the system. Finally, the form includes a PHP call to the `wp_nonce_field` function, which was previously seen when creating plugin configuration panels, to add hidden fields that will be used as a security measure in the associated data processing function.

When submitted, the form action will send visitor content to the page where the book review form is displayed. This new content will be intercepted and processed in the code that will be added in the next recipe.



It should be noted that trying to submit a book review at this time will result in the display of a blank page since we have not implemented a processing function for user data yet.

There's more...

While engaging visitors is important, users may not want to allow just anyone to post content to their site. An easy way to control the source of user submissions is to show the book review submission form only to those users who have registered accounts on the site.

Controlling access to client-side user form

By using the `is_user_logged_in` function, the form display function in this recipe can be quickly modified to display the form only to those users who are registered and logged into the site.

```
function ch6_brus_book_review_form() {
    // Check if user is logged in
    if ( !is_user_logged_in() ) {
        echo '<p>You need to be a site member to be able to';
        echo ' submit book reviews. Sign up to gain access!</p>';
        return;
    } ?>

    <!-- ... Previous form HTML display code goes here ... -->
}
```

See also

- ▶ *Creating a new simple shortcode recipe in Chapter 2, Plugin Framework Basics*

Saving user-submitted content in custom post types

When visitors click on the Submit button on the form created in the previous recipe, the target of the form is set to be the same page that is hosting the submission form. Since this page is not capable of handling form data, we must implement an action hook that intercepts this post data and sends it to a processing function that we will define.

This recipe shows how to implement a function responsible for processing user input.

Getting ready

You should be running the final version of the Book Reviews plugin created in *Chapter 4, The Power of Custom Post Types*, and should have already followed the *Creating a client-side content submission form* recipe. Alternatively, you can get the file from the code bundle and rename the file `ch6-book-review-user-submission-v1.php` to `ch6-book-review-user-submission.php`.

How to do it...

1. Navigate to the `ch6-book-review-user-submission` directory of the WordPress plugin folder of your development installation.
2. Open the file `ch6-book-review-user-submission.php` in a text editor.
3. Add the following line of code before the closing `?>` PHP command at the end of the file to register a function that will intercept user-submitted book reviews:

```
add_action( 'template_redirect',  
           'ch6_brus_match_new_book_reviews' );
```

4. Add the following block of code to implement the `ch6_brus_match_new_book_reviews` function:

```
function ch6_brus_match_new_book_reviews( $template ) {  
    if ( !empty( $_POST['ch6_brus_user_book_review'] ) ) {  
        ch6_brus_process_user_book_reviews();  
    } else {  
        return $template;  
    }  
}
```

5. Insert the following code to provide an implementation for the `ch6_brus_process_user_book_reviews`:

```
function ch6_brus_process_user_book_reviews() {  
    // Check that all required fields are present and non-empty  
    if ( wp_verify_nonce( $_POST['br_user_form'],  
                        'add_review_form' ) &&  
        !empty( $_POST['book_title'] ) &&  
        !empty( $_POST['book_author'] ) &&  
        !empty( $_POST['book_review_text'] ) &&  
        !empty( $_POST['book_review_book_type'] ) &&  
        !empty( $_POST['book_review_rating'] ) ) {  
        // Create array with received data  
        $new_book_review_data = array(  
            'post_status' => 'publish',
```

```

        'post_title' => $_POST['book_title'],
        'post_type' => 'book_reviews',
        'post_content' => $_POST['book_review_text'] );

// Insert new post in site database
// Store new post ID from return value in variable
$new_book_review_id =
    wp_insert_post( $new_book_review_data );

// Store book author and rating
add_post_meta( $new_book_review_id, 'book_author',
    wp_kses( $_POST['book_author'], array() ) );
add_post_meta( $new_book_review_id, 'book_rating',
    (int) $_POST['book_review_rating'] );

// Set book type on post
if ( term_exists( $_POST['book_review_book_type'],
    'book_reviews_book_type' ) ) {
    wp_set_post_terms( $new_book_review_id,
        $_POST['book_review_book_type'],
        'book_reviews_book_type' );
}
// Redirect browser to book review submission page
$redirectaddress =
    ( empty( $_POST['_wp_http_referer'] ) ? site_url() :
        $_POST['_wp_http_referer'] );
wp_redirect( add_query_arg( 'addreviewmessage', '1',
    $redirectaddress ) );
exit;
} else {
    // Display message if any required fields are missing
    $abortmessage = 'Some fields were left empty. Please ';
    $abortmessage .= 'go back and complete the form.';
    wp_die($abortmessage);
    exit;
}
}

```

6. In the original `ch6_brus_book_review_form` function, add the following code after the `wp_nonce_field` function call:

```

<?php if ( isset( $_GET['addreviewmessage'] )
    && $_GET['addreviewmessage'] == 1 ) { ?>

    <div style="margin: 8px;border: 1px solid #ddd;
        background-color: #fff0;">

```

```
        Thank for your submission!  
    </div>  
  
<?php } ?>  
  
<!-- Post variable to indicate user-submitted items -->  
<input type="hidden" name="ch6_brus_user_book_review" value="1" />
```

7. Save and close the plugin file.
8. Go back to the book review submission form, and submit a review to send all fields to the newly created processing function. After processing the new content, the script will return to the form, which will display a confirmation message.

How it works...

After sending all post data to the page containing the book review submission form in the previous recipe, the first few steps of this recipe assign a function to the `template_redirect` action hook to allow us to capture new book review content. This hook is executed early in the WordPress processing sequence. If found, we call the processing function that is defined in the rest of the recipe.

The first thing that is done in our processing function is to check if the proper hidden data field is found as part of the post data using the `wp_verify_nonce` function. If it is not present, indicating that someone may be trying to use the script without having used the front-end form, it will display an error message.

When we are sure that our data storage script is being legitimately called, we continue processing the actual data by first checking to see if all fields are present and are not empty. If that is not the case, we display an error message asking the user to go back and complete the form using the `wp_die` function.

If all fields have been received correctly, the recipe continues to process the incoming data by preparing an array of information that includes the newly submitted title and review text, along with a post status and the `book_reviews` post type name. The resulting array is sent to the `wp_insert_post` function to store the information. As we can see, `wp_insert_post` only requires a single parameter that is fulfilled using the array that we just created. While we only define four elements of that array, many more are available, which can be seen by consulting the WordPress Codex.

Now, calling `wp_insert_post` only takes care of storing some key data elements that belong in the post data. We must follow up this code with calls to `update_post_meta` and `wp_set_post_terms` to store the remaining user information to the site database.

Once all information is stored, we use a combination of the `wp_redirect` and `add_query_arg` functions to send the user back to the page where he submitted a book review, while making sure that only one instance of the `addreviewmessage` variable is in the target address.

Last, but not least, this recipe makes a small modification to the code that rendered the book review form to add a confirmation message that is shown to visitors when information is accepted by the plugin.

There's more...

In a world of spam bots and real people who are set on creating bogus content on any site, setting new book reviews to be immediately visible on the site might not be wise.

Moderating user-submitted content

Instead of setting a status of `publish` for new post entries, we can use a value of `draft` to make the new entry visible only in the back-end administration area. To give plugin users more flexibility, you could also give them a way to decide what method they prefer in a configuration panel.

See also

- ▶ *Creating a client-side content submission form* recipe
- ▶ *Processing and storing plugin configuration data* recipe in *Chapter 3, User Settings and Administration Pages*
- ▶ *Adding a new section to the custom post type editor* recipe in *Chapter 4, The Power of Custom Post Types*

Sending e-mail notifications upon new submissions

Just like WordPress sends out e-mail notifications to the administrator when new comments are posted, sending out e-mails when visitors post new book reviews allows site managers to review new content as it comes in and decide if they approve it to be published online.

This recipe shows how to prepare e-mail data and send it using the `wp_mail` function.

Getting ready

You should be running the final version of the Book Reviews plugin created in *Chapter 4, The Power of Custom Post Types*, and should have already followed the *Saving user-submitted data in custom post types* recipe (including changing the post status to `draft` as indicated in the *There's more...* section.) Alternatively, you can get the file from the code bundle and rename `ch6-book-review-user-submission-v2.php` to `ch6-book-review-user-submission.php`. Finally, you should have access to a WordPress installation on a hosted web server as e-mails are usually not sent when running it on a local installation. Be sure to have access to the e-mail account associated with the site administrator to see the resulting e-mail.

How to do it...

1. Navigate to the `ch6-book-review-user-submission` directory of the WordPress plugin folder of your development installation.
2. Open the file `ch6-book-review-user-submission.php` in a text editor.
3. Insert the following line of code to register a function to be called back when new posts are submitted:

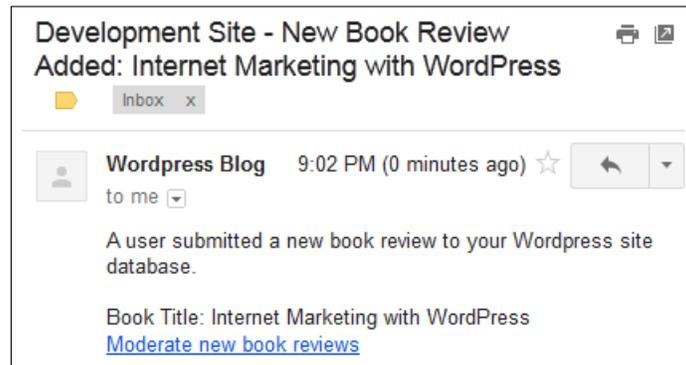
```
add_action( 'wp_insert_post', 'ch6_brus_send_email', 10, 2 );
```

4. Insert the following block of code to implement the `ch6_brus_send_email` function:

```
function ch6_brus_send_email( $post_id, $post ) {  
  
    // Only send e-mails for user-submitted book reviews  
    if ( isset( $_POST['ch6_brus_user_book_review'] ) &&  
        'book_reviews' == $post->post_type ) {  
  
        $admin_mail = get_option('admin_email');  
        $headers = 'Content-type: text/html';  
        $message = 'A user submitted a new book review to your '  
        $message .= 'Wordpress site database.<br /><br />';  
        $message .= 'Book Title: ' . $post->post_title ;  
        $message .= '<br />';  
        $message .= '<a href="';  
        $message .= add_query_arg( array(  
                                'post_status' => 'draft',  
                                'post_type' => 'book_reviews' ),  
                                admin_url( 'edit.php' ) );  
        $message .= '">Moderate new book reviews</a>';  
        $email_title = htmlspecialchars_decode( get_bloginfo(),  
            ENT_QUOTES ) . " - New Book Review Added: " .  
            htmlspecialchars( $_POST['book_title'] );  
        // Send e-mail  
        wp_mail( $admin_mail, $email_title, $message, $headers );  
    }  
}
```

5. Save and close the plugin file.

- Go back to the book review submission form and submit a book review. An e-mail will be sent to the address associated with the site administrator, containing some information from the new review.



How it works...

The `wp_mail` function can be used by any plugin to send out e-mail messages. It takes five arguments to define all elements of the outgoing message:

```
wp_mail( $destination, $subject, $message, $headers,
        $attachments );
```

The first three arguments are required and respectively define the e-mail address of the intended recipient, the title of the message, and its content. As we have seen in this recipe, the content is mainly specified using standard HTML syntax, while the target e-mail address is retrieved from the `options` table by using the `get_option` function. As for the title, it is built from a number of textual elements such as the blog title and book review title to create the final result.

The next parameter is optional and provides header information for the e-mail, with the most important piece of information in that section being the character set. The last parameter can optionally be used to specify the path of one or more files to be sent as e-mail attachments.

To make it easier for site administrators to manage new entries, part of the message body contains a link to the custom post management page of the WordPress site administration area to quickly display all unapproved entries (set as draft items).

See also

- ▶ *Creating a client-side content submission form recipe*

Implementing a captcha on user forms

A common security measure on website forms is to use **captcha** codes, where distorted letters and numbers are displayed with surrounding visual noise, to check that the person submitting data is not a spam robot. The form that we have been building to accept visitor-submitted book reviews could benefit from this type of technology.

This recipe shows how to download and incorporate a simple captcha library in the submission form created earlier and how to check whether the submitted information matches the content from the captcha image.

Getting ready

You should be running the final version of the **Book Reviews** plugin created in *Chapter 4, The Power of Custom Post Types*, and should have already followed the *Sending e-mail notifications upon new submissions* recipe. Alternatively, you can get the resulting file from the code bundle and rename it from `ch6-book-review-user-submission\ch6-book-review-user-submission-v3.php` to `ch6-book-review-user-submission.php` before starting the recipe.

How to do it...

1. Download the EasyCaptcha PHP script from <http://kestras.kuliukas.com/EasyCaptcha/>.
2. Navigate to the `ch6-book-review-user-submission` directory of the WordPress plugin folder of your development installation.
3. Extract the contents of the downloaded archive to the plugin directory to create the EasyCaptcha subdirectory.
4. Navigate to the EasyCaptcha directory and open the `easycaptcha.php` file in a code editor.
5. Change the 17-character string on line 21 to a different random string:

```
setcookie('Captcha',
    md5("HDBHAYYEJKPWIKJHDD".$captchaText.
        $_SERVER['REMOTE_ADDR'].$time).'.'.$time, null, '/');
```

6. Open the file named `ch6-book-review-user-submission.php` in a code editor.
7. Insert a new row in the form table to display a captcha and ask visitors to re-type the text shown in the image:

```
<tr>
    <td>Re-type the following text<br />
        
```

```

        </td>
        <td><input type="text" name="book_review_captcha" /></td>
    </tr>

```

8. Locate the `ch6_brus_process_user_book_reviews` function and add an extra item, highlighted in the following code, to the list of fields getting checked to make sure that they are not empty:

```

if ( wp_verify_nonce( $_POST['br_user_form'],
                    'addreview_form' ) &&
    !empty( $_POST['book_title'] ) &&
    !empty( $_POST['book_author'] ) &&
    !empty( $_POST['book_review_text'] ) &&
    !empty( $_POST['book_review_book_type'] ) &&
    !empty( $_POST['book_review_rating'] ) &&
    !empty( $_POST['book_review_captcha'] ) ) {

```

9. Insert the following code to check the captcha value, before the existing data processing code, making sure that the 17-character text string in the call to `md5` matches the previously modified string:

```

// Variable used to determine if submission is valid
$valid = false;

// Check if captcha text was entered
if ( empty( $_POST['book_review_captcha'] ) ) {
    $abortmessage = 'Captcha code is missing. Go back and ';
    $abortmessage .= 'provide the code.';
    wp_die( $abortmessage );
    exit;
} else {
    // Check if captcha cookie is set
    if ( isset( $_COOKIE['Captcha'] ) ) {
        list( $hash, $time ) =
            explode( '.', $_COOKIE['Captcha'] );

        // The code under the md5's first section needs to match
        // the code entered in easycaptcha.php

        if ( md5('HDBHAYYEJKPWIKJHDD' .
                $_REQUEST['book_review_captcha'] .
                $_SERVER['REMOTE_ADDR'] . $time ) != $hash ) {
            $abortmessage = 'Captcha code is wrong. Go back ';
            $abortmessage .= 'and try to get it right or reload ';
            $abortmessage .= 'to get a new captcha code.';

```

```

        wp_die( $abortmessage );
        exit;
    } elseif( ( time() - 5 * 60 ) > $time ) {
        $abortmessage = 'Captcha timed out. Please go back, ';
        $abortmessage .= 'reload the page and submit again.';
        wp_die( $abortmessage );
        exit;
    } else {
        // Set flag to accept and store user input
        $valid = true;
    }
} else {
    $abortmessage = 'No captcha cookie given. Make sure ';
    $abortmessage .= 'cookies are enabled.';
    wp_die( $abortmessage );
    exit;
}
}

```

10. Add an if statement, highlighted in the following code, around the previous content storage code to look at the value of the \$valid variable:

```

if ( $valid ) {
    // Create array with received data
    $new_book_review_data = array(
        'post_status' => 'draft',
        'post_title' => $_POST['book_title'],
        'post_type' => 'book_reviews',
        'post_content' => $_POST['book_review_text']
    );

    <!-- ... Data Processing code ... -->

    // Redirect browser to book review submission page
    $redirectaddress = ( empty( $_POST['_wp_http_referer'] ) ?
        site_url() : $_POST['_wp_http_referer'] );

    wp_redirect( $redirectaddress );
    exit;
}
} else {
    // Display message if any required fields are missing

```

11. Save and close the code file.
12. Go back to the book review submission form to see the new captcha field.



The screenshot shows a form with the text "Re-type the following text" above a captcha image. The captcha image displays the characters "Wm qS" in a stylized, overlapping font. To the right of the text is an empty text input field. Below the input field is a "Submit Review" button.

How it works...

The EasyCaptcha PHP script is a simple tool that can generate and display a captcha image and store the string that it is generated in a **cookie**. The resulting cookie can be queried in the data processing code to check if the value entered by the user matches the image that was displayed.

Before using this script, it is important to change the 17-character string as instructed in the recipe to make the resulting file difficult to find for anyone looking to trick the system.

The recipe's code gets the captcha image to be displayed by using a standard HTML `img` tag and pointing to the EasyCaptcha main PHP file as the image path.

On the data processing side, our code checks to see if the user captcha text matches with what was stored in the cookie and if it was generated less than five minutes ago. Based on the result of these verifications, the `valid` variable is set to be `true` or one of the few different error messages will be displayed to the user.

If the cookie validation result is positive, the previously created data processing and storage code is executed as before.

For more advanced content filtering, look up the Akismet API (<http://akismet.com/development/api/>).

7

Creating Custom MySQL Database Tables

In this chapter, we will cover the following topics around the creation of custom database tables:

- ▶ Creating new database tables
- ▶ Deleting custom tables on plugin removal
- ▶ Updating custom table structure on plugin upgrade
- ▶ Displaying custom table data in an admin page
- ▶ Inserting and updating records in custom tables
- ▶ Deleting records from custom tables
- ▶ Displaying custom database table data in shortcodes
- ▶ Implementing a search function to retrieve custom table data
- ▶ Importing data from a user file into custom tables

Introduction

As seen in *Chapter 4, The Power of Custom Post Types*, custom post types provide a very powerful and easy way to create and manage custom content in a WordPress installation. That being said, if the new items that you wish to create do not benefit from having access to the built-in text editor and have a large amount of data fields that need to be stored in the system, storing them using custom post types can actually become cumbersome. More specifically, each custom field requires a separate function call to be associated with a custom post. Also, custom fields have limited functionality since they store all their information in simple text fields, making it difficult to perform ordered queries based on special data types, such as dates.

An alternative solution to manage custom content is to create new tables in the site database and offer a custom interface to manage these new items.

While working directly with the site database might sound like a tall order, and should really only be done if custom post types don't work as desired, WordPress actually offers a utility class that makes it very easy to create new database tables, store information in these new structures, and perform data retrieval queries. Although having a basic level of **Structured Query Language (SQL)** knowledge will help understand all of the recipes in this chapter as we create a bug tracking system, each recipe thoroughly explains how each command works to produce the end result.

Creating new database tables

The first step in the creation of custom data elements to be stored in a custom database table is to create the table itself. This is done by preparing a standard SQL command that specifies the name of the table and its desired content, then getting WordPress to execute it on the site database.

This recipe shows how to prepare and execute a query that creates a table to hold bug reports.

Getting ready

You should have access to a WordPress development environment, either on your local computer or on a remote server, where you will be able to load your new plugin files.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch7-bug-tracker`.
3. Navigate to the directory and create a text file called `ch7-bug-tracker.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 7 - Bug Tracker`.
5. Add the following line of code before the closing `>> PHP` command at the end of the file to register a function to be called on plugin activation:

```
register_activation_hook( __FILE__, 'ch7bt_activation' );
```

6. Add the following code segment to provide an implementation for the `ch7bt_activation` function:

```
function ch7bt_activation() {  
    // Get access to global database access class  
    global $wpdb;
```

```

    // Create table on main blog in network mode or single blog
    ch7bt_create_table( $wpdb->get_blog_prefix() );
}

```

7. Insert the following code to provide an implementation for the `ch7bt_create_table` function, responsible for the actual table creation:

```

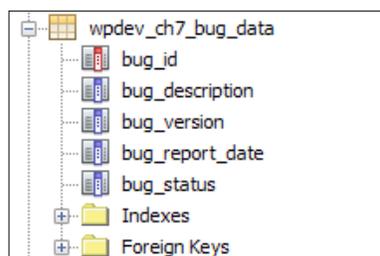
function ch7bt_create_table( $prefix ) {
    // Prepare SQL query to create database table
    // using function parameter

    $creation_query =
    'CREATE TABLE IF NOT EXISTS ' . $prefix . 'ch7_bug_data (
        `bug_id` int(20) NOT NULL AUTO_INCREMENT,
        `bug_description` text,
        `bug_version` varchar(10) DEFAULT NULL,
        `bug_report_date` date DEFAULT NULL,
        `bug_status` int(3) NOT NULL DEFAULT 0,
        PRIMARY KEY (`bug_id`)
    );';

    global $wpdb;
    $wpdb->query( $creation_query );
}

```

8. Save and close the plugin file.
9. Navigate to the **Plugins** management page and activate the **Chapter 7 – Bug Tracker** plugin.
10. Using phpMyAdmin or the NetBeans IDE, connect to your MySQL database to see that a new table was created when the plugin was activated.



How it works...

Similar to the creation of configuration options, custom database tables are typically created when a plugin is activated in a WordPress installation. By using the activation hook, we register code to be executed when the plugin is first activated and when upgrades are performed. When the callback is executed, we have our first encounter with the global `wpdb` class. This utility class is instantiated by WordPress and gives us access to a number of methods that can be used to interact with the underlying MySQL site database as well as help prevent data-related security risks. These methods vary in complexity, ranging from simple calls that will quickly insert or update records to more complex member functions that require knowledge of SQL commands to produce the expected results.

Before making the call to create the actual table, the activation function makes a call to the `get_blog_prefix` method of the `wpdb` class to retrieve the table prefix associated with the site (set to `wp_` in a default installation). On retrieval, this prefix is immediately sent to the `ch7bt_create_table` function to build an SQL command designed to create a new table.

While the SQL command has multiple lines, we can see that it is actually quite simple if we break it down into small sections. The first line of the command specifies that a new table named `<prefix>ch7_bug_data` should be created if it does not exist already on the server. If the creation takes place, the following five lines specify the name and data type for each field, along with information indicating if the field can contain a `NULL` value and what the default value should be in some cases. There is also a special command associated with the `bug_id` field, called the `AUTO_INCREMENT` command, that tells the system to automatically populate this field with auto-incrementing values when new records are added to the table. Last but not least, the last line of the code indicates that the primary key for the table is the `bug_id` field.



Even if the table name contains upper-case characters, the resulting table will only have lower-case characters in its name.

Once the query is ready, it is stored in a variable and executed by calling the `query` method of the `wpdb` object. This method executes any SQL command on the site database and returns a numeric value indicating how many rows were affected by the query.

There's more...

While the previous code is relatively manageable, things might get a bit more complicated when dealing with larger number of fields or with network WordPress installation.

Using phpMyAdmin to simplify code creation

Instead of writing the table creation code from scratch, the phpMyAdmin database management tool can come in handy to prepare this code.

#	Column	Type	Collation	Attributes	Null	Default	Extra	Action
<input type="checkbox"/>	1 bug_id	int(20)			No	None	AUTO_INCREMENT	Change Drop
<input type="checkbox"/>	2 bug_description	text	latin1_swedish_ci		Yes	NULL		Change Drop
<input type="checkbox"/>	3 bug_version	varchar(10)	latin1_swedish_ci		Yes	NULL		Change Drop
<input type="checkbox"/>	4 bug_report_date	date			Yes	NULL		Change Drop
<input type="checkbox"/>	5 bug_status	int(3)			No	0		Change Drop

For example, to create the table that was used in this recipe, follow these steps:

1. Select the `wordpressdev` database in phpMyAdmin.
2. Under the **Create new table on database wordpressdev** section, enter `wpdev_ch7_bug_data` in the **Name** field and the number 5 as the **Number of fields**.
3. Click on the **Go** button.
4. In the table creation grid that is displayed, set the name of each **Field** based on the **Column** names listed in the previous screenshot.
5. Set the **Type** of each **Field** based on the **Type** column in the previous screenshot.
6. For items that have a value in parentheses next to their **Type**, use the numeric value to indicate the **Length/Values** of these items.
7. Set the **Default** value for each field based on the previous screenshot. You can select **NULL** from the drop-down list for items that have a NULL default. For items that have a specific value, select **As defined:** in the drop-down and indicate the value in the adjacent field.
8. For items that are allowed to have a **NULL** value (shown with a **Yes** in the previous screenshot), check the **Null** box.
9. Select **PRIMARY** under the **Index** drop-down list for the `bug_id` field to indicate that it will be the primary key for the table.
10. Check the **A_I** box for the `bug_id` field to indicate that it should auto-increment when new values are inserted in the table.
11. Click on the **Go** button to complete the table creation process.

At this time, phpMyAdmin will create the table on the server and display the SQL command that it used to perform this task. You will see that this command is very similar to the one used in the recipe's code.

To display this command at a later time, select the database and click on the **Export** tab. In the **Export** dialog, uncheck the **Data** box and click on the **Go** button to see the related SQL command.

Create tables in network installation

One of WordPress' many strengths is the ability to create and manage multiple sites from a single installation. In these situations, each site has its own set of tables in the MySQL database. Therefore, when preparing a plugin that creates custom tables and may be used in network installations, extra code must be put in place to create the new tables under each site's structure.

The first changes are done in the `ch7bt_activation` function where we check if we are dealing with a multisite installation. If that is the case, we cycle through each existing site and make a call to create the new table as we have seen in the main recipe code.

```
function ch7bt_activation() {
    // Get access to global database access class
    global $wpdb;

    // Check to see if WordPress installation is a network
    if ( is_multisite() ) {

        // If it is, cycle through all blogs, switch to them
        // and call function to create plugin table
        if ( !empty( $_GET['networkwide'] ) ) {
            $start_blog = $wpdb->blogid;

            $blog_list =
                $wpdb->get_col( 'SELECT blog_id FROM ' .
                    $wpdb->blogs );

            foreach ( $blog_list as $blog ) {
                switch_to_blog( $blog );

                // Send blog table prefix to creation function
                ch7bt_create_table( $wpdb->get_blog_prefix() );
            }
            switch_to_blog( $start_blog );
            return;
        }
    }

    // Create table on main blog in network mode or single blog
    ch7bt_create_table( $wpdb->get_blog_prefix() );
}
```

While this will handle creating custom tables in all existing network sites when the plugin is activated, additional code needs to be put in place to create the additional table when new sites are created.

```
// Register function to be called when new blogs are added
// to a network site
add_action( 'wpmu_new_blog', 'ch7bt_new_network_site' );

function ch7bt_new_network_site( $blog_id ) {
    global $wpdb;

    // Check if this plugin is active when new blog is created
    // Include plugin functions if it is
    if ( !function_exists( 'is_plugin_active_for_network' ) )
        require_once( ABSPATH .
            '/wp-admin/includes/plugin.php' );

    // Select current blog, create new table and switch back
    if ( is_plugin_active_for_network( plugin_basename
        ( __FILE__ ) ) ) {
        $start_blog = $wpdb->blogid;
        switch_to_blog( $blog_id );

        // Send blog table prefix to table creation function
        ch7bt_create_table( $wpdb->get_blog_prefix() );

        switch_to_blog( $start_blog );
    }
}
```

The `ch7bt_create_table` function itself does not require any modifications since it was already designed to receive a table prefix from other functions and use it to build a query.

Deleting custom tables on plugin removal

It is always a good practice for plugins to provide an uninstallation procedure to remove content that they added to a site's database or filesystem. When dealing with custom database tables, all records should be dropped along with the table itself when a site administrator decides to delete a plugin.

This recipe shows how to implement a data removal script to delete the bug storage table that was created in the previous recipe.

Getting ready

You should have already followed the *Creating new database tables* recipe to have an existing table to remove. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v1-1.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a text file called `uninstall.php` in the `ch7-bug-tracker` directory and open it in a code editor.
3. Start the new script with the standard `<?php` opening tags.
4. Add the following code to perform the deletion of tables created to store bugs from a single or network WordPress installation:

```
// Check that file was called from WordPress admin
if( !defined( 'WP_UNINSTALL_PLUGIN' ) )
    exit();

global $wpdb;

// Check if site is configured for network installation
if ( is_multisite() ) {
    if ( !empty( $_GET['networkwide'] ) ) {
        // Get blog list and cycle through all blogs
        $start_blog = $wpdb->blogid;

        $blog_list =
            $wpdb->get_col( 'SELECT blog_id FROM ' .
                $wpdb->blogs );

        foreach ( $blog_list as $blog ) {
            switch_to_blog( $blog );

            // Call function to delete bug table with prefix
            ch7bt_drop_table( $wpdb->get_blog_prefix() );
        }
        switch_to_blog( $start_blog );
        return;
    }
}

ch7bt_drop_table( $wpdb->prefix );
```

5. Implement the `ch7bt_drop_table` function that was referenced in the previous block of code:

```
function ch7bt_drop_table( $prefix ) {
    global $wpdb;
    $wpdb->query( $wpdb->prepare( 'DROP TABLE ' . $prefix .
        'ch7_bug_data' ) );
}
```

6. Close the script with a closing `?>` command.
7. Save and close the code file.
8. Navigate to the **Plugins** management page and deactivate the **Chapter 7 – Bug Tracker** plugin.
9. Make a copy of the entire plugin directory before performing the next step to avoid deleting all of your work.
10. Click on the plugin's **Delete** link, then click on **Yes, Delete these files and data**.
11. Using phpMyAdmin or the NetBeans IDE, connect to your MySQL database to verify that the bug data table has been deleted.

How it works...

As we have seen in *Chapter 2, Plugin Framework Basics*, all of the code contained in a file called `uninstall.php` gets executed when a plugin is deleted. In this case, our code's main purpose is to run a query against the site database to remove the bug table.

Before doing so, the first few lines of the file check for the presence of a variable (`WP_UNINSTALL_PLUGIN`) to confirm that the code has been called as part of the plugin deletion process, and not by an external user.

Once the legitimacy of the execution has been confirmed, the code that runs is similar to the table creation code, where we first get access to the WordPress database management class, followed by a check to see if the WordPress installation is a single site or a network installation. In the first case, we make a single call to the `ch7bt_drop_table` function to drop the bug table, while we make multiple calls to that function for every existing site under a network environment.

The query to remove the table is actually quite simple, making a call to the `query` method of the `wpdb` class to execute a `DROP TABLE` SQL command.

See also

- ▶ [Creating new database tables recipe](#)

Updating custom table structure on plugin upgrade

Over the lifetime of a plugin, as it gets expanded to provide additional functionality, there may be a need to store more data than was originally intended in custom database tables. As you may know, WordPress itself makes regular changes to its own database structure during the upgrade process to store new information. To do this, it uses a simple function called `dbDelta` that we can also access from our plugin's code.

This recipe shows how to alter the previous table creation code to load the WordPress upgrade API and use the database upgrade function to add an extra field to the existing bug storage table.

Getting ready

You should have already followed the *Creating new database tables* recipe to have table creation code to modify. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v1-1.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch7-bug-tracker` directory and edit `ch7-bug-tracker.php`.
3. Locate the `ch7bt_create_table` function.
4. Remove the `IF NOT EXISTS` text on the first line of the table creation query.
5. Add an extra line to the table creation code to add a field to hold the bug title:

```
$creation_query = 'CREATE TABLE ' . $prefix . 'ch7_bug_data ('
    `bug_id` int(20) NOT NULL AUTO_INCREMENT,
    `bug_description` text,
    `bug_version` varchar(10) DEFAULT NULL,
    `bug_report_date` date DEFAULT NULL,
    `bug_status` int(3) NOT NULL DEFAULT 0,
    `bug_title` VARCHAR( 128 ) NULL,
    PRIMARY KEY (`bug_id`)
);';
```

6. Replace the following lines of code:

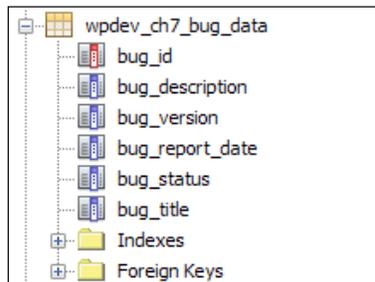
```
global $wpdb;
$wpdb->query( $creation_query );
```

With:

```
require_once( ABSPATH . 'wp-admin/includes/upgrade.php' );
dbDelta( $creation_query );
```

7. Save and close the plugin file.
8. Navigate to the **Plugins** management page.
9. Deactivate and reactivate the **Chapter 7 – Bug Tracker** plugin.

- Using phpMyAdmin or the NetBeans IDE, connect to your MySQL database to see that the new `bug_title` field has been added to the `bug` storage table.



How it works...

The `dbDelta` function is part of the utility functions that WordPress calls when performing version upgrades. When called, it parses the table creation SQL command that it receives and figures out the difference between the table structure that it describes and the current table, if the table exists. Once that difference has been established, it performs the necessary changes to align the two structures.

If both structures are identical, it leaves the table as is. With this approach in place, any changes to the structure can simply be implemented by altering the table creation query. As such, the `dbDelta` function can actually be used from the first version of a plugin to ensure an easy upgrade path.

See also

- ▶ [Creating new database tables recipe](#)

Displaying custom table data in an admin page

After creating one or more custom database tables to store data, the next step in the creation of a custom item management system is to build an interface to populate them. While custom post types have a very organized structure to edit entries, creating an interface for custom tables is much more similar to creating plugin configuration panels as we have seen in *Chapter 3, User Settings and Administration Pages*.

This recipe shows how to create an interface that will display a list of bugs stored in the system, provide a link to create new entries, and offer a way to edit existing entries.

Getting ready

You should have already followed the *Updating custom table structure on plugin upgrade* recipe to have a custom table in place with the full required structure. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v2.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch7-bug-tracker` directory and edit `ch7-bug-tracker.php`.
3. Insert the following line of code to register a function to be called when the administration menu is being built:

```
add_action( 'admin_menu', 'ch7bt_settings_menu' );
```

4. Add the following code to provide an implementation for the `ch7bt_settings_menu` function:

```
function ch7bt_settings_menu() {
    add_options_page( 'Bug Tracker Data Management',
                    'Bug Tracker', 'manage_options',
                    'ch7bt-bug-tracker',
                    'ch7bt_config_page' );
}
```

5. Append the following block of code to provide an implementation for the `ch7bt_config_page` function, responsible to render the configuration page:

```
function ch7bt_config_page() {
    global $wpdb;
    ?>

    <!-- Top-level menu -->
    <div id="ch7bt-general" class="wrap">
    <h2>Bug Tracker <a class="add-new-h2" href="<?php echo
        add_query_arg( array( 'page' => 'ch7bt-bug-tracker',
                            'id' => 'new' ),
                    admin_url('options-general.php') ); ?>">
    Add New Bug</a></h2>

    <!-- Display bug list if no parameter sent in URL -->
    <?php if ( empty( $_GET['id'] ) ) {
        $bug_query = 'select * from ';
        $bug_query .= $wpdb->get_blog_prefix() . 'ch7_bug_data ';
        $bug_query .= 'ORDER by bug_report_date DESC';
```

```

        $bug_items =
            $wpdb->get_results( $wpdb->prepare( $bug_query ),
                                ARRAY_A );
    ?>

    <h3>Manage Bug Entries</h3>

    <table class="wp-list-table widefat fixed" >

    <thead><tr><th style="width: 80px">ID</th>
    <th style="width: 300px">Title</th>
    <th>Version</th></tr></thead>

    <?php
        // Display bugs if query returned results
        if ( $bug_items ) {
            foreach ( $bug_items as $bug_item ) {
                echo '<tr style="background: #FFF">';
                echo '<td>' . $bug_item['bug_id'] . '</td>';
                echo '<td><a href="';
                echo add_query_arg( array(
                    'page' => 'ch7bt-bug-tracker',
                    'id' => $bug_item['bug_id'] ),
                    admin_url( 'options-general.php' ) );
                echo '">' . $bug_item['bug_title'] . '</a></td>';
                echo '<td>' . $bug_item['bug_version'] .
                    '</td></tr>';
            }
        } else {
            echo '<tr style="background: #FFF">';
            echo '<td colspan=3>No Bug Found</td></tr>';
        }
    ?>
    </table><br />

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

```

6. Save and close the plugin file.

7. Navigate to the new **Bug Tracker** item under the administration page's **Settings** menu to see the newly-created page, showing that there are currently no bugs stored in the system.



How it works...

The first few steps of the recipe use functions that were previously covered in *Chapter 3, User Settings and Administration Pages* to register a callback that will add a menu to the **Settings** section of the admin menu. When the new menu page is visited, the `ch7bt_config_page` function is called to render the page contents, using a mix of HTML and PHP code.

After rendering the page title, along with a link that will be used to create new bugs, the page display code checks to see if the page address contains a variable called `id`. This ID will be used in subsequent recipes to indicate whether the user wants to create or edit bugs. It will not be set when a visitor clicks on the **Bug Tracker** menu item, resulting in the current recipe code getting called.

The next section uses the `get_results` method of the `wpdb` database management class to retrieve information from the database. In this call, the first parameter is an SQL query, whereas the second argument indicates the desired format to be used to return data. While we specified that we want an associative array in this case, other options are to return a numerically indexed array (`ARRAY_N`), an object (`OBJECT`), or an array of objects (`OBJECT_K`). The `SELECT *` command in the query indicates that we want all fields in the table to be returned, while the `ORDER` command specifies the field that should be used to order results and the order direction (`ASC` or `DESC`).

Once the `get_results` method has been executed, we check to see if any data was retrieved from the database, and proceed to perform a `foreach` loop through all records to display them in a standard HTML table if data is found. If no records were returned by the query, we display a short message indicating that no bugs were found.

See also

- ▶ *Creating an administration page menu item in the Settings menu* recipe in *Chapter 3, User Settings and Administration Pages*
- ▶ *Rendering the admin page contents using HTML* recipe in *Chapter 3, User Settings and Administration Pages*

Inserting and updating records in custom tables

Now that we have a basic infrastructure in place to display existing bugs, the next logical step is to create a form that will be used to insert and update records in a custom table.

This recipe shows how to add a form to manage bugs when users select an entry in the bug tracking list or indicate that they want to create a new entry by using the appropriate link.

Getting ready

You should have already followed the *Displaying custom table data in an admin page* recipe to have an existing framework in place. Alternatively, you can get the resulting code (ch7-bug-tracker\ch7-bug-tracker-v3.php) from the code bundle and rename the file to ch7-bug-tracker.php.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the ch7-bug-tracker directory and edit ch7-bug-tracker.php.
3. Find the ch7bt_config_page function and locate the bracket that closes out the if statement (<?php } ?>) situated towards the end of its body.
4. Insert the following code block, right before the closing bracket from the if statement identified in the previous step:

```
<?php } elseif ( isset( $_GET['id'] ) &&
                ( $_GET['id'] == 'new' ||
                  is_numeric( $_GET['id'] ) ) ) {

    $bug_id = $_GET['id'];
    $bug_data = array();
    $mode = 'new';

    // Query database if numeric id is present
    if ( is_numeric( $bug_id ) ) {
        $bug_query = 'select * from ' . $wpdb->get_blog_prefix();
        $bug_query .= 'ch7_bug_data where bug_id = ' . $bug_id;

        $bug_data =
            $wpdb->get_row( $wpdb->prepare( $bug_query ),
                ARRAY_A );

        // Set variable to indicate page mode
        if ( $bug_data ) $mode = 'edit';
```

```

} else {
    $bug_data['bug_title'] = '';
    $bug_data['bug_description'] = '';
    $bug_data['bug_version'] = '';
    $bug_data['bug_status'] = '';
}

// Display title based on current mode
if ( $mode == 'new' ) {
    echo '<h3>Add New Bug</h3>';
} elseif ( $mode == 'edit' ) {
    echo '<h3>Edit Bug #' . $bug_data['bug_id'] . ' - ' . $bug_data['bug_title'] . '</h3>';
}
?>

<form method="post"
    action="<?php echo admin_url( 'admin-post.php' ); ?>"
    <input type="hidden" name="action" value="save_ch7bt_bug" />
    <input type="hidden" name="bug_id"
        value="<?php echo esc_attr( $bug_id ); ?>" />

    <!-- Adding security through hidden referrer field -->
    <?php wp_nonce_field( 'ch7bt_add_edit' ); ?>

    <!-- Display bug editing form -->
    <table>
        <tr>
            <td style="width: 150px">Title</td>
            <td><input type="text" name="bug_title" size="60"
                value="<?php echo esc_attr(
                    $bug_data['bug_title'] ); ?>" /></td>
        </tr>
        <tr>
            <td>Description</td>
            <td><textarea name="bug_description" cols="60">
                <?php echo esc_textarea(
                    $bug_data['bug_description'] ); ?></textarea></td>
        </tr>
        <tr>
            <td>Version</td>
            <td><input type="text" name="bug_version"
                value="<?php echo esc_attr(
                    $bug_data['bug_version'] ); ?>" /></td>
        </tr>
    </table>

```

```

        <td>Status</td>
        <td>
            <select name="bug_status">
                <?php
                    // Display drop-down list of bug statuses
                    // from list in array
                    $bug_statuses = array( 0 => 'Open', 1 => 'Closed',
                                           2 => 'Not-a-Bug' );

                    foreach( $bug_statuses as $status_id => $status ) {
                        // Add selected tag when entry matches
                        // existing bug status
                        echo '<option value="' . $status_id . '" ' .
                            selected( $bug_data['bug_status'],
                                       $status_id );
                        echo '>' . $status;
                    }
                ?>
            </select>
        </td>
    </tr>
</table>
<input type="submit" value="Submit" class="button-primary"/>
</form>

```

5. Add the following line of code to register a function that will be called on initialization of the administration page:

```
add_action( 'admin_init', 'ch7bt_admin_init' );
```

6. Add the following block of code at the end of the plugin file, to register a function to be called when bugs are created or updated:

```
function ch7bt_admin_init() {
    add_action( 'admin_post_save_ch7bt_bug',
                'process_ch7bt_bug' );
}
```

7. Append the following block of code to process user-submitted data and store it in the site database:

```
function process_ch7bt_bug() {
    // Check if user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );
}
```

```
// Check if nonce field is present for security
check_admin_referer( 'ch7bt_add_edit' );

global $wpdb;

// Place all user submitted values in an array (or empty
// strings if no value was sent)
$bug_data = array();

$bug_data['bug_title'] =
( isset( $_POST['bug_title'] ) ? $_POST['bug_title'] : '' );

$bug_data['bug_description'] =
( isset( $_POST['bug_description'] ) ?
    $_POST['bug_description'] : '' );

$bug_data['bug_version'] =
( isset( $_POST['bug_version'] ) ?
    $_POST['bug_version'] : '' );

// Set bug report date as current date
$bug_data['bug_report_date'] = date( 'Y-m-d' );

// Set status of all new bugs to 0 (Open)
$bug_data['bug_status'] =
( isset( $_POST['bug_status'] ) ?
    $_POST['bug_status'] : 0 );

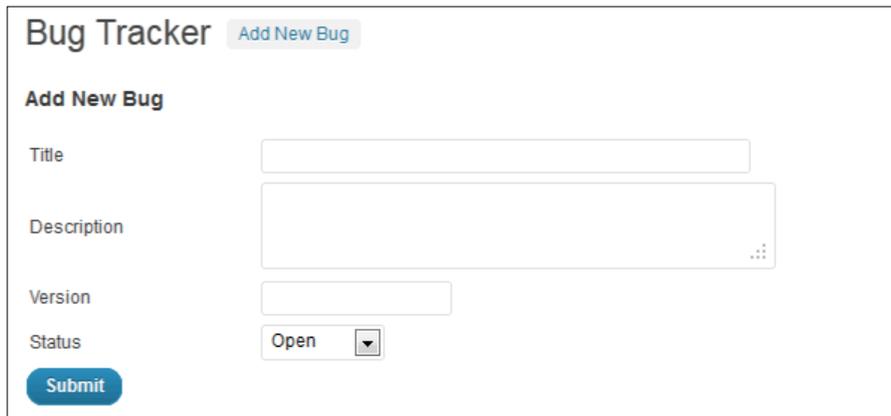
// Call the wpdb insert or update method based on value
// of hidden bug_id field
if ( isset( $_POST['bug_id'] ) && $_POST['bug_id'] == 'new' ) {
    $wpdb->insert( $wpdb->get_blog_prefix() .
        'ch7_bug_data', $bug_data );
} elseif ( isset( $_POST['bug_id'] ) &&
    is_numeric( $_POST['bug_id'] ) ) {
    $wpdb->update( $wpdb->get_blog_prefix() .
        'ch7_bug_data', $bug_data,
        array( 'bug_id' => $_POST['bug_id'] ) );
}

// Redirect the page to the user submission form
wp_redirect( add_query_arg( 'page', 'ch7bt-bug-tracker',
    admin_url( 'options-general.php' ) ) );

exit;
}
```

8. Save and close the plugin file.

9. Navigate to the new **Bug Tracker** item under the administration page's **Settings** menu and click on the **Add New Bug** link to create an entry.

The screenshot shows a web form titled "Bug Tracker" with a sub-header "Add New Bug" and a blue "Add New Bug" link. The form contains four input fields: "Title" (a single-line text box), "Description" (a multi-line text area with a small grid icon in the bottom right), "Version" (a single-line text box), and "Status" (a dropdown menu with "Open" selected). A blue "Submit" button is located at the bottom left of the form.

10. Click on **Submit** to store the new bug in the site database. The newly-created bug will appear in the bug listing created in the previous recipe.
11. Click on the new entry's name to review its information and update it.

How it works...

If you tried clicking on the **Add New Bug** link created in the previous recipe, you were presented with a page that only contained the panel's title. This is due to the fact that we had not implemented code to display a bug creation and editing form when the `id` variable is present in the site address.

The first few steps of this recipe aim to rectify this by checking for the presence of a variable called `id` in the page URL with a value set to the text `new` or a numeric value.

While both of these situations will result in displaying a bug edition form, the second condition first performs a database query using the `wpdb` object's `get_row` method to try to retrieve a bug with the designated ID. The `get_row` method is similar to the `get_results` method used in the previous recipe, but will only return a single row, even if more than one result was found by the query. If the query is successful, the values that were retrieved are used to customize the form title and set initial field values.

The form itself is a standard HTML form that includes many of the elements that we saw in previous recipes, such as a call to `wp_nonce_field` to provide security from external attacks. We have also added a hidden field containing the bug ID that was found in the page URL to facilitate data processing when a bug is submitted.

Once the form is in place, we make a call to `add_action` to register a callback that will be executed when the newly-created form is submitted.

The callback, named `process_ch7bt_bug`, starts off by doing a bit of validation. Namely, it checks to see if the current user has administrative rights and if the nonce field that should be part of the form data is present. If both of these conditions are met, a data array is created from user post data, the current system date, and a hard-coded status value.

The resulting array is stored in the site database using one of two `wpdb` object methods, `insert` or `update`, based on the value found in the hidden `bug_id` field. Both methods expect to receive the name of the target table, along with an associative array containing the names and values of each table field to be stored. Additionally, the `update` method requires a third parameter that indicates the field name and value to be used to locate the field to be updated. In both cases, you will notice that the `bug_id` field is not specified in the array of new values since it gets automatically set to an incremental value by the database server.

The last step in this function is to build a clean URL to the plugin configuration page and use the resulting address in a call to `wp_redirect`.

See also

- ▶ [Displaying custom table data in an admin page recipe](#)

Deleting records from custom tables

After adding data to custom tables, site administrators are likely to delete some of these entries down the road. Since we have been building an interface to view, create, and modify database entries, the task of selecting items to be deleted also falls under our responsibility. Thankfully, we can easily expand the existing bug display list to add checkboxes for selection and a button to trigger the actual deletion.

This recipe shows how to add deletion capabilities to our bug tracking system.

Getting ready

You should have already followed the *Inserting and updating records in custom tables* recipe to have an existing framework to augment. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v4.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch7-bug-tracker` directory and edit `ch7-bug-tracker.php`.
3. Find the `ch7bt_config_page` function and locate the `Manage Bug Entries` `h3` header in its content.
4. Insert the following highlighted lines of code right after the header to create a form:

```
<h3>Manage Bug Entries</h3>

<form method="post"
    action="<?php echo admin_url( 'admin-post.php' ); ?>"
    <input type="hidden" name="action" value="delete_ch7bt_bug" />

<!-- Adding security through hidden referrer field -->
<?php wp_nonce_field( 'ch7bt_deletion' ); ?>
```

5. A few lines down, add an empty column in the table header, before the `ID` field, as highlighted in the following line of code:

```
<thead><tr><th style="width: 50px"></th>
    <th style='width: 80px'>ID</th>
```

6. Within the main bug list display loop, insert the following highlighted code segments to add a checkbox in front of each item:

```
echo '<tr style="background: #FFF">';
echo '<td><input type="checkbox" name="bugs[]" value="";
echo esc_attr( $bug_item['bug_id'] ) . ' " /></td>';

echo '<td>' . $bug_item['bug_id'] . '</td>';
```

7. A few lines down, change the value of the `colspan` table row parameter from 3 to 4:

```
echo '<td colspan=4>No Bug Found</td></tr>';
```

8. Append the following highlighted lines of code after the `table` close tag to display a deletion button and terminate the form section:

```
</table><br />

<input type="submit" value="Delete Selected"
    class="button-primary"/>

</form>
```

9. Find the `ch7bt_admin_init` function and add the following function call at the end of its body:

```
add_action( 'admin_post_delete_ch7bt_bug',
    'delete_ch7bt_bug' );
```

10. Navigate to the bottom of the file and add the following code block to provide an implementation for the `delete_ch7bt_bug` function, responsible for processing deletion requests generated by the new form:

```
function delete_ch7bt_bug() {
    // Check that user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );

    // Check if nonce field is present
    check_admin_referer( 'ch7bt_deletion' );

    // If bugs are present, cycle through array and call SQL
    // command to delete entries one by one
    if ( !empty( $_POST['bugs'] ) ) {
        // Retrieve array of bugs IDs to be deleted
        $bugs_to_delete = $_POST['bugs'];

        global $wpdb;

        foreach ( $bugs_to_delete as $bug_to_delete ) {
            $query = 'DELETE from ' . $wpdb->get_blog_prefix();
            $query .= 'ch7_bug_data ';
            $query .= 'WHERE bug_id = ' .
                intval( $bug_to_delete );
            $wpdb->query( $wpdb->prepare( $query ) );
        }
    }

    // Redirect the page to the user submission form
    wp_redirect( add_query_arg( 'page', 'ch7bt-bug-tracker',
        admin_url( 'options-general.php' ) ) );
    exit;
}
```

11. Save and close the plugin file.
12. Navigate to the new **Bug Tracker** item under the administration page's **Settings** menu to see the new interface elements that were added to the bug listing.



How it works...

While the actual deletion of data from our custom table can be done with a single call to run the `DELETE` SQL command, we first need the user to indicate which entries need to be removed. This selection interface can be easily added to the existing bug listing created in an earlier recipe.

This recipe starts in familiar territory with the creation of a standard HTML form to surround the original bug listing. In addition to the bug list, the form also includes a hidden field to indicate the name of the action to be called when the user submits the form along with a nonce field to ensure that access to the deletion process is secure.

With this initial code in place, the next section of the recipe modifies the original table listing to add a checkbox at the front of every row. As can be seen in the code, the `name` property of the checkbox is a bit different than regular HTML syntax, ending with two square parentheses. This syntax, used in conjunction with each item's `bug_id`, results in the creation of an array of checked items, ID numbers that is sent to the form processing function on submission.

The last change that is done in the bug listing display code is to add a deletion button and to close the form.

To associate a callback with the newly-created form, the next addition made by the recipe is a call to `add_action` to associate the `admin_post_<actionname>` variable action name with the `delete_ch7bt_bug` function.

When called, the bug deletion function, like most other submission processing code that we have created before, first starts with a few verifications to make sure that the user has appropriate permissions and that the hidden security fields that were placed in the form are present. When both of these formalities are confirmed, the code goes on to check for the presence of a bug array and proceeds to cycle through all entries if one was found. In that loop, we get access to the global `wpdb` class and we can use it to build and execute SQL queries that delete a single database row at a time using the `bug_id` numbers that were submitted.

As an added security measure, notice the use of the `intval` function in front of the `$bug_to_delete` variable to make sure that no one is trying to get external commands to be processed in an attempt to corrupt or hijack the database.

See also

- ▶ *Inserting and updating records in custom tables* recipe

Displaying custom database table data in shortcodes

The purpose of creating custom tables is often to store information to be shared with site visitors. As such, it is important to give users the ability to easily display their new content stored in custom tables on their site. The most straightforward method to achieve this goal is to create one or more shortcodes that can be inserted on any post or page to render the desired information.

This recipe shows how to implement a new shortcode that will be used to display a bug listing on a page.

Getting ready

You should have already followed the *Deleting records from custom tables* recipe to have an existing framework to augment. Alternatively, you can get the resulting code (ch7-bug-tracker\ch7-bug-tracker-v5.php) from the code bundle and rename the file to ch7-bug-tracker.php.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the ch7-bug-tracker directory and edit ch7-bug-tracker.php.
3. Add the following line of code at the bottom of the file to declare a new shortcode and its associated display function:

```
add_shortcode( 'bug-tracker-list', 'ch7bt_shortcode_list' );
```

4. Insert the following code block right after the section header to implement the ch7bt_shortcode_list function that is responsible for displaying a bug listing:

```
function ch7bt_shortcode_list() {
    global $wpdb;

    // Prepare query to retrieve bugs from database
    $bug_query = 'select * from ' . $wpdb->get_blog_prefix();
    $bug_query .= 'ch7_bug_data ';
    $bug_query .= 'ORDER by bug_id DESC';

    $bug_items = $wpdb->get_results( $wpdb->prepare( $bug_query ),
                                    ARRAY_A );
```

```

// Prepare output to be returned to replace shortcode
$output = '';
$output .= '<table>';

// Check if any bugs were found
if ( !empty( $bug_items ) ) {
    $output .= '<tr><th style="width: 80px">ID</th>';
    $output .= '<th style="width: 300px">Title / Desc</th>';
    $output .= '<th>Version</th></tr>';

    // Create row in table for each bug
    foreach ( $bug_items as $bug_item ) {
        $output .= '<tr style="background: #FFF">';
        $output .= '<td>' . $bug_item['bug_id'] . '</td>';
        $output .= '<td>' . $bug_item['bug_title'] . '</td>';
        $output .= '<td>' . $bug_item['bug_version'];
        $output .= '</td></tr>';
        $output .= '<tr><td></td><td colspan="2">';
        $output .= $bug_item['bug_description'];
        $output .= '</td></tr>';
    }
} else {
    // Message displayed if no bugs are found
    $output .= '<tr style="background: #FFF">';
    $output .= '<td colspan=3>No Bugs to Display</td>';
}

$output .= '</table><br />';

// Return data prepared to replace shortcode on page/post
return $output;
}

```

5. Save and close the plugin file.
6. Create a new page and insert the newly-created shortcode [bug-tracker-list] in the page body.

7. View the page to see a list of bugs stored in the system.

ID	TITLE / DESC	VERSION
3	Second Entry Web site is not refreshing correctly.	1.0
2	First Bug Report This is a first bug report stored in a new custom database table.	1.0

How it works...

Creating a new shortcode to display custom table data is done in a very similar way to previous recipes. First, we declare the new code, along with the name of the function that will be called to generate text to replace it when found in posts or pages. Then, we create a display function to prepare all output and return it to WordPress.

The only distinction here is in the way we query the information. The recipe uses the `get_results` method of the `wpdb` class to query all bugs that exist in the custom database table using the `SELECT` SQL command. After this call is executed, all items found are returned in an associative array that can easily be displayed in a table form using a `foreach` loop.

If no entries were found, the recipe displays a simple message to inform the visitor.

Implementing a search function to retrieve custom table data

While content created using custom post types can be automatically searched by the built-in WordPress search engine, custom database tables don't benefit from the same treatment. Instead, plugin developers choosing this mechanism to store information must build their own search functionality.

This recipe shows how to add a search box to the bug listing created in the previous section and how to use the resulting query data to narrow down the list of bugs that are displayed by the shortcode.

Getting ready

You should have already followed the recipe entitled *Displaying custom database table data in shortcodes* to have an existing framework to augment. Alternatively, you can get the resulting code (ch7-bug-tracker\ch7-bug-tracker-v6.php) from the code bundle and rename the file to ch7-bug-tracker.php.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the ch7-bug-tracker directory and edit ch7-bug-tracker.php.
3. Find the ch7bt_shortcode_list function and add the following highlighted code after the initial global \$wpdb call to check if a search string was entered by a visitor:

```
global $wpdb;

if ( !empty( $_GET['searchbt'] ) ) {
    $search_string = $_GET['searchbt'];
    $search_mode = true;
} else {
    $search_string = "Search...";
    $search_mode = false;
}
```

4. Insert the following highlighted lines of code, in the middle of the existing query string to add where parameters using the user search text, if present:

```
$bug_query = 'select * from ' . $wpdb->get_blog_prefix();
$bug_query .= 'ch7_bug_data ';

// Add search string in query if present
if ( $search_mode ) {
    $search_term = '%'. $search_string . '%';
    $bug_query .= "where bug_title like '%s' ";
    $bug_query .= "or bug_description like '%s' ";
} else {
    $search_term = '';
}

$bug_query .= 'ORDER by bug_id DESC';
```

5. Change the following line of code:

```
$bug_items =
    $wpdb->get_results( $wpdb->prepare( $bug_query ),
                        ARRAY_A );
```

To:

```
$bug_items =
    $wpdb->get_results( $wpdb->prepare( $bug_query,
                                        $search_term, $search_term ),
                        ARRAY_A );
```

6. Add the following code block, before the table starts rendering, to display a simple search form:

```
$output = '';

$output .= '<form method="get" id="ch7_bt_search">';
$output .= '<div>Search bugs ';
$output .= '<input type="text" onfocus="this.value=\'\'" ';
$output .= 'value="" . esc_attr( $search_string ) . \'\' ';
$output .= 'name="searchbt" />';
$output .= '<input type="submit" value="Search" />';
$output .= '</div>';
$output .= '</form><br />';
```

```
$output .= '<table>';
```

7. Save and close the plugin file.
 8. Visit the bug display page that was previously created to see the new search form. Enter a search string and click on the **Search** button to see a list of results.

Search bugs <input type="text" value="Second"/> <input type="button" value="Search"/>		
ID	TITLE / DESC	VERSION
3	Second Entry	1.0
Web site is not refreshing correctly.		

How it works...

This recipe implements a simple search engine by displaying a short form and capturing a user search string using the standard HTML `GET` method. If a search string is found in the page address, we modify the bug retrieval query that was in place by adding a `where` clause that looks for the search string anywhere in the `bug_title` or `bug_description` fields.

While it might seem natural to insert the search string directly in the query and execute it, we use the `wpdb` class' `prepare` method to assemble the query and validate the search string to avoid malicious intent. This method works in a very similar way to the standard PHP `sprintf` function, with placeholders to represent the places where variables should be substituted.

The remainder of the shortcode display function remains identical, displaying a list of varying length depending on the presence of a search string and the number of entries that match the query.

See also

- ▶ *Displaying custom database table data in shortcodes* recipe

Importing data from a user file into custom tables

To avoid long data entry sessions, a nice addition to a system like the Bug Tracker that we have been putting in place in this chapter is to provide users with the ability to import large amounts of entries from an external file in a single operation. To accomplish this task, the **Comma-Separated Values (CSV)** file format is very convenient since it can be edited by most spreadsheet editors and can be read using standard PHP function calls.

This recipe implements a CSV-based import function in our bug tracking system.

Getting ready

You should have already followed the *Implementing a search function to retrieve custom table data* recipe to have an existing framework to augment. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v7.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch7-bug-tracker` directory and edit `ch7-bug-tracker.php`.
3. Find the `ch7bt_config_page` function and add the following highlighted code block at the end of the bug listings section, after the end of the existing deletion form:

```
<input type="submit" value="Delete Selected"
      class="button-primary"/>
</form>

<!-- Form to upload new bugs in csv format -->
<form method="post"
      action="<?php echo admin_url( 'admin-post.php' ); ?>"
      enctype="multipart/form-data">

<input type="hidden" name="action" value="import_ch7bt_bug" />

<!-- Adding security through hidden referrer field -->
<?php wp_nonce_field( 'ch7bt_import' ); ?>

<h3>Import Bugs</h3>
  Import Bugs from CSV File
  (<a href="<?php echo plugins_url( 'importtemplate.csv',
    __FILE__ ); ?>">Template</a>)
  <input name="importbugsfile" type="file" /> <br /><br />

<input type="submit" value="Import" class="button-primary"/>
</form>
```

4. Locate the `ch7bt_admin_init` function and add the following line of code at the end of its body to register a function to process submissions of the bug import form:

```
add_action( 'admin_post_import_ch7bt_bug',
            'import_ch7bt_bug' );
```

5. Insert the following block of code to provide an implementation for the `import_ch7bt_bug` function:

```
function import_ch7bt_bug() {
    // Check that user has proper security level
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );
```

```

// Check if nonce field is present
check_admin_referer( 'ch7bt_import' );

// Check if file has been uploaded
if( array_key_exists( 'importbugsfiler', $_FILES ) ) {
    // If file exists, open it in read mode
    $handle = fopen( $_FILES['importbugsfiler']['tmp_name'],
                    'r' );

    // If file is successfully open, extract a row of data
    // based on comma separator, and store in $data array
    if ( $handle ) {
        while ( ( $data = fgetcsv( $handle, 5000, ',' ) ) !==
                FALSE ) {
            $row += 1;

            // If row count is ok and row is not header row
            // Create array and insert in database
            if ( count( $data ) == 4 && $row != 1 ) {
                $new_bug = array(
                    'bug_title' => $data[0],
                    'bug_description' => $data[1],
                    'bug_version' => $data[2],
                    'bug_status' => $data[3],
                    'bug_report_date' => date( 'Y-m-d' ) );

                global $wpdb;

                $wpdb->insert( $wpdb->get_blog_prefix() .
                            'ch7_bug_data', $new_bug );
            }
        }
    }
}

// Redirect the page to the user submission form
wp_redirect( add_query_arg( 'page', 'ch7bt-bug-tracker',
                            admin_url( 'options-general.php' ) ) );
exit;
}

```

6. Save and close the plugin file.
7. Create a new text file in the plugin directory called `importtemplate.csv` and open it in a text editor.
8. Insert the following text in the newly-created file to provide an example bug to import:


```

"Title","Description","Version","Status"
"Test Import Bug","This is a test import bug","1.0","0"

```

9. Save and close the CSV text file.
10. Navigate to the new **Bug Tracker** item under the administration page's **Settings** menu to see the new **Import Bugs** section.
11. Use the file import dialog to locate the `importtemplate.csv`.
12. Import the list of bugs in the system to see its contents added to the database.



How it works...

This recipe creates a small form on the **Bug Tracker** management page that is solely responsible for uploading one or more bugs to the database. By editing the content of the `importtemplate.csv` file and selecting it in the import dialog, users can quickly populate the system by loading data straight to the custom database table that was created by the plugin when it was first installed.

In addition to the file upload field, the form contains the usual hidden nonce and action name fields. It also features an `enctype` property to allow files to be uploaded.

When the user submits a file to be uploaded, the registered callback function first checks to see whether the user who made the submission has appropriate rights and whether the nonce security fields were present as part of the post data. If both of these conditions are met, the recipe goes on to check to see if a file has been correctly uploaded to the web server by using the `array_key_exists` function to search through the standard PHP `$_FILES` global variable. As you can see, the text that it searches for is the name of the file upload field from the form.

If a file has been uploaded, the `fopen` function opens it and stores a pointer to it in a local variable. After a quick verification of the pointer's existence, the code moves to a `while` loop to process each line of the incoming file with the `fgetcsv` function. This function reads one line of the file at a time, analyzes its content to find all of the comma-separated fields that are present, and stores the resulting data in a numeric array.

The rest of the `import` function creates an array with the imported data and stores it in the database by using the `wpdb` class' `insert` method, as we have seen in a previous recipe.

See also

- ▶ [Inserting and updating records in custom tables recipe](#)

8

Leveraging JavaScript, jQuery, and AJAX Scripts

This chapter focuses on incorporating JavaScript in plugins by exploring the following topics:

- ▶ Safely loading jQuery onto WordPress web pages
- ▶ Displaying a pop-up dialog using the built-in ThickBox plugin
- ▶ Controlling pop-up dialog display using shortcodes
- ▶ Displaying a calendar day selector using the Datepicker plugin
- ▶ Adding tooltips to admin page form fields using the TipTip plugin
- ▶ Using AJAX to dynamically update partial page contents

Introduction

JavaScript libraries, especially the very popular jQuery library and its numerous plugins, can do wonders in bringing a website to life with slick animations, dynamic data queries, and advanced visual features. Unfortunately, for all of their benefits, these scripts can also be difficult to work with. For example, loading more than one copy of jQuery can destroy all the setup that was done by the other instances. Also, a single error in any of the JavaScript code found within a page, breaks all of them.

WordPress' answer to this convoluted architecture is two-fold. As a first step, it comes pre-packaged with a copy of jQuery and many other popular JavaScript libraries that plugin developers can use without having to load their own version. Then, to prevent multiple copies from being loaded on a page, it offers easy-to-use functions that queue up scripts and styles to identify duplicates before rendering pages.

This chapter shows how to safely load JavaScript and jQuery files that are provided with WordPress or that come from external sources to add powerful new functionality to front-facing pages and plugin configuration panels. It also explains how to securely run AJAX queries to refresh partial page sections.

Safely loading jQuery onto WordPress web pages

While it might be tempting to provide your own copy of jQuery as part of a new plugin that uses the popular JavaScript library, or to access a copy from the Google API website, WordPress actually provides a copy of jQuery in its installation and makes it very easy to load it.

By using the appropriate utility function to load jQuery, developers make a request to WordPress to load this library instead of doing it themselves. Once all requests have been received, they are analyzed for duplicates and a single instance of each script is loaded to reduce the chance of conflicts between multiple copies of the same library.

This recipe shows how to load the jQuery script for use on front-facing site pages.

Getting ready

You should have access to a WordPress development environment.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch8-load-jquery`.
3. Navigate to the directory and create a text file called `ch8-load-jquery.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 8 - Load jQuery`.
5. Add the following line of code before the closing `?>` PHP command to register a function to be called when script loading requests are processed:

```
add_action( 'wp_enqueue_scripts', 'ch8lj_front_facing_pages' );
```

6. Add the following code segment to provide an implementation for the `ch8lj_front_facing_pages` function:

```
function ch8lj_front_facing_pages() {  
    wp_enqueue_script( 'jquery' );  
}
```

7. Save and close the plugin file.

8. Navigate to the **Plugins** management page and activate the **Chapter 8 – Load jQuery** plugin.
9. Visit any page on the site and launch your browser's source viewer function.
10. Search for the keyword `jquery` to see that a copy of the script is now loaded from the WordPress `wp-includes` folder:

```
<script type='text/javascript' src='http://localhost/  
wp-includes/js/jquery/jquery.js?ver=1.7.1'></script>
```

How it works...

The key component of this recipe is the `wp_enqueue_script` function, which allows developers to load their own JavaScript files or to ask WordPress to load one of the scripts that it comes packaged with. While the function requires many arguments when loading your own scripts, which we'll cover in a later recipe, it only needs a single argument to load built-in scripts. In this example that argument is `jquery`. To get a full list of default scripts available with WordPress, check out the Codex page for the function (http://codex.wordpress.org/Function_Reference/wp_enqueue_script).

Once you know which script to load, the call to `wp_enqueue_script` should be made from one of three action hooks, depending on the target page(s) where the script should be loaded. These are: `wp_enqueue_scripts` for front-facing pages, `admin_enqueue_scripts` for administration pages, and `login_enqueue_scripts` for the login page, with the first one fulfilling our requirement for this recipe.

There's more...

Veteran jQuery developers should be aware that the copy delivered with WordPress has a small caveat.

jQuery noconflict mode

To avoid internal conflicts with other JavaScript and jQuery libraries, the version of jQuery that comes bundled with WordPress is configured in the `noconflict` mode. This means that the `$` shortcut that can normally be used to access jQuery will not be available. As such, all examples found in this chapter spell out the jQuery keyword.

To regain access to this shortcut, you can use the following syntax in your code:

```
jQuery( document ).ready( function($) {  
    // $ shortcut is now available for this function  
})  
);
```

Displaying a pop-up dialog using the built-in ThickBox plugin

As annoying as they can be to visitors, pop-up dialogs are a feature that many website administrators are using to help them advertise special offers or get readers to subscribe to their content. Since it uses pop-up dialogs in its own administrative pages, WordPress comes bundled with a jQuery script called ThickBox that can be used to display these type of dialogs.

This recipe shows how to load the ThickBox script and use it to render a pop-up dialog.

Getting ready

You should have access to a WordPress development environment.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch8-pop-up-dialog`.
3. Navigate to the directory and create a text file called `ch8-pop-up-dialog.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 8 – Pop-Up Dialog`.
5. Add the following line of code before the closing `?>` PHP command to register a function to be called when script loading requests are made:

```
add_action( 'wp_enqueue_scripts', 'ch8pud_load_scripts' );
```

6. Add the following code segment to provide an implementation for the `ch8pud_load_scripts` function:

```
function ch8pud_load_scripts() {  
    wp_enqueue_script( 'jquery' );  
    add_thickbox();  
}
```

7. Insert the following line of code to register a function to display content in the page footer:

```
add_action( 'wp_footer', 'ch8pud_footer_code' );
```

8. Append the following block of code to provide an implementation for the `ch8pud_footer_code` function:

```
function ch8pud_footer_code() { ?>  
    <script type="text/javascript">  
        jQuery( document ).ready(function() {
```


How it works...

Similar to the previous recipe, we start by assigning a function to the `wp_enqueue_scripts` action hook. When executed, the callback makes a call to `wp_enqueue_script` to request for jQuery to be loaded from the local copy of WordPress. The next line calls the `add_thickbox` function, which is a utility function that makes multiple calls to `wp_enqueue_script` and `wp_enqueue_style` to load the appropriate JavaScript and stylesheet in the page header.

Once all required elements are loaded, the next section of the recipe outputs a block of JavaScript code to the page footer that uses jQuery to register a function that will be called when the entire page is loaded. When this happens, the `setTimeout` JavaScript function is used to register a function that will be called 2000 milliseconds later and take care of calling `tb_show` to display the pop-up dialog. `tb_show` has three arguments, with the first one indicating the dialog title, the second containing the address of the content to render within the box, and the third expecting a path to a group of images to be displayed. In our case, the last argument is left null. Notice that the width and height (in pixels) of the dialog are indicated as part of the address of the content page to be displayed.



If you are using the default WordPress twenty-eleven theme, the pop-up dialog will be obstructed by the theme header image, since it has been set to have a high z-index value in the theme's stylesheet.

To resolve this issue, edit the `wp-content/themes/twenty-eleven/style.css` file and remove line 508:

```
z-index: 9999;
```

There's more...

While the recipe displays a valid dialog, developers might want a bit more control over how it can be closed and when it gets displayed.

Removing the dialog close button

By default, the ThickBox script offers a close button on the top-right corner of the pop-up dialog that can be used to close it at any time. This may not be desirable if you expect visitors to provide feedback or perform a specific action before dismissing the dialog. By adding the `modal` keyword to the content URL—set to a value of `true`—ThickBox will remove the dialog title bar, including the close button.

```
tb_show( 'Pop-Up Message', '<?php echo plugins_url(
    'content.html?width=420&height=220&modal=true', __FILE__ )?>',
    null );
```

Once the close button is gone, we can call the `tb_remove` JavaScript function to close the dialog. The following is an example of a simple link that will close the dialog:

```
<div><a href="#" onclick="tb_remove();">Close Dialog</a></div>
```

Displaying pop-up dialogs on select pages

While the recipe's original code displays a pop-up dialog on every single page of a site, it may be better to show it only on specific pages, such as the front page. To accomplish this, we can place a condition around the two `add_action` calls to check if the visitor is making a request to see the front page:

```
If ( is_front_page() ) {
    add_action( 'wp_enqueue_scripts', 'ch8pud_load_scripts' );
    add_action( 'wp_footer', 'ch8pud_footer_code' );
}
```

A similar technique can be used by substituting the `is_front_page` function by the `is_page('id_title_or_slug')` function, which checks if the current page numeric ID, title, or post slug matches the value that it receives as an argument. In that situation, a plugin configuration page would allow users to easily select one or more pages on which the dialog should appear.

Controlling pop-up dialog display using shortcodes

As you may be aware, loading scripts and styles on a page where they won't be used unnecessarily slows down that page's rendering time since the browser will still need to download and validate the content of these external files. While the previous recipe offered one way to select specific pages where scripts and styles should be loaded, a different approach is to analyze the page contents for the presence of a special code to make that decision.

This recipe shows how to add a filter to the previous recipe to search for a shortcode in posts and pages to decide when to display a pop-up dialog.

Getting ready

You should have already followed the *Displaying a pop-up dialog using the built-in ThickBox plugin* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch8-pop-up-dialog\ch8-pop-up-dialog-v1.php`) from the code bundle and rename the file to `ch8-pop-up-dialog.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch8-pop-up-dialog` directory and then to edit `ch8-pop-up-dialog.php`.
3. Find the `ch8pud_load_scripts` function and add the following highlighted lines of code:

```
function ch8pud_load_scripts() {  
    // Only load scripts if variable is set to true  
    global $load_scripts;  
  
    if ( $load_scripts ) {  
        wp_enqueue_script( 'jquery' );  
        add_thickbox();  
    }  
}
```

4. Locate the `ch8pud_footer_code` function and modify the code, adding the following highlighted lines of code to the function body:

```
function ch8pud_footer_code() {  
    // Only load scripts if keyword is found on page  
    global $load_scripts;  
    if ( $load_scripts ) { ?>  
  
    <script type="text/javascript">  
        jQuery( document ).ready( function() {  
            setTimeout(  
                function(){  
                    tb_show( 'Pop-Up Message',  
                        '<?php echo plugins_url(  
                            'content.html?width=420&height=220',  
                            __FILE__ )?>', null );  
                }, 2000 );  
            });  
        }  
    </script>  
  
    <?php }  
}
```

5. Add the following line of code to register a function that will filter post and page contents before any other parsing and formatting is performed:

```
add_filter( 'the_posts',  
            'ch8pud_conditionally_add_scripts_and_styles' );
```

6. Append the following block of code to provide an implementation for the `ch8pud_conditionally_add_scripts_and_styles` function:

```
function ch8pud_conditionally_add_scripts_and_styles( $posts ) {
    // Exit function immediately if no posts are present
    if ( empty( $posts ) ) return $posts;

    // Global variable to indicate if scripts should be loaded
    global $load_scripts;
    $load_scripts = false;

    // Cycle through posts and set flag true if
    // keyword is found
    foreach ( $posts as $post ) {
        $shortcode_pos = stripos
            ( $post->post_content, '[popup]', 0 );
        if ( $shortcode_pos !== false ) {
            $load_scripts = true;
            return $posts;
        }
    }

    // Return posts array unchanged
    return $posts;
}
```

7. Insert the following function call to declare a new shortcode, along with a function responsible for replacing it with content:

```
add_shortcode( 'popup', 'ch8pud_popup_shortcode' );
```

8. Add the following code block to provide a simple implementation for the `ch8pud_popup_shortcode` function:

```
function ch8PUD_popup_shortcode() {
    return;
}
```

9. Save and close the plugin file.
10. Visit the site front page and you will notice that the pop-up dialog is no longer displayed.
11. Create a new page and insert the `[popup]` shortcode in the page contents.
12. View the new page to see that the new pop-up dialog appears, while the `[popup]` shortcode is not shown.

How it works...

Global PHP variables are powerful tools that can help us share data between functions in a plugin. By using the keyword `global` in front of the name of a variable, a site's PHP interpreter will know that it has to access a common memory space to store and access information.

While the existing action hooks were first modified to query a global variable to determine whether or not they should load scripts and output code to the page footer, the bulk of the work is actually done by the filter function that gets associated to the `the_posts` hook. This function receives an array of all posts and pages that are destined to be displayed and must determine if a special keyword is present to set the `load_scripts` variable appropriately.

As you can see from the recipe's code, the text that we chose to look for, `[popup]`, is a shortcode. While we could have selected any text as the trigger to display a pop-up dialog, we chose a shortcode since it would be easy to make it disappear by providing a simple rendering function for it that returns an empty string.

See also

- ▶ *Displaying a pop-up dialog using the built-in ThickBox plugin recipe*

Displaying a calendar day selector using the Datepicker plugin

For all of its great administrative control panels and user interface elements, WordPress still has a simplistic approach to date selection, making users interact with a drop-down box and text fields to indicate the month, day, year, and time when a post or page is to be published. A much more interesting way to enter this type of information is to use a pop-up calendar that allows users to navigate through visual representations of each month and pick the desired date.

This recipe shows how to use the jQuery Datepicker script that is provided by default with WordPress to display a pop-up calendar to provide an easy way to select dates.

Getting ready

You should have access to a WordPress development environment.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch8-calendar-picker`.

3. Visit www.jqueryui.com/download and download the latest version of the jQuery UI package, with all components selected.
4. Open the resulting file with an archive management tool and only extract the `css` folder to the newly-created plugin directory.
5. Create a text file called `ch8-calendar-picker.php` in the plugin directory.
6. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 8 - Calendar Picker`.
7. Add the following line of code before the closing `?>` PHP command at the end of the file to register a function to be called when script loading requests are made:

```
add_action( 'admin_enqueue_scripts', 'ch8cp_admin_scripts' );
```

8. Add the following code segment to provide an implementation for the `ch8cp_admin_scripts` function:

```
function ch8cp_admin_scripts() {
    wp_enqueue_script( 'jquery' );
    wp_enqueue_script( 'jquery-ui-core' );
    wp_enqueue_script( 'jquery-ui-datepicker' );
    wp_enqueue_style( 'datepickercss',
        plugins_url( 'css/ui-lightness/ jquery-ui-1.8.17.custom.css',
            __FILE__ ), array(), '1.8.17' );
}
```

 The name of the stylesheet CSS file should be updated to reflect the name of the version of jQuery UI that was downloaded.

9. Insert the following line of code to register a function to be called when meta boxes are created:

```
add_action( 'add_meta_boxes', 'ch8cp_register_meta_box' );
```

10. Append the following block of code to provide an implementation for the `ch8cp_register_meta_box` function:

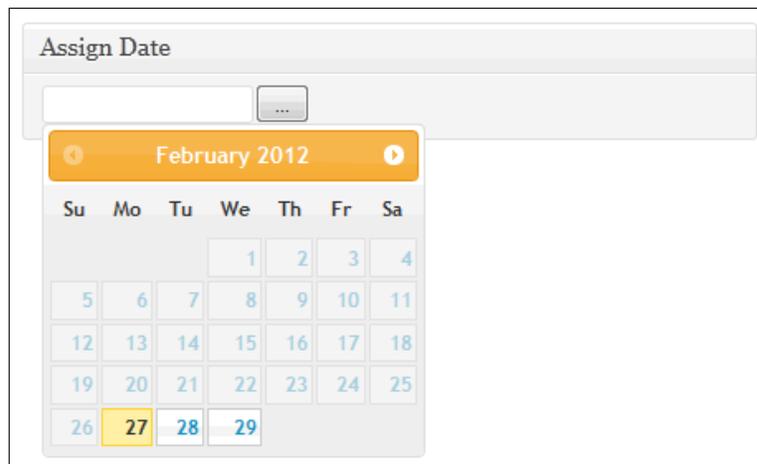
```
function ch8cp_register_meta_box() {
    add_meta_box( 'ch8cp_datepicker_box', 'Assign Date',
        'ch8cp_date_meta_box', 'post', 'normal' );
}
```

11. Insert the following code block to implement the `ch8cp_date_meta_box` function that was declared in the call to `add_meta_box`:

```
function ch8cp_date_meta_box( $post ) { ?>
    <input type="text" id="ch8cp_date" name="ch8cp_date" />

    <!-- JavaScript function to display calendar button -->
    <!-- and associate date selection with field -->
    <script type='text/javascript'>
        jQuery( document ).ready( function() {
            jQuery( '#ch8cp_date' ).datepicker( { minDate: '+0',
                dateFormat: 'yy-mm-dd', showOn: 'both',
                constrainInput: true } );
        } );
    </script>
<?php }
```

12. Save and close the plugin file.
13. Navigate to the **Plugins** management page and activate the **Chapter 8 – Calendar Picker** plugin.
14. Select any item in the **Posts** management section and edit it to see the new date assignment meta box.
15. Click on the **...** button or click in the **Assign Date** textbox to display the pop-up calendar and select a date.



How it works...

Just like we saw with jQuery and ThickBox in the previous recipes, WordPress comes bundled with many jQuery libraries. Two of these libraries, jQuery UI and jQuery UI Datepicker, can be used to display a pop-up calendar and associate it with a text field on a form. That being said, the distribution of these scripts is missing the associated stylesheet and images that are required to display a fully rendered calendar.

This recipe starts by visiting the jQuery UI site and downloading a copy of the complete library, which includes all the required layout files. Once the download is complete, we are only interested in getting a copy of the style data since all other necessary scripts are provided by WordPress. After registering a function with `admin_enqueue_scripts`, we make three function calls to load the required JavaScript files in the admin page header. We also make a call to load the stylesheet that we just downloaded. The `wp_enqueue_style` function has many parameters. In this example, we are providing values for the first four of them to indicate the name of the style, the path to the style file, an empty list of dependencies, and a version number. This function also has a fifth parameter, which we are not using here, to indicate if the script should be loaded in the header or footer, where the default is the header.

Once all of the required scripts are in place, the remainder of the code creates a meta box in the post editor, displays a text field in that box, and outputs JavaScript code that will be called when the page is completely rendered to associate the pop-up calendar with the text field. As part of the calendar's options, we specify that the user will only be able to select future dates with the `minDate` parameter along with the desired date format.

Adding tooltips to admin page form fields using the TipTip plugin

Documentation is a very important step of plugin development as it will allow users to understand how to configure the plugins you create. That being said, users will not typically go very far to find the information they need, resulting in many unnecessary questions in discussion forums or in e-mails.

As discussed in *Chapter 3, User Settings and Administration Pages*, one way to provide documentation is to create a help tab that appears in the top-right corner of the plugin's configuration panel. While that approach is much easier for users than to find a readme file or go back to the official WordPress plugin repository, it still requires them to actively seek and click a link to open that section.

That's where tooltips come into play. By using a jQuery plugin to render clean good-looking tooltips, we can add documentation to a plugin that will be displayed contextually based on the configuration fields that the user is currently interacting with.

This recipe shows how to download and integrate the TipTip jQuery library to display tooltips when configuration fields are used.

Getting ready

You should have already followed the *Displaying a calendar day selector using the Datepicker plugin* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch8-calendar-picker\ch8-calendar-picker-v1.php`) from the code bundle and rename the file to `ch8-calendar-picker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch8-calendar-picker` directory.
3. Create a new subdirectory called `tiptip`.
4. Visit the TipTip jQuery home page available at <http://drew.tenderapp.com/kb/tiptip-jquery-plugin/tiptip-downloads>.
5. Download Version 1.3 of the plugin source code to your local computer.
6. Open the resulting file with an archive management tool and extract the `jquery.tipTip.js` and `tipTip.css` files to the `tiptip` directory.
7. Open the main plugin file `ch8-calendar-picker.php` in a code editor.
8. Find the `ch8cp_admin_scripts` function and add the following lines of code at the end:

```
wp_enqueue_script( 'tiptipjs',
                  plugins_url( 'tiptip/jquery.tipTip.js',
                               __FILE__ ),
                  array(), '1.3' );
wp_enqueue_style( 'tiptip',
                 plugins_url( 'tiptip/tipTip.css', __FILE__ ),
                 array(), '1.3' );
```

9. Locate the `ch8cp_date_meta_box` function and modify the line that renders the textbox as shown in the following highlighted code:

```
<input type="text" class="ch8cp_tooltip"
       title="Please enter a date" id="ch8cp_date"
       name="ch8cp_date" />
```

10. Again in the `ch8cp_date_meta_box` function, add the following highlighted block of code to the existing block of JavaScript code:

```
<script type='text/javascript'>
    jQuery( document ).ready( function() {
        jQuery( '#ch8cp_date' ).datepicker( { minDate: '+0',
        dateFormat: 'yy-mm-dd', showOn: 'both',
        constrainInput: true } );
    });
```

```

        jQuery( '.ch8cp_tooltip' ).each( function() {
            jQuery( this ).tipTip();
        }
    );
});
</script>

```

11. Save and close the plugin file.
12. Select any item in the **Posts** management section and edit it.
13. Move the mouse over the date field to see the new tooltip appear.



How it works...

The TipTip library turns regular HTML `title` tags into nice-looking tooltips that appear when users position their mouse cursor over an item or select it.

This recipe starts by downloading the TipTip script from the plugin author's website. Once downloaded, we only extract two of the three files that the archive contains. The third file, containing the keyword minified in its name, is not needed as it is a second copy of the plugin code with every space and line feed removed to make it as compact as possible.

Once we have the desired files in place, we load them in the admin page header by adding calls to the `wp_enqueue_script` and `wp_enqueue_style` functions in the callback that was already associated with the `admin_enqueue_script` action hook. Similar to `wp_enqueue_style`, the `wp_enqueue_script` function has five parameters, which indicate the name of the script, the location of the script file, a list of any dependencies for the script, a version number, and an option to indicate if the script should be loaded in the site header or footer.

Once the library is loaded, activating the tooltips is quite simple. First, we select a class name for our items and add it to all items that are destined to have help text associated with them. Then, we add the help text in a `title` tag on each item. Note that the item in question could be anything from a `div`, to a form input component or a table row. Finally, we make a call to a jQuery function to find all items that have the right class and execute the TipTip function on them. After execution, all selected items will have their title text appear as tooltips.

See also

- ▶ *Displaying a calendar day selector using the Datepicker plugin recipe*

Using AJAX to dynamically update partial page contents

When users create complex websites with lots of dynamic content, such as Twitter widgets or other components that fetch external data, refreshing the entire page every time a user interacts with the site can quickly become a gruelling experience for visitors.

In such situations, using **AJAX (Asynchronous JavaScript and XML)** can greatly accelerate user navigation by only displaying subsets of data on visitor-facing pages and dynamically retrieving updates to isolated sections. More specifically, AJAX allows the browser to send requests to a web server, including data parameters, and to insert the data that it receives back in the web page, replacing or augmenting the original content.

This recipe shows how to add AJAX support to the bug tracking system created in *Chapter 7, Creating Custom MySQL Database Tables*.

Getting ready

You should have already followed the *Importing data from a user file into custom tables* recipe in *Chapter 7, Creating Custom MySQL Database Tables* to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch7-bug-tracker\ch7-bug-tracker-v8.php`) from the code bundle and rename the file to `ch7-bug-tracker.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch7-bug-tracker` directory and edit `ch7-bug-tracker.php`.
3. Locate the `ch7bt_shortcode_list` function and find the section where the SQL query is being prepared.
4. Add an extra line to the query (the highlighted line of code in the following code block) to show only open bugs (bugs with a `bug_status` field set to 0):

```
$bug_query = 'select * from ' . $wpdb->get_blog_prefix();  
  
$bug_query .= 'ch7_bug_data '  
$bug_query .= 'where bug_status = 0 ';
```

5. Make the changes highlighted in the following code, to the code building the search query:

```
if ( $search_mode ) {
    $search_term = '%' . $search_string . '%';
    $bug_query .= "and ( bug_title like '%s' ";
    $bug_query .= "or bug_description like '%s' ) ";
}
```

6. Find the code responsible for drawing the search form and add the following highlighted block of code after it to display a link to be clicked to show closed bugs:

```
$output .= '</form><br />';

$output .= '<a class="show_closed_bugs">';
$output .= 'Show closed bugs';
$output .= '</a>';

$output .= '<div class="bug_listing">';
```

7. Find the code that ends the bug table and add a closing div tag, as shown highlighted in the following line of code:

```
$output .= '</table></div><br />';
```

8. Insert this code segment after the bug display table to add the JavaScript responsible for providing the AJAX-based data replacement functionality:

```
$output .= "<script type='text/javascript'>";

$nonce = wp_create_nonce( 'ch8bt_ajax' );

$output .= "function replacecontent( bug_status )" .
    "{ jQuery.ajax( {" .
    "    type: 'POST'," .
    "    url: ajax_url," .
    "    data: { action: 'ch8bt_buglist_ajax'," .
    "        _ajax_nonce: '" . $nonce . "'," .
    "        bug_status: bug_status }," .
    "    success: function( data ) {" .
    "        jQuery('.bug_listing').html( data );" .
    "    }" .
    "    });" .
    "};";

$output .= "jQuery( document ).ready( function() {";
$output .= "jQuery('.show_closed_bugs').click( function()
    { replacecontent( 1 ); } ";
$output .= "));";
$output .= "</script>";
```

9. Add the following line of code at the end of the plugin file to register a function to add content to the page header:

```
add_action( 'wp_head', 'ch8bt_declare_ajaxurl' );
```

10. Append the following block of code to provide an implementation for the `ch8bt_declare_ajaxurl` function:

```
function ch8bt_declare_ajaxurl() { ?>
<script type="text/javascript">
    var ajax_url = '<?php echo admin_url( 'admin-ajax.php' ); ?>';
</script>
<?php }
```

11. Insert the following lines of code to register functions that will be called when AJAX requests are received from public or logged in users with an action variable set to `ch8bt_buglist_ajax`:

```
add_action( 'wp_ajax_ch8bt_buglist_ajax', 'ch8bt_buglist_ajax' );
add_action( 'wp_ajax_nopriv_ch8bt_buglist_ajax',
    'ch8bt_buglist_ajax' );
```

12. Add the following block of code to provide an implementation for the `ch8bt_buglist_ajax` function:

```
function ch8bt_buglist_ajax() {
    check_ajax_referer( 'ch8bt_ajax' );

    if ( isset( $_POST['bug_status'] ) &&
        is_numeric($_POST['bug_status'] ) ) {
        global $wpdb;

        // Prepare query to retrieve bugs from database
        $bug_query = 'select * from ' .
            $wpdb->get_blog_prefix();
        $bug_query .= 'ch7_bug_data where bug_status = ' .
            intval( $_POST['bug_status'] );
        $bug_query .= ' ORDER by bug_id DESC';

        $bug_items = $wpdb->get_results(
            $wpdb->prepare( $bug_query ), ARRAY_A );

        // Prepare output to be returned to AJAX requestor
        $output = '<div class="bug_listing"><table>';

        // Check if any bugs were found
        if ( $bug_items ) {
            $output .= '<tr><th style="width: 80px">ID</th>';
            $output .= '<th style="width: 300px">';
            $output .= 'Title / Desc</th><th>Version</th></tr>';
        }
    }
}
```

```

// Create row in table for each bug
foreach ( $bug_items as $bug_item ) {
    $output .= '<tr style="background: #FFF">';
    $output .= '<td>' . $bug_item['bug_id'] . '</td>';
    $output .= '<td>' . $bug_item['bug_title'] . '</td>';
    $output .= '<td>' . $bug_item['bug_version'];
    $output .= '</td></tr>';
    $output .= '<tr><td></td><td colspan="2">';
    $output .= $bug_item['bug_description'];
    $output .= '</td></tr>';
}
} else {
    // Message displayed if no bugs are found
    $output .= '<tr style="background: #FFF">';
    $output .= '<td colspan="3">No Bugs to
        Display</td>';
}

$output .= '</table></div><br />';

echo $output;
}

die();
}

```

13. Add the following line of code to register a function to be called when scripts are being queued up:

```
add_action( 'wp_enqueue_scripts', 'ch8bt_load_jquery' );
```

14. Insert the following code block to provide an implementation for the `ch8bt_load_query` function:

```
function ch8bt_load_jquery() {
    wp_enqueue_script( 'jquery' );
}

```

15. Save and close the plugin file.
16. Visit the bug listing page that was previously created to see that only opened bugs are displayed.

17. Click on the link to display closed bugs to see how the list gets replaced with closed issues.

Search bugs

[Show closed bugs](#)

ID	TITLE / DESC	VERSION
2		1.0

This is a first bug report stored in a new custom database table.

How it works...

AJAX page interactions are powered by JavaScript code and allow users to create pages with content that gets dynamically updated. To add this functionality to our bug tracking system, we start this recipe by modifying the existing shortcode bug query to only retrieve entries that have an open status (value of 0).

Once this is done, we move on to add two new elements to the initial shortcode output: a link to display closed bugs and a block of JavaScript code. The link itself is quite simple, containing a class name and a text label that visitors will be able to click. The JavaScript code is a bit more complex. Essentially, the script makes a request for the `replacecontent` function to be called when the `show_closed_bugs` link is clicked by visitors. In turn, the `replacecontent` function contains a single call to the jQuery `ajax` function. This function takes a number of arguments, starting with the type of operation, which is set to `POST`. This indicates that all variables sent in the request URL will be stored in a standard `$_POST` variable array.

The second parameter is the URL to which the request should be sent. The variable used here is defined in the header code that is generated by the `ch8bt_declare_ajaxurl` function and points to the WordPress `admin-ajax.php` script URL. While the name of this script starts with the word `admin`, it can also be used to process AJAX requests from visitor-facing pages.

After these first two arguments is a `data` array that contains a number of data elements, such as the name of the action, a nonce field to secure the request, and the status of the bugs that should be retrieved. Finally, the `success` parameter indicates that the data received back from the AJAX request should be used to replace the HTML content of the `.bug_listing` div section of the existing page.

To process this request, our plugin goes on to register the function `ch8bt_buglist_ajax` to be called when one of two variable name actions are matched: `wp_ajax_<actionname>` or `wp_ajax_nopriv_<actionname>`. In both cases, `<actionname>` is the string that was sent as part of the data parameters in the AJAX request. Upon receiving the request, the callback generates an updated bug table, echoes the resulting HTML code, and makes a call to the standard PHP `die()` function. While this last step might seem strange, it is needed to avoid having a trailing `1` at the end of the new HTML, indicating that AJAX processing was successfully performed by WordPress.

While the `ch8bt_buglist_ajax` function shares a lot of code with the existing `ch7bt_shortcode_list` function, it is easier to create a separate code block that only contains the necessary elements for this example. That being said, combining the two functions would make future updates to the table layout easier to maintain.

See also

- ▶ *Importing data from a user file into custom tables* recipe in Chapter 7, *Creating Custom MySQL Database Tables*

9

Adding New Widgets to the WordPress Library

In this chapter, we will learn how to create our own widget through the following topics:

- ▶ Creating a new widget in WordPress
- ▶ Displaying configuration options
- ▶ Validating configuration options
- ▶ Implementing the widget display function
- ▶ Adding a custom dashboard widget

Introduction

Widgets were introduced in WordPress Version 2.2. They allow users to easily populate sidebars or other areas of their website theme with blocks of content that are provided by WordPress itself (post or page data) or by any plugins that have been installed (for example, bug tracking system information). Looking at a WordPress installation, the default set of widgets include the **Archives** widget, which lists monthly post archives, and the **Links** widget, providing an easy way to display the links stored in your WordPress site.

Following its open design, WordPress provides functions that allow plugin developers to create new widgets that users will be able to add to their page design. This chapter shows how to use the widget class to create a custom widget. It also covers a second type of widget, the **Dashboard** widget, which can be used to display plugin-specific information on the front page of the administrative area.

Creating a new widget in WordPress

The first step in creating a custom widget is to define its name and indicate which class contains all of its implementation functions. Once the new element has been registered with the system, it will immediately appear in the widget list where users will be able to drag-and-drop it to their sidebars.

This recipe defines a new widget that displays recent book reviews from the custom post type category created in *Chapter 4, The Power of Custom Post Types*.

Getting ready

You should have already followed the *Updating page title to include custom post data using plugin filters* recipe from *Chapter 4, The Power of Custom Post Types* to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v8.php`) from the code bundle and activate the **Chapter 4 – Book Reviews** plugin.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch9-book-review-widget`.
3. Navigate to the directory and create a text file called `ch9-book-review-widget.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 9 – Book Review Widget`.
5. Add the following line of code before the closing `?>` PHP command to register a function to be called when widgets are initialized:

```
add_action( 'widgets_init', 'ch9brw_create_widgets' );
```

6. Add the following code segment to provide an implementation for the `ch9brw_create_widgets` function:

```
function ch9brw_create_widgets() {  
    register_widget( 'Book_Reviews' );  
}
```

7. Insert the following block of code to declare the `Book_Reviews` class, along with its constructor method:

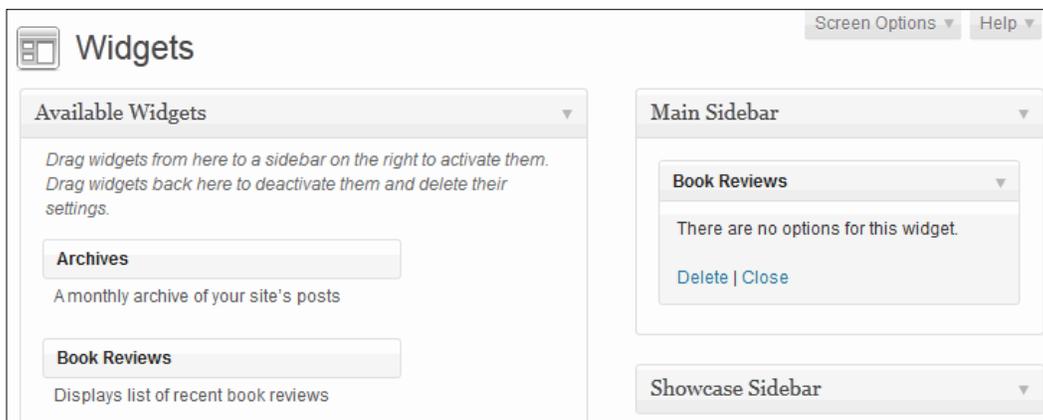
```
class Book_Reviews extends WP_Widget {  
    // Construction function  
    function __construct () {
```

```

parent::__construct( 'book_reviews', 'Book Reviews',
    array( 'description' =>
        'Displays list of recent book
        reviews' ) );
    }
}

```

8. Save and close the plugin file.
9. Navigate to the **Plugins** management page and activate the **Chapter 9 – Book Review Widget** plugin.
10. Visit the **Widgets** section of the **Appearance** administration page to see the newly-created **Book Reviews** widget appear as part of the list of **Available Widgets**.
11. Drag-and-drop the new widget to one of the available sidebars, listed on the right-hand side, to create a widget instance and see that the widget currently has no available options to configure it.



How it works...

The `widgets_init` action hook is used to register a function to be executed when widgets are being created by WordPress. When the callback occurs, we create a new widget by calling the simple `register_widget` function. As can be seen in the recipe, this function requires a single argument that indicates the name of the class that contains the widget definition.

The rest of the recipe declares the widget implementation class, which extends the WordPress `WP_Widget` class. While the class has many potential member functions, this recipe only defines the class constructor, which initializes the object instance by sending a unique identifier, a title, and a `parent` to the parent class. As with any other functions declared in plugins, it is important to give unique names to the widget class and widget identifier in order to avoid conflict with other plugins.

When the plugin is activated, users can see the new widget immediately and are able to add one or more instance of the new element as part of a sidebar's content. However, the new widget will not render anything other than an error message on website pages until its `widget` method is implemented in a later recipe in this chapter.

There's more...

As you may have noticed, this recipe creates a separate plugin file and directory from the main book review plugin created in *Chapter 4, The Power of Custom Post Types*.

Plugins extending other plugins

While we could have placed the widget creation code in the same file as the book review plugin, placing it in a separate file is just as valid. Some plugins distributed on the official `wordpress.org` repository actually use that technique to break up their functionality into more manageable code segments. The only thing to be careful with this technique is to be sure that all elements that a secondary plugin is dependent upon are loaded before referring to them in callback functions.

In this case, since widgets are created late in the WordPress initialization process, the custom post type that will be required by the widget will be available.

See also

- ▶ *Updating page title to include custom post data using plugin filters* recipe in *Chapter 4, The Power of Custom Post Types*

Displaying configuration options

Similar to the plugin configuration pages, widgets can have one or more options to allow users to specify how some aspects of the component will behave. These options can be configured individually for each instance of a widget that is added to a site layout. To handle all of the logistics around multiple possible widget instances, WordPress actually takes care of most of the data handling and storage tasks.

This recipe shows how to add a new method to the book review widget class to display configuration options.

Getting ready

You should have already followed the *Creating a new widget in WordPress* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch9-book-review-widget\ch9-book-review-widget-v1.php`) from the code bundle and rename the file to `ch9-book-review-widget.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch9-book-review-widget` directory and edit `ch9-book-review-widget.php`.
3. Find the `Book_Reviews` class and add the following block of code within the class to define the form method:

```
function form( $instance ) {
    // Retrieve previous values from instance
    // or set default values if not present
    $render_widget = ( !empty( $instance['render_widget'] ) ?
        $instance['render_widget'] : 'true' );

    $nb_book_reviews = ( !empty( $instance['nb_book_reviews'] ) ?
        $instance['nb_book_reviews'] : 5 );

    $widget_title = ( !empty( $instance['widget_title'] ) ?
        esc_attr( $instance['widget_title'] ) :
        'Book Reviews' );
?>

<!-- Display fields to specify title and item count -->
<p>
    <label for="<?php echo
        $this->get_field_id( 'render_widget' ); ?>">
    <?php echo 'Display Widget'; ?>
    <select
        id="<?php echo $this->get_field_id
        ( 'render_widget' ); ?>"
        name="<?php echo $this->get_field_name
        ( 'render_widget' ); ?>">
        <option value="true"
            <?php selected( $render_widget, 'true' ); ?>>
            Yes</option>
        <option value="false"
            <?php selected( $render_widget, 'false' ); ?>>
            No</option>
    </select>
</label>
</p>
<p>
    <label for="<?php echo
```

```
        $this->get_field_id( 'widget_title' ); ?>">
<?php echo 'Widget Title: '; ?>
<input type="text"
    id="<?php echo $this->get_field_id
    ( 'widget_title' ); ?>"
    name="<?php echo $this->get_field_name
    ( 'widget_title' ); ?>"
    value="<?php echo $widget_title; ?>" />
</label>
</p>
<p>
<label for="<?php echo
    $this->get_field_id( 'nb_book_reviews' ); ?>">
<?php echo 'Number of reviews to display: '; ?>
<input type="text"
    id="<?php echo $this->get_field_id
    ( 'nb_book_reviews' ); ?>"
    name="<?php echo $this->get_field_name
    ( 'nb_book_reviews' ); ?>"
    value="<?php echo $nb_book_reviews; ?>" />
</label>
</p>

<?php
}
```

4. Save and close the plugin file.
5. Refresh the **Appearance | Widgets** administration page and expand the **Book Reviews** widget instance to see the newly-created options.
6. Change the widget options and click on **Save** to update its configuration.



How it works...

When users create a new widget instance, WordPress automatically manages configuration options for that element using an array variable. It also calls the widget class' `form` method, if present, to render the widget instance's options in a configuration panel.

The first few lines of the `form` method check to see if the `instance` array contains options to specify whether the widget should be displayed, the number of book reviews to be shown, and the title that should be displayed at the beginning of the widget. If either of these options are missing, we use the PHP ternary conditional operator (`? :`) to assign default values to the `render_widget`, `nb_book_reviews`, and `widget_title` functions. This operator expects three expressions, ordered as follows: `(expr1) ? (expr2) : (expr3)`. It will then return `expr2` if `expr1` is true and `expr3` if it's false.

With these two variables in place, the rest of the `form` method's code uses a mix of HTML and PHP code to render the configuration fields that are shown in the widget editor. The `get_field_id` and `get_field_name` methods, seen throughout this code, are used to generate unique identifiers that will help WordPress to store data separately for all widget instances.

As can be seen in this recipe, the widget class is able to automatically process and save widget configuration parameters. However, it should be noted that allowing WordPress to handle this task by itself means that no validation will be performed on the data entered. This could cause problems if a user enters text instead of the number of reviews to be displayed. The next recipe shows how to handle data validation.

See also

- ▶ [Creating a new widget in WordPress recipe](#)

Validating configuration options

The widget configuration panel that was put in place in the previous recipe was functional, allowing users to change options and save updated values to the site database. That being said, all WordPress does by default when the user saves a widget is to store values directly to the site database. Since accepting user data blindly can lead to functionality problems and security risks if wrong or malicious values are entered, it is preferable to add data validation rules through the creation of an `update` method that will be able to verify configuration data before it is saved. This recipe shows how to implement a widget's `update` method.

Getting ready

You should have already followed the *Displaying configuration options* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch9-book-review-widget\ch9-book-review-widget-v2.php`) from the code bundle and rename the file to `ch9-book-review-widget.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch9-book-review-widget` directory and edit `ch9-book-review-widget.php`.
3. Find the `Book_Reviews` class and add the following block of code within the class to define the `update` method:

```
function update( $new_instance, $old_instance ) {
    $instance = $old_instance;

    // Only allow numeric values
    if ( is_numeric ( $new_instance['nb_book_reviews'] ) )
        $instance['nb_book_reviews'] =
            intval( $new_instance['nb_book_reviews'] );
    else
        $instance['nb_book_reviews'] = $instance['nb_book_reviews'];

    $instance['widget_title'] =
        strip_tags( $new_instance['widget_title'] );

    $instance['render_widget'] =
        strip_tags( $new_instance['render_widget'] );

    return $instance;
}
```

4. Save and close the plugin file.
5. Visit the **Widgets** section of the **Appearance** administration page and expand the **Book Reviews** widget instance.
6. Enter a textual value in the **Number of reviews to display** field and save the widget. You will see that the field's value reverts to the last valid number saved for this field.

How it works...

The `update` method receives two arrays of data and must return a single array to be saved to the site database. The two incoming arrays contain the new option values entered by the user and the values that were previously stored for the widget, respectively.

To start from known values, the method's implementation starts by making a copy of the old values to a new variable called `$instance`. It follows this initialization by calling the standard PHP `strip_tags` function on each user-entered value to remove potential HTML or PHP tags, saving the return value in the `$instance` array. It also calls the PHP `is_numeric` and `intval` function on the entry indicating the number of reviews to be displayed to make sure that it's a numeric value. If anything other than a number was entered, the previous field value will be saved and displayed back to the user.

See also

- ▶ *Displaying configuration options* recipe

Implementing the widget display function

For all of the widget creation work that we have done so far, our new creation does not display any content on the website yet. When displaying an area that contains widgets, WordPress tries to call a method named `widget` for each user-selected widget to output the desired content to the browser.

This recipe shows how to implement a `widget` method to display a list of recent book reviews when the widget is instantiated in a sidebar.

Getting ready

You should have already followed the *Validating configuration options* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch9-book-review-widget\ch9-book-review-widget-v3.php`) from the code bundle and rename the file to `ch9-book-review-widget.php`.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Navigate to the `ch9-book-review-widget` directory and edit `ch9-book-review-widget.php`.

3. Find the `Book_Reviews` class and add the following block of code within the class to define the widget method:

```
function widget( $args, $instance ) {
    if ( $instance['render_widget'] == 'true' ) {
        // Extract members of args array as individual variables
        extract( $args );

        // Retrieve widget configuration options
        $nb_book_reviews =
            ( !empty( $instance['nb_book_reviews'] ) ?
              $instance['nb_book_reviews'] : 5 );

        $widget_title = ( !empty( $instance['widget_title'] ) ?
            esc_attr( $instance['widget_title'] ) :
            'Book Reviews' );

        // Preparation of query string to retrieve book reviews
        $query_array = array( 'post_type' => 'book_reviews',
            'post_status' => 'publish',
            'posts_per_page' =>
                $nb_book_reviews );

        // Execution of post query
        $book_review_query = new WP_Query();
        $book_review_query->query( $query_array );

        // Display widget title
        echo $before_widget;
        echo $before_title;
        echo apply_filters( 'widget_title', $widget_title );
        echo $after_title;

        // Check if any posts were returned by query
        if ( $book_review_query->have_posts() ) {
            // Display posts in unordered list layout
            echo '<ul>';

            // Cycle through all items retrieved
            while ( $book_review_query->have_posts() ) {
                $book_review_query->the_post();
                echo '<li><a href="' . get_permalink() . ">";
                echo get_the_title( get_the_ID() ) . '</a></li>';
            }

            echo '</ul>';
        }

        wp_reset_query();
        echo $after_widget;
    }
}
```

4. Save and close the plugin file.
5. Visit the website's front page to see the newly-added widget contents displayed in the sidebar.



How it works...

Similar to action hooks that we have seen in the earlier chapters, the `widget` method is meant to directly output HTML code to the browser that will be displayed when an instance of the new widget has been created in a sidebar.

The `widget` method starts by checking whether or not the widget should be displayed. If it should, it continues by calling the standard PHP `extract` function on the first parameter received, an array named `$args`. Calling this function parses the array and creates variables for each element found, making it easier for the following code to access the elements that should be placed before and after the widget title and widget content.

After this initial statement, the recipe continues by retrieving the number of items to display and the widget title from the `$instance` array, which has been received as the second method parameter using the same technique that was shown when implementing the `form` method.

The rest of the code is very similar to the book review shortcode created in *Chapter 4, The Power of Custom Post Types* (displaying custom post type data in shortcodes), where we assemble a query string that indicates the type and maximum quantity of data that we want to retrieve from the database. The resulting query is executed by creating a new instance of the WordPress `WP_Query` object. If results are found, the following recipe code cycles through all entries and outputs code to render an unordered list of all items found. Last but not least, the recipe formats the widget content by outputting the values of the `$before_widget`, `$after_widget`, `$before_title`, `$after_title` variables, and user-specified widget title in the right places.

See also

- ▶ [Creating a new widget in WordPress recipe](#)

Adding a custom dashboard widget

While widgets are primarily used by website administrators to easily add content to their front-facing websites, WordPress contains another type of widget that plugin developers can use to enhance user experience. Dashboard plugins are sections that appear on the front page of a site's administration area. These sections can offer any kind of functionality, from simple information displays indicating how much data is stored in a plugin to forms that allow site administrators to quickly perform configuration tasks.

This recipe shows how to add a new Dashboard widget that indicates how many book reviews are stored in the system, along with links to quickly access them.

Getting ready

You should have already followed the *Updating page title to include custom post data using plugin filters* recipe from *Chapter 4, The Power of Custom Post Types* to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v8.php`) from the code bundle and activate the **Chapter 4 – Book Reviews** plugin.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch9-book-review-dashboard-widget`.
3. Navigate to the directory and create a text file called `ch9-book-review-dashboard-widget.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 9 – Book Review Dashboard Widget`.
5. Add the following line of code before the closing `?>` PHP command to register a function to be called when the dashboard contents are being prepared:

```
add_action( 'wp_dashboard_setup',  
           'ch9brdw_add_dashboard_widget' );
```

6. Add the following code segment to provide an implementation for the `ch9brdw_add_dashboard_widget` function:

```
function ch9brdw_add_dashboard_widget() {
    wp_add_dashboard_widget( 'book_reviews_dashboard_widget',
                            'Book Reviews',
                            'ch9brdw_dashboard_widget' );
}
```

7. Insert the following block of code to implement the `ch9brdw_dashboard_widget` function declared in the previous step:

```
function ch9brdw_dashboard_widget() {
    $book_review_count = wp_count_posts( 'book_reviews' );
    ?>
    <a href="<?php echo add_query_arg( array(
                                    'post_status' => 'publish',
                                    'post_type' => 'book_reviews' ),
                                    admin_url( 'edit.php' ) ); ?>">
        <strong>
            <?php echo $book_review_count->publish; ?>
        </strong> Published
    </a>
    <br />
    <a href="<?php echo add_query_arg( array(
                                    'post_status' => 'draft',
                                    'post_type' => 'book_reviews' ),
                                    admin_url( 'edit.php' ) ); ?>">
        <strong>
            <?php echo $book_review_count->draft; ?>
        </strong> Draft
    </a>
    <?php }
```

8. Save and close the plugin file.
9. Navigate to the **Plugins** management page and activate the **Chapter 9 – Book Review Dashboard Widget** plugin.

10. Navigate to the site's **Dashboard** to see the new **Book Reviews** widget at the bottom of the page.



How it works...

Any plugin can register its own dashboard widget when WordPress is putting together content for this administrative landing page. After registering a function to be called during the Dashboard set up phase, our recipe makes a call to the `wp_add_dashboard_widget` function to add our own element to the site when the callback is executed. The `wp_add_dashboard_widget` function requires three parameters that need to provide a unique identifier for the new item, a title to be displayed at the top of the widget, and a function that will be responsible for generating the widget's contents. The `wp_add_dashboard_widget` function also has an optional fourth parameter that can be used when the widget needs to process form data as part of the Dashboard widget contents.

As can be seen in the previous screenshot, Dashboard widgets are displayed using WordPress meta boxes, where any HTML code echoed by the content display function directly appears in the box.

While the display function is mostly composed of HTML code, we also make a call to the `wp_count_posts` utility function, which easily returns the number of posts for a given post type.

The new widget can be hidden and moved to a new location on the Dashboard, like any other built-in widget. Just like the front-facing widget plugin created earlier in this chapter, it should be noted that all code in this plugin is in a separate file than the original book review plugin, to organize its code separately from the original plugin file created in *Chapter 4, The Power of Custom Post Types*.

See also

- ▶ *Updating page title to include custom post data using plugin filters* recipe in *Chapter 4, The Power of Custom Post Types*

10

Enabling Plugin Internationalization

In this chapter, we will learn about plugin localization through the following topics:

- ▶ Changing the WordPress language configuration
- ▶ Adapting default user settings for translation
- ▶ Making admin page code ready for translation
- ▶ Modifying shortcode output for translation
- ▶ Translating text strings using Poedit
- ▶ Loading a language file in the plugin initialization

Introduction

WordPress is a worldwide phenomenon, with users embracing the platform all around the globe. To create a more specific experience for users in different locales, WordPress offers the ability to translate all of its user and visitor-facing content, resulting in numerous localizations becoming available for download online. Like most other functionalities in the platform, internationalization is also available to plugin developers through a set of easy-to-use functions. The main difference being that plugin translations are typically included with the extension, instead of being downloaded separately as is the case with WordPress.

To prepare their plugin to be localized, developers must use special internationalization functions when dealing with text elements. Once this structure is in place, any user can create localizations by themselves for languages that they know and submit them back to the plugin author for inclusion in a future update to the extension.

This chapter explains how to prepare a plugin to be translated and shows how to use the Poedit tool to create a new language file for a simple plugin.

Changing the WordPress language configuration

The first step to translating a plugin is to configure WordPress to a different language setting other than English. This will automatically trigger mechanisms in the platform to look for alternate language content for any internationalized string.

In this recipe we will set the site to **French**.

Getting ready

You should have access to a WordPress development environment.

How to do it...

1. Navigate to the root of your WordPress installation.
2. Open the file called `wp-config.php` in a code editor.
3. Change the line that declares the site language from `define('WPLANG', '');` to `define('WPLANG', 'fr_FR');`.
4. Save and close the configuration file.

How it works...

Whenever WordPress renders a page for visitors or site administrators, it executes the contents of the `wp-config.php` file, which declares a number of site-wide constants. One of these constants is the site language. By default, this constant has no value, indicating that WordPress should display all content in U.S. English. If defined, the system tries to find a translation file under the `wp-content/languages` or `wp-includes/languages` directories of the site to locate translation strings for the target language. In this case, it will try to find a file called `fr_FR.mo`. While it will not actually find this file in a default installation, setting this configuration option will facilitate the creation and testing of a plugin translation file in later recipes.

To learn more about translation files and find out where to download them from, visit the WordPress Codex available at http://codex.wordpress.org/WordPress_in_Your_Language.

Adapting default user settings for translation

As mentioned in the introduction, plugin code needs to be specifically written to allow text items to be translated. This work starts in the plugin's activation routine, where default plugin option values are set, to find alternate values when a language other than English is specified in the site's configuration file.

This recipe shows how to assign a translated string to a plugin's default options array on initialization.

Getting ready

You should have already followed the *Changing the WordPress language configuration* recipe to have a specified translation language for the site.

How to do it...

1. Navigate to the WordPress plugin directory of your development installation.
2. Create a new directory called `ch10-hello-world`.
3. Navigate to the directory and create a text file called `ch10-hello-world.php`.
4. Open the new file in a code editor and add an appropriate header at the top of the plugin file, naming the plugin `Chapter 10 - Hello World`.
5. Add the following line of code before the plugin's closing `?>` PHP command to register a function to be called when the plugin is activated:

```
register_activation_hook( __FILE__,
                        'ch10hw_set_default_options_array' );
```

6. Insert the following block of code to provide an implementation for the `ch10hw_set_default_options_array` function:

```
function ch10hw_set_default_options_array() {
    if ( false === get_option( 'ch10hw_options' ) ) {
        $new_options = array();
        $new_options['default_text'] =
            __( 'Hello World', 'ch10hw_hello_world' );

        add_option( 'ch10hw_options', $new_options );
    }
}
```

7. Save and close the plugin file.

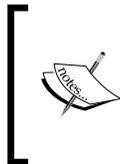
8. Navigate to the **Plugins** management page and activate the **Chapter 10 – Hello World** plugin.
9. Using phpMyAdmin or the NetBeans IDE, find the `options` table entry where the `option_name` field has a value of `ch10hw_options` to see the newly-created option.

#	option_id	option_name	option_value	autoload
1	298	ch10hw_options	a:1:{s:12:"default_text";s:11:"Hello World";}	yes

How it works...

The `__` function (that's two underscores) is a WordPress utility function that tries to find a translation for the text that it receives in its first argument, within the text domain specified in the second argument. A text domain is essentially a subsection of the global translation table that is managed by WordPress. In this example, the text to be translated is the string `Hello World`, for which the system tries to find a translation in the `ch10hw_hello_world` domain. Since this domain is not available at this time, the function returns the original string that it received as its first parameter. The plugin code assigns the value it receives to the default configuration array.

It should be noted that the `__` function is actually an alias for the `translate` function. While both functions have the same functionality, using `__` makes the code shorter when it contains a lot of text elements to be translated.



While it may be tempting for developers to use a variable or constant in the first parameter of the `__` function if they need to display the same text multiple times, this should not be done as it will cause problems with the translation lookup mechanism.

See also

- ▶ [Changing the WordPress language configuration recipe](#)

Making admin page code ready for translation

While the previous recipe showed how to look up the translation of a text item and return its value for further processing in the plugin code, there are many instances where it is more practical to display the translated content immediately.

This recipe shows how to translate the contents of a simple administration page for immediate display.

Getting ready

You should have already followed the *Adapting default user settings for translation* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code for that recipe from the code bundle. You should rename the file `ch10-hello-world\ch10-hello-world-v1.php` to `ch10-hello-world.php` before starting.

How to do it...

1. Navigate to the `ch10-hello-world` folder of the WordPress plugin directory of your development installation.
2. Open the `ch10-hello-world.php` file in a text editor.
3. Add the following line of code at the end of the file to register a function to be called when WordPress is building the administration pages menu:

```
add_action( 'admin_menu', 'ch10hw_settings_menu' );
```

4. Add the following code section to provide an implementation for the `ch10hw_settings_menu` function:

```
function ch10hw_settings_menu() {
    add_options_page(
        __( 'Hello World Configuration', 'ch10hw_hello_world' ),
        __( 'Hello World', 'ch10hw_hello_world' ),
        'manage_options',
        'ch10hw-hello-world', 'ch10hw_config_page' );
}
```

5. Insert the following block of code to create the `ch10hw_config_page` function, declared in the call to `add_options_page`:

```
function ch10hw_config_page() {
    $options = get_option( 'ch10hw_options' );
    ?>

    <div id="ch10hw-general" class="wrap">
    <!-- Echo translation for "Hello World" to the browser -->
    <h2><?php _e( 'Hello World', 'ch10hw_hello_world' ); ?></h2>

    <form method="post" action="admin-post.php">

    <input type="hidden" name="action"
        value="save_ch10hw_options" />

    <?php wp_nonce_field( 'ch10hw' ); ?>
```

```
<!-- Echo translation for "Hello World" to the browser -->
<?php _e( 'Default Text', 'ch10hw_hello_world' ); ?>:
<input type="text" name="default_text"
      value="<?php echo esc_html( $options['default_text']
); ?>"/>
<br />

<input type="submit" value="<?php _e( 'Submit',
      'ch10hw_hello_world' ); ?>" class="button-primary"/>
</form>
</div>
<?php }
```

6. Add the following line of code to register a function to be executed when the administration panel is being prepared to be displayed:

```
add_action( 'admin_init', 'ch10hw_admin_init' );
```

7. Append the following code segment to provide an implementation for the `ch10hw_admin_init` function:

```
function ch10hw_admin_init() {
    add_action( 'admin_post_save_ch10hw_options',
                'process_ch10hw_options' );
}
```

8. Provide code for the `process_ch10hw_options` function, declared in the previous step, by inserting the following code:

```
function process_ch10hw_options() {
    if ( !current_user_can( 'manage_options' ) )
        wp_die( 'Not allowed' );

    check_admin_referer( 'ch10hw' );

    $options = get_option( 'ch10hw_options' );

    $options['default_text'] = $_POST['default_text'];

    update_option( 'ch10hw_options', $options );
    wp_redirect( add_query_arg( 'page', 'ch10hw-hello-world',
                                admin_url( 'options-general.php' ) ) );
    exit;
}
```

9. Save and close the plugin file.

10. Navigate to the administration page of your development WordPress installation.
11. Click on the **Settings** section in the left-hand navigation menu to expand it. You will see a new menu item called **Hello World** in the tree. Selecting the new entry displays the plugin's simple configuration form, as shown in the following screenshot:



The screenshot shows a simple configuration form for a plugin titled "Hello World". It features a text input field labeled "Default Text" with the value "Hello World" entered. Below the input field is a blue button with the text "Submit".

How it works...

This recipe makes use of the `__` function, covered in the previous recipe, along with the `_e` function. This second function's purpose is similar to `__`, except that it immediately echoes the outcome of the translation lookup to the browser. It should be used for all text elements that would previously have just been simple text in HTML code. Of course, making a call to this function requires the presence of standard opening and closing PHP tags (`<?>` and `?>`) to be executed amongst the surrounding HTML.

The rest of this plugin's code takes care of storing user updates to the site database, as covered previously in *Chapter 3, User Settings and Administration Pages*.

See also

- ▶ *Adapting default user settings for translation recipe*

Modifying shortcode output for translation

As we have seen in numerous recipes, shortcodes are powerful tools that provide an easy way for users to add content to their site posts and pages. Since this content is presented to users, it can benefit from a translation just as much as the site's administration pages.

This recipe shows how to translate shortcode output before it is displayed. It also explains how to deal with variable data elements that can be positioned differently between languages.

Getting ready

You should have already followed the *Making admin page code ready for translation* recipe entitled to have a starting point for this recipe. Alternatively, you can get the resulting code for that recipe from the code bundle. You should rename the file `ch10-hello-world\ch10-hello-world-v2.php` to `ch10-hello-world.php` before starting.

How to do it...

1. Navigate to the `ch10-hello-world` folder of the WordPress plugin directory of your development installation.
2. Open the `ch10-hello-world.php` file in a text editor.
3. Add the following line of code at the end of the file to declare a new shortcode that will be available to content authors:

```
add_shortcode( 'hello-world', 'ch10hw_hello_world_shortcode' );
```

4. Add the following code section to provide an implementation for the `ch10hw_hello_world_shortcode` function:

```
function ch10hw_hello_world_shortcode() {  
    $options = get_option( 'ch10hw_options' );  
    $output = sprintf( __( 'The current text string is: %s.',  
        'ch10hw_hello_world' ),  
        $options['default_text' ] );  
    return $output;  
}
```

5. Save and close the plugin file.
6. Create a new page and insert the new shortcode `[hello-world]` in the content.
7. View the page to see the output of the shortcode.

Hello World

The default text string is: Hello World.

How it works...

This recipe shows something that's a bit more complex than the previous two, as we want the shortcode output to be a combination of static text with a dynamic element, and we want that element to appear in different places based on the grammatical structure of the target language. The way to achieve this functionality is to combine the `__` internationalization function with the `sprintf` standard PHP function.

The purpose of the `sprintf` function is to insert a variable in a string. It performs this task by looking for a placeholder in the target string sent in the first argument, and replaces it with the variable that it receives as its second argument. Some examples of placeholders are `%s` for a string and `%d` for an integer. With this functionality in mind, we use a placeholder as part of the string to be translated so that users who create localization files can choose where the value will be placed as part of the sentence structure. Once the translation has been obtained by the `__` function, we can immediately send the alternate language string to `sprintf` to create the final text.

See also

- ▶ *Adapting default user settings for translation recipe*

Translating text strings using Poedit

After inserting all the necessary code to look up translations for text elements, we need to create the actual translation files. While there are multiple tools available to perform this task, we will focus our efforts around the most popular one, the free multi-platform Poedit.

This recipe shows how to extract all strings to be translated from the plugin's code using Poedit, translate them, and save the resulting language file under the plugin directory.

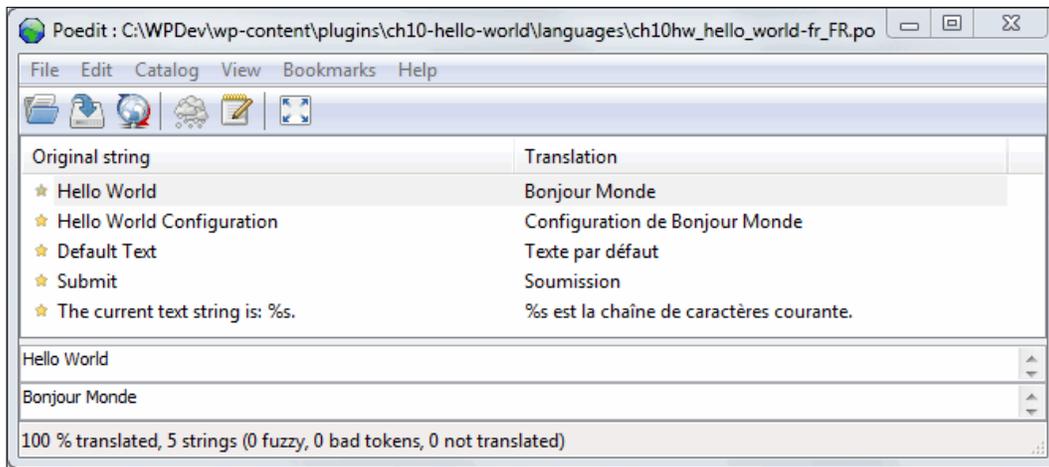
Getting ready

You should have already followed the *Modifying shortcode output for translation* recipe to have a starting point for this recipe. Alternatively, you can get the resulting code for that recipe from the code bundle. You should rename the file `ch10-hello-world\ch10-hello-world-v3.php` to `ch10-hello-world.php` before starting.

How to do it...

1. Navigate to the `ch10-hello-world` folder of the WordPress plugin directory of your development installation.
2. Create a new subdirectory named `languages`.
3. Navigate to the Poedit download page and download the appropriate version of the tool for your computer (<http://www.poedit.net/download.php>).
4. Install and start the Poedit application.
5. Select the **New Catalog...** item under the application's **File** menu.
6. Set the **Project name and version** field to `Hello World` under the **Project Info** tab.
7. Switch to the **Paths** tab.
8. Create a new entry in the **Paths** list by pressing the **New item** button.
9. Set the value of the new path entry to `..` (two period characters).
10. Switch to the **Keywords** tab.
11. Select and delete the **gettext** and **gettext_noop** entries in the **Keywords** list.
12. Select the remaining entry (`_`), click on the **Edit item** button, then change its value to `__` (two underscores instead of a single one).
13. Click on the **Add Item** button and create a second entry with a value of `_e`.

14. Click on the **OK** button to close the **Settings** dialog.
15. In the **Save As...** dialog that automatically comes up, navigate to the newly-created `languages` folder under the plugin's directory and set the target filename to `ch10hw_hello_world-fr_FR.po`.
16. Click on **OK** in the **Update summary** dialog to acknowledge that five new strings were found in the plugin's source code.
17. Select the items one by one in order to display them in the lower section of the window.
18. Enter a translation for each text element in the lower dialog box. The following screenshot shows the translations of each item to French:



19. Save the translation file once completed.

How it works...

The Poedit tool searches through PHP files, looking for functions that have specific names, as specified in the **Keywords** configuration section. It looks through all files located in the same directory as the catalog itself and in any additional folders specified under the **Paths** section of the catalog settings. By specifying `..` as an additional path, we tell Poedit to look one directory up from the `languages` folder, where the plugin files are located.

Based on the configuration that we specified, Poedit is able to find all instances of the `__` and `_e` functions in the plugin code and retrieve the text strings that are set as the first argument to these functions. Once all strings have been found, Poedit provides a simple interface to provide translations for each string and save the resulting translation file. Upon saving, Poedit actually creates two files. The first, with a `.po` extension, is a simple text file that contains a flat textual version of the original strings and the localized versions of each item. The second, sporting a `.mo` extension, is a file that is optimized for quick access on the web server.

The name of the language files is made from two parts: the name of the text domain, `ch10hw_hello_world`, which was used in all of our calls to the `__` and `_e` functions in the previous recipes, and the target language code, `fr_FR`, to match the language configuration that we set earlier in this chapter.

There's more...

If you are only comfortable with English, create a template file that users will be able to import to start their translation.

Translation template file

When you are only familiar with English, you can create a translation template that only contains the text to be translated by saving the catalog as a `.pot` file, instead of a `.po/.mo` combination. In addition to the special extension, the filename should not contain a language tag (for example, `ch10hw_hello_world.pot`).

See also

- ▶ *Adapting default user settings for translation recipe*

Loading a language file in the plugin initialization

The final step to plugin translation is to put the code in place to load a translation file. This is done by registering an action hook callback and calling a single function when it gets executed.

This recipe shows how to load the translation file created in the previous recipe.

Getting ready

You should have already followed the *Making admin page code ready for translation* and *Translating text strings using Poedit* recipes to have the proper files required for this recipe. Alternatively, you can get the resulting code for these recipes from the code bundle. You should rename the file `ch10-hello-world\ch10-hello-world-v3.php` to `ch10-hello-world.php` and copy the `languages` folder to the right location before starting.

How to do it...

1. Navigate to the `ch10-hello-world` folder of the WordPress plugin directory of your development installation.
2. Open the `ch10-hello-world.php` file in a text editor.
3. Add the following line of code at the end of the file to register a function to be called when the plugin is initialized:

```
add_action( 'init', 'ch10hw_plugin_init' );
```

4. Add the following code section to provide an implementation for the `ch10hw_plugin_init` function:

```
function ch10hw_plugin_init() {  
    load_plugin_textdomain( 'ch10hw_hello_world',  
                            false,  
                            dirname( plugin_basename( __FILE__ ) )  
                            . '/languages' );  
}
```

5. Save and close the plugin file.
6. Navigate to the **Settings** menu to see if the plugin's menu item has changed.
7. Select the **Bonjour Monde** item to see the translated configuration page.



How it works...

The `load_plugin_textdomain` function has three arguments. When called, it looks in the folder specified in the last parameter for a `.mo` file with a name starting with the text domain specified in the first parameter, followed by the current language set in the WordPress configuration file. If found, the translation file is loaded in memory and is used to search for translations every time the `__` or `_e` functions are encountered during execution. The middle argument, set to a `false` value, is obsolete but is still needed for backward compatibility.

Once all hooks are in place in the plugin code, and a first translation file (or template) is made available with the plugin, users can easily modify text elements to other languages, which they can use immediately. They can also provide these new translations back to the plugin author for inclusion in future updates.

There's more...

As a plugin evolves over time, new text items may need to be translated. There may also be a need to use more advanced translation functions and translate JavaScript code.

Updating a translation file

When new calls to the `__` or `_e` functions are made in a plugin, the translation file needs to be updated to take new text elements into account. To do this, start the Poedit tool and open the existing catalog. Then, select the **Update from sources** item under the **Catalog** menu. This will extract all text items and identify new entries. Once this is done, new items can be translated and saved back to the catalog file.

Advanced translation functions

While we used the most common internationalization functions in this chapter, there are a few more advanced functions that may be useful in your efforts:

- ▶ `_n($singular, $plural, $number, $domain)`: This function will look up one of the first two strings received, depending on whether the number is one or more.
- ▶ `_x($text, $context, $domain)`: Adds a parameter to the localization lookup to add a context parameter. This is useful when dealing with words that have the same spelling but different meanings.
- ▶ `_ex($text, $context, $domain)`: Same as `_x` but echoes the result of the lookup.
- ▶ `_nx($singular, $plural, $number, $context, $domain)`: Same as `_n` with the additional context parameter from `_x`.

There are also a number of functions that will perform a localization lookup immediately followed by the escape of the resulting string. These functions include `esc_attr__()`, `esc_attr_e()`, `esc_html__()`, `esc_html_x()`, and many more. For a full list of internationalization functions, visit <http://codex.wordpress.org/L10n>.

Localizing JavaScript files

JavaScript files are a bit more tricky to translate as they are often read from an external file that cannot contain any PHP code. The solution to this is the `wp_localize_script` function. When called, this function declares new variables in scripts that have already been queued up to be loaded and populates these variables with localized strings. Upon execution, the code will be able to access and display the proper text on-screen. The following code snippet is an example showing how to use the function:

```
wp_enqueue_script( 'script_handle' );
$translation_vars = array( 'string_val' =>
    __( 'Text to be translated' ) );
wp_localize_script( 'script_handle', 'javascript_object',
    $translation_vars );
```

In the previous code example , a new object called `javascript_object` will be created inside the `script_handle` script, with a data member called `string_val` that contains a translation of the target text in the current WordPress language, if available.

See also

- ▶ *Translating text strings using Poedit recipe*

11

Distributing Your Plugin on `wordpress.org`

In this chapter, we will discuss how to distribute your creations, covering the following topics:

- ▶ Creating a readme file for your plugin
- ▶ Applying for your plugin to be hosted on `wordpress.org`
- ▶ Uploading your plugin using Subversion
- ▶ Providing a plugin banner image

Introduction

Once you have a version of your new plugin that is ready to be distributed to the masses, you need to decide if you will join the official WordPress repository or self-publish it.

In most cases, the preferred option is to add your new extension to the official WordPress plugin repository, where you have many benefits, including free hosting, the ability for users to be notified of new updates, and a powerful search engine that users can access on `wordpress.org` or from the **Plugins** section of their site's administration pages. Other benefits of hosting on the official repository include download statistics and the creation of a free forum to facilitate user support. To qualify for this hosting, your work must be open source and must comply with the GNU General Public License, Version 2 (also known as GPL v2), a common open source software license that WordPress itself uses. To learn more about the GPL v2 license, visit <http://www.gnu.org/licenses/gpl-2.0.html>.

In comparison, self-hosting gives you full control over pricing, distribution license, and general presentation of your work, but it makes it harder for people to find your plugin and relies on implementing a custom update notification mechanism yourself or using third-party libraries or having users manually download updates when available.

Before making your plugin public, you should be sure that you are ready to deal with user feedback and questions. Once your creation is available for download, WordPress site administrators will quickly download it, install it, and may find that your work covers most, but not all, of their needs. When this happens, you will start getting requests to add functionality. This interaction with users is usually a great experience that can bring new ideas to the table that will enhance your work, but you should also be ready to accept criticism and invest time to fix issues and implement new features. You also need to think of the time that will be involved in testing your extension against new versions of WordPress, which typically come out two to three times a year.

This chapter explains how to prepare your work to be uploaded to the official plugin repository, including the application for an account, the actual submission using Subversion, and how to customize your plugin page to give it a very unique look.

Creating a readme file for your plugin

If you look at any plugins on the official WordPress repository, you will see that their page contains a lot of information, including a description of the extension, a list of frequently asked questions, and installation notes. As you may have noticed from the work that we have done so far, this data does not reside in the main plugin's code file. Instead, the official WordPress repository looks for this information in a specially formatted `readme.txt` file that needs to be included with the plugin.

This recipe shows how to create a `readme.txt` file for the Book Reviews plugin that we created in *Chapter 4, The Power of Custom Post Types*.

Getting ready

You should have already followed the *Updating page title to include custom post data using plugin filters* recipe from *Chapter 4, The Power of Custom Post Types* to have a starting point for this recipe. Alternatively, you can get the resulting code (`ch4-book-reviews\ch4-book-reviews-v8.php`) for that recipe from the code bundle.

How to do it...

1. Navigate to the `ch4-book-reviews` folder of your WordPress plugins directory.
2. Create a new text file named `readme.txt` and open it in a code editor.

3. Insert the following text in the file:

```
=== Book Reviews ===
Contributors: ylefebvre
Donate link: http://ylefebvre.ca/wordpress-plugins/book-reviews
Tags: book, reviews
Requires at least: 3.0
Tested up to: 3.4
Stable tag: trunk
```

Create your own book review web site!

```
== Description ==
```

This plugin lets you add a book review system to your WordPress site. Using custom post types, administrators will be able to create and edit book reviews to be published on your site.

```
== Installation ==
```

1. Download the plugin
1. Upload the book-reviews folder to your site's wp-content/plugins directory
1. Activate the plugin in the WordPress Admin Panel
1. Start creating new book reviews!
1. View the resulting list of reviews by accessing /book-reviews on your site.

```
== Changelog ==
```

```
= 1.0 =
```

```
* First version of the plugin.
```

```
== Frequently Asked Questions ==
```

There are currently no FAQs at this time.

```
== Screenshots ==
```

1. The review edition page

4. Save and close the text file.
5. Navigate to the **Book Reviews** edition page and take a screenshot using a third-party screen capture tool or your operating system's built-in function.
6. Save the resulting image as `screenshot-1.jpg` in the plugin directory.

How it works...

The `readme.txt` file uses a wiki-like syntax, with the number of equal signs (=) indicating the level of each section header. The first and most important section is the header, which contains important information such as the plugin's name, the author's wordpress.org username, donation link, search tags, supported versions, along with a one-line description of its functionality. This last item will always be visible as users navigate through your plugin's pages.

The initial header is followed by multiple sections, which correspond to the various tabs that appear within a plugin's display pages. More specifically, these sections contain a complete description of the extension's capability, a step-by-step guide to install and use your work, a change log containing a list of all versions with a summary of changes for each of them, frequently-asked questions, and screenshots. It is also possible for plugin authors to create their own arbitrary section using the same syntax.

As with standard wiki syntax, the repeating `1.` in front of each installation step will be converted to incrementing values when the system displays these bullets using an ordered list on the live site. Finally, if screenshots are listed in the `readme.txt` file, the wordpress.org site will search for files whose name starts with the keyword `screenshot-`, followed by a number corresponding to the values listed in the screenshot section, and display them with the associated text as a legend. When taking screenshots of your plugin in action, make sure that they are clear and meaningful as visitors will often decide if they will download your creations based on these images.

There's more...

To keep plugin code files more organized and have complete control over releases, you should consider using the Subversion tags.

Releasing specific plugin versions using tags

Tags are a Subversion concept that allow developers to identify a group of files at a specific point in time and label them with a name. This name can be used to specify the version of your plugin that wordpress.org visitors will be able to download. While this recipe specifies a value of `trunk` as the `Stable Tag`, indicating that the latest version of the files uploaded to the plugin's `trunk` folder will be released, it's possible to indicate any other tag name in this field. In addition to keeping your work more organized, working with tags allows you to commit partially implemented new plugin features to your repository without having them automatically available for all to access.

Applying for your plugin to be hosted on wordpress.org

After creating proper documentation for your creation, the next step towards its publication on the official plugin repository is to apply for hosting. This is simply done by submitting a request form in the Developer Center section on `wordpress.org`.

This recipe shows how to apply for plugin hosting and offers tips to follow for quick acceptance.

How to do it...

1. Point your web browser to the plugin hosting request form page that is available at `http://wordpress.org/extend/plugins/add/`.
2. Log in to the `wordpress.org` website using the form at the top of the plugin submission page with your existing credentials or create a new account if you don't currently have one.
3. Fill in the **Plugin Name** and **Plugin Description** fields.
4. Optionally, provide the address of a page on your own site where additional information on the plugin can be found.
5. Submit the form using the **Send Post** button.

How it works...

Plugin submission is a fairly simple process, where any requests will usually be approved within a few days, giving developers access to a Subversion repository that they can use to upload their work and share it with the community.

Before submitting your request, you should search through existing plugins to be sure that you have not selected a name that already exists on the repository, as that will likely result in your request being turned down. You can do this by using the website's search engine, as well as trying to access an address that was named based on your plugin name. For example, following our Book Reviews plugin example, you could check to see if the address `http://wordpress.org/extend/plugins/book-reviews` exists. Finally, you should be sure to give a good description of your plugin's functionality.

It should be noted that your plugin does not need to be 100 percent complete or functional when you apply to be listed on the repository. Applying for this access early during your development process helps you secure the name to your idea before someone else takes it, and also gives you access to a hosted Subversion version control repository to easily keep backups of your work during development. If you work on a plugin with one or more people, this last benefit will make it very easy to exchange code between all contributors. Using the release tag mechanism described in the previous recipe, you can select the exact moment when your work is ready to become public and ready for download.

See also

- ▶ [Creating a readme file for your plugin recipe](#)

Uploading your plugin using Subversion

If you thought that using Subversion in the recipes of *Chapter 1, Preparing a Local Development Environment*, was overkill when you're working on a plugin locally, you will see that this knowledge comes in very handy once your hosting request has been approved by the WordPress team, as the system's backend relies on that version control system.

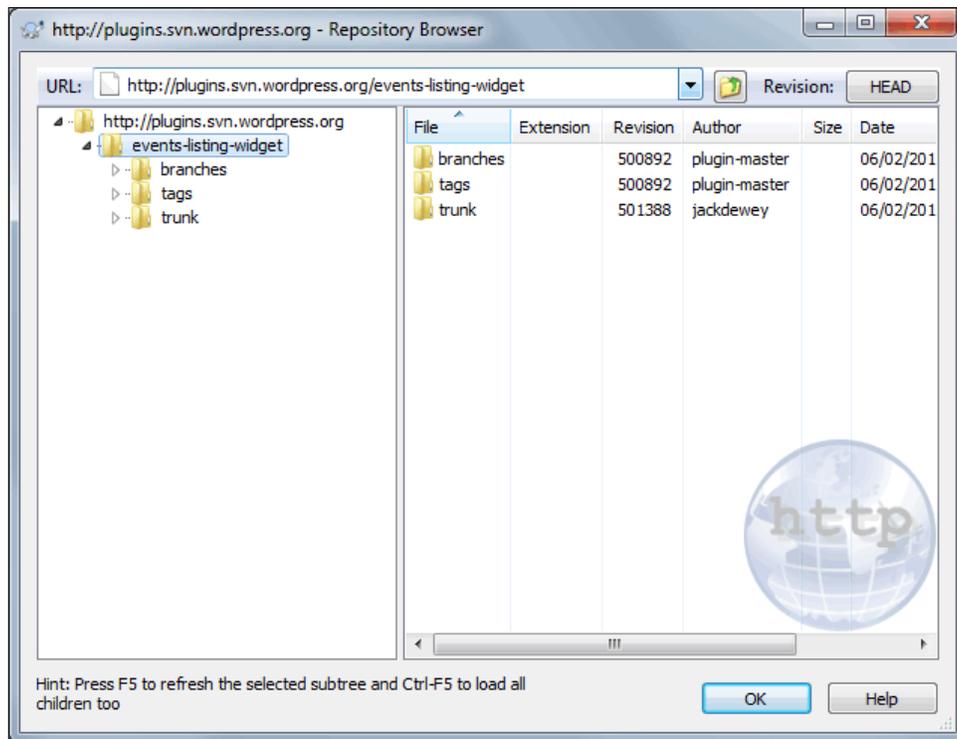
This recipe shows how to submit your creation to the `wordpress.org` site once a repository has been created for you.

Getting ready

You should have already followed the *Applying for your plugin to be hosted on wordpress.org* recipe to have an approved repository on the official site. You should have also installed a Subversion client as shown in the *Creating a local Subversion repository* recipe in *Chapter 1, Preparing a Local Development Environment*. Finally, you should have plugin files ready for upload.

How to do it...

1. Right-click in a file explorer and select the **TortoiseSVN | Repo-browser** menu.
2. Enter the address of your new repository, as indicated in your hosting approval e-mail. For example, for a plugin named **Book Reviews**, the address would be `http://plugins.svn.wordpress.org/book-reviews`.



- Right-click on the plugin's name in the left-hand side tree view, and select the **Checkout** option.
- Select a local folder on your computer as the **Checkout directory**.
- Click on **OK** to create a local copy of the server structure with the accompanying version control data.
- Copy your plugin's files to the **trunk** folder of the resulting directory structure.
- Select all files, right-click on them, and select the **TortoiseSVN | Add...** menu.
- Right-click on the **trunk** folder and select the **SVN Commit...** menu option.
- Enter a **Message** indicating that you are uploading the first version of this plugin.
- Click on **OK** to upload your files to the official repository.
- When prompted for authentication, use your `wordpress.org` **Username** and **Password**. Click on the **Save authentication** checkbox to avoid providing these credentials each time.

12. Approximately 10 to 15 minutes later, you will receive an e-mail confirming that new files have been uploaded to the repository. You will then be able to visit your plugin's page and download it. For our example **Book Reviews** plugin, the address of the page would be <http://wordpress.org/extend/plugins/book-reviews/>.

The screenshot shows the WordPress plugin page for 'Book Reviews'. At the top, it says 'Create your own book review web site!' with a 'Download Version 1.0' button. Below this are tabs for 'Description', 'Installation', 'FAQ', 'Screenshots', 'Changelog', 'Stats', and 'Admin'. The author is listed as 'jackdewey'. The description states: 'This plugin lets you add a book review system to your WordPress site. Using custom post types, administrators will be able to create and edit book reviews to be published on your site.' The tags are 'book, reviews'. On the right side, it lists 'Requires: 3.0 or higher', 'Compatible up to: 3.2.1', 'Last Updated: 2012-1-7', and 'Downloads: 0'. There are also sections for 'Average Rating' and 'My Rating', both showing five stars.

How it works...

The official WordPress plugin repository uses Subversion to manage all code files, provide version control services to developers, and find information to populate the extension's page. When your new repository gets created, it contains three main directories: `trunk`, `tags`, and `branches`.

The `trunk` directory is usually the main location where you place the latest version of your plugin files. Following the steps in the recipe, we copy our files to this location and commit them to the server. Once uploaded, the `wordpress.org` servers take care of creating a zipped copy of your work.

The `tags` directory is designed to hold pointers to various versions of your creation over time, as discussed in the *Creating a readme file for your plugin* recipe. This functionality used in conjunction with the **Stable tag** field of your plugin's `readme.txt` file, allows you to redirect users to a known working version of your work while you commit and test potentially unstable work to the `trunk`. New tags are created using the **Branch/Tag** item of the **TortoiseSVN** menu and associating a name to a specific revision. The `branches` directory has a similar function to `tags`, but is more focused towards the creation of alternate versions of plugins, or in-development revisions that include specific functionality.

There's more...

If you want to execute your plugin's code in a local WordPress development installation as you are writing it, the following section shows you how to manage your code.

Checking out plugins to your development installation

When checking out the complete plugin directory, you end up with a structure that cannot be executed directly in a local development installation of WordPress for testing and development purposes. Instead of checking out the entire directory structure, you can limit your selection to the `trunk` directory. This will only copy the contents of that specific folder to your system and you can set the target folder to be located directly under the `plugins` directory.

See also

- ▶ *Creating a readme file for your plugin* recipe
- ▶ *Checking out files from a Subversion repository* recipe in *Chapter 1, Preparing a Local Development Environment*
- ▶ *Committing changes to a Subversion repository* recipe in *Chapter 1, Preparing a Local Development Environment*

Providing a plugin banner image

While the plugin listing that we put in place by creating a `readme.txt` file and uploading it to the official plugin repository is perfectly functional, it does not really stand out amongst the sea of extensions that are available on the site. Thankfully, `wordpress.org` recently introduced a mechanism allowing plugin developers to add a banner image to their listing. This image can be anything from a simple picture to a complex graphic to advertise your creation.

This recipe explains how to prepare an image for your plugin and how to upload it to your repository.

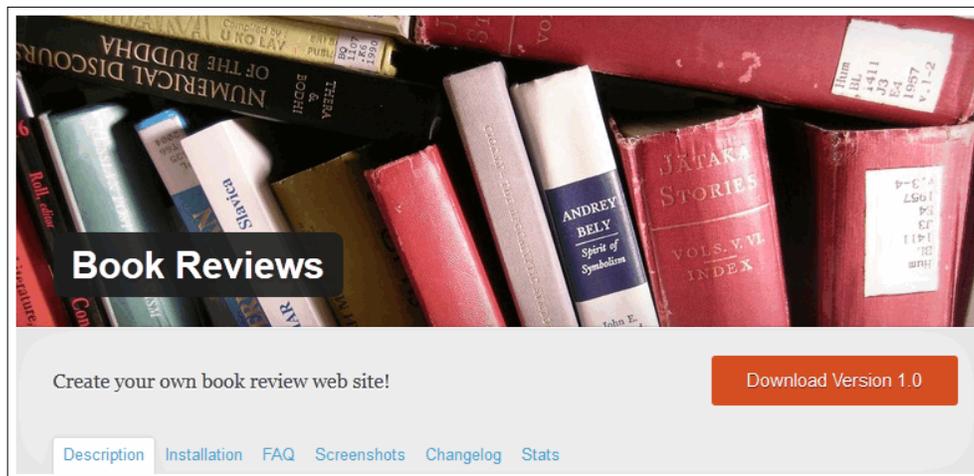
Getting ready

You should have already followed the *Applying for your plugin to be hosted on wordpress.org* and *Uploading your plugin using Subversion* recipes to have an approved repository on the official repository and plugin files uploaded to the server.

How to do it...

1. Create a new image that is exactly 772 x 250 pixels.
2. Save the image as a PNG file with the name `banner-772x250.png`.

3. Right-click in a file explorer and select the **TortoiseSVN | Repo-Browser** menu.
4. Enter the address of your plugin repository. For example, for a plugin named **Book Reviews**, the address would be `http://plugins.svn.wordpress.org/book-reviews/`.
5. Create a new top-level directory named `assets`, at the same level as `trunk`, `tags`, and `branches`.
6. Select the **assets** directory, then drag-and-drop the new image file in the folder to upload it to the server.
7. Specify a **Log Message** in the dialog that appears to explain why the file is being uploaded.
8. Visit your plugin's page on `wordpress.org` to see the image in place.



How it works...

When files are uploaded to the plugin repositories, the `wordpress.org` site checks for the presence of a specific image file with a specific name for the plugin banner. If this file is present, it changes the layout of the plugin page to incorporate the image. It is important to respect the image format and the specified dimensions when creating a plugin banner to make sure that it is displayed properly on the site. You should also make sure that no important content, text, or something similar is located in the bottom-left part of the image as that is where the plugin's name will be displayed.

See also

- ▶ *Uploading your plugin using Subversion recipe*

Index

Symbols

`$query_params` variable 149

`$valid` variable 196

`?>` character 75

`_e` function 273, 277

`__` function 270

A

action hooks

user settings, accessing from 107, 108

Activate option 75

add_action call 137

add_action function 47

add_filter function 52, 56

add_help_tab function 100

add_menu_page function 87

add_meta_box function 113, 114, 137, 170

Add New button 132

add_option function 76

add_options_page function 84, 87

add_query_arg function 190

addreviewmessage variable 190

add_settings_field function 106

add_shortcode function 186

add_thickbox function 236

admin code

splitting, from main plugin file 115, 116

administration page menu item

creating, on settings menu 82-84

admin_menu hook 85, 156

admin page

code, preparing for translation 270-273

custom table data, displaying 209-212

admin page content

rendering, HTML used 89-91

rendering, settings API used 100-106

admin page form fields

tooltips adding, TipTip plugin used 243-245

admin sections pages

formatting, meta boxes used 109-115

admin_url() function 50

AJAX

used, to dynamically update partial page contents 246-251

API 74

Application Programming Interface. *See* API

apply_filters function 53

archive page

creating, for custom post types 143-146

archives widget 253

arrays

used, for storing user settings 78, 79

AUTO_INCREMENT command 202

B

banner image, plugin

providing 289, 290

Bonjour Monde item 278

Book Review Dashboard Widget plugin 265

Book_Reviews class 257, 260, 262

Book Reviews menu item 132

Book Reviews plugin 194

book_reviews post type 135, 139

Book Review system 130

branches directory 288

C

calendar day selector

displaying, Datepicker plugin used 240-243

Calendar Picker plugin 242

captcha

implementing, on user forms 194-197

category editor

hiding, from custom post type 153-156

ch2lfa_footer_analytics_code function 57

ch2lfa_link_filter_analytics function 57

ch2tf_title_filter function 51

ch2ts_twitter_feed_shortcode function 64

ch2ye_youtube_embed_shortcode function 127

ch3mim_admin_menu function 86

ch3sapi_config_page function 103

ch3sapi_display_check_box function 103

ch4_br_add_book_review_fields function 137

ch4_br_add_columns function 157

ch4_br_admin_init function 135

ch4_br_author_column_sortable function 159

ch4_br_book_review_list function 147, 149

ch4_br_book_type_filter_list function 162

ch4_br_column_ordering function 159

ch4_br_create_book_post_type function 131, 137, 151

ch4_br_display_review_details_meta_box function 136

ch4_br_format_book_review_title function 164

ch4_br_template_include function 139

ch5_cfu_form_add_enctype function 177

ch5_cfu_register_meta_box function 178

ch5_hcf_remove_custom_fields_metabox function 176

ch5_psl_register_meta_box function 168, 169, 171

ch5_psl_save_source_data function 169

ch6_brus_book_review_form function 184, 189

ch6_brus_match_new_book_reviews function 188

ch6_brus_process_user_book_reviews function 195

ch7bt_create_table function 201

ch8bt_declare_ajaxurl function 248

ch8bt_load_query function 249

ch8cp_date_meta_box function 242, 244

ch8cp_register_meta_box function 241

ch8pud_footer_code function 234

ch8pud_load_scripts function 234, 238

ch9-book-review-widget directory 260

ch9brdw_add_dashboard_widget function 265

ch9brdw_dashboard_widget function 265

ch9brw_create_widgets function 254

ch10hw_plugin_init function 278

check_admin_referer 94

Checkout option 287

client-side content submission form

client-side user form access, controlling 187

creating 183-186

Codex

URL 268

coding errors

troubleshooting 59-62

columns

additional columns, displaying in custom post list page 157-161

Commit button 35

Commit Message field 35

Compare with working copy menu item 27

computer

web server, installing 6

configuration options, widgets

displaying 256

displaying, steps for 257, 258

validating 259, 261

working 259

confirmation message

displaying, on saved options 95-97

content_url() function 50

Cornerstone

URL 17

Create a XAMPP Desktop icon option 7

custom categories

adding, for custom post type 150-153

custom dashboard widget

adding 264, 266

custom database table

data, displaying in shortcodes 222

custom field section
hiding, in post editor 175, 176

Custom Fields editor 137

custom help pages
adding 97-100

custom meta boxes
used, for adding extra fields to meta boxes 168-171

custom post data
displaying, in theme templates 172-174

custom post list page
additional columns, displaying 157-161
filters, adding for custom taxonomies 161-163

custom post type
about 129
archive page, creating for 143-146
category editor, hiding from 153-156
creating 130
creating, steps for 130-133
custom categories, adding 150-153
data, displaying in shortcodes 146-149
data, initializing 77
new section, adding 135-138
permalinks slug, changing 134
single custom post type items displaying, custom templates used 138-142
using 130

custom table data
displaying, in admin page 209-212
retrieving, by implementing search function 224-227

custom tables
records, deleting 218, 221
records, inserting 213-217
records, updating 213-217
removing, on plugin removal 205-207
structure, updating on plugin upgrade 207, 209

custom taxonomies
filters for, adding to custom post list page 161-163

custom templates
used, for displaying single custom post type items 138-142

D

dashboard widget 253

data
importing, from user file into custom tables 227, 230

database tables
creating 200-203

Datepicker plugin
used, for displaying calendar day selector 240-243

dbdelta function 209

deactivation function 77

dedicated code editor
installing 29-31

dedicated text editor
installing 29-31

default user settings
creating, on plugin initialization 74-76

dialog close button
removing 236

direct file upload permission
post editor, extending for 177-182

Directory option 9

DocumentRoot configuration option 9

do_meta_boxes function 114

do_settings_sections function 105

do_shortcode function 150

drop-down list settings field
rendering 106

E

EasyCaptcha PHP script 197

EasyPHP
URL 10

e-mail notifications
on new submissions, sending 191-193

external files
loading, WordPress path utility function used 48-50

external images
loading, WordPress path utility function used 48-50

F

favicon meta tag 48

Featured Image meta box 133

fields

extra fields, adding to meta boxes 168-171

file history

viewing 27

files

checking, from Subversion repository 19-21

initial files, importing to local Subversion repository 17-19

filter hooks

user settings, accessing from 107, 108

filters

for custom taxonomies, adding to custom post list page 161-163

foreach loop 171

form method 259, 263

G

get_blog_prefix method 202

get_field_id method 259

get_field_name method 259

get_option function 91, 95, 193

get_permalink function 56

get_post_meta function 138, 142, 146, 172

get_post_type function 53

get_query_var function 149

get_results method 212

get_template_directory_uri() function 50

get_the_ID() template function 142

get_theme_root() function 50

get_the_title function 56

Git

URL 17

global wp_query object 146

Google Images

URL 132

Go To dialog box 31

GPL v2 license

URL 281

H

header

creating 40-42

Hide Custom Fields plugin 176

home_url() function 50

hook function 44

hooks

searching, in WordPress source code 48

HTML

used, for rendering admin page contents 89-91

I

IconArchive

URL 132

IDE 31

includes_url() function 50

Install Apache as service option 7

Install button 8

Installed Plugins list 42

Install MySQL as service option 7

integrated development environment. *See* IDE

intval function 261

is_front_page function 53, 237

is_user_logged_in function 187

items

hiding, from default menu 87-89

J

JavaScript files

localizing 279

jQuery

loading, into WordPress web pages 232

L

labels 153

language configuration, WordPress

changing 268-270

language file

loading, in plugin initialization 277, 279

link statistics tracking code

inserting in page body, plugin filters used 56-58

links widget 253
load_plugin_textdomain function 278
local Subversion repository
about 15-17
initial files, importing 17-19
local WordPress installation 11
locate_template function 142

M

MAMP
URL 10
Mercurial
URL 17
meta boxes
about 167, 168
adding, to post types 171
extra fields, adding to post editor 168-171
used, for formatting admin sections pages 109-115
meta box mechanism 135
multi-level administration menu
creating 85-87
multiple sets
of user settings, managing from single admin page 122-128
MySQL database server
managing, from NetBeans interface 36, 37

N

NetBeans
WordPress plugin creation module, installing 43, 44
NetBeans IDE
installing 31, 33
NetBeans interface
MySQL database server, managing 36, 37
Subversion repository, interacting with 34, 35
new submissions
e-mail notifications, sending on 191-193
noconflict mode 233
Notepad++
URL 29
not_found label 133

O

object-oriented PHP
used, to write plugins 70-72
option_id parameter 127
output content
adding to page headers, plugin actions used 44-47

P

Packt Publishing
URL 132
PagaVCS tool
URL 17
page body
link statistics tracking code inserting, plugin filters used 56-58
page headers
output content adding, plugin actions used 44-47
page title
modifying, plugin filters used 50, 52
updating to include custom post data, plugin filters used 164-166
parameters
used, for creating new shortcode 65, 66
partial page contents
updating partially, AJAX used 246-251
Permalinks 85
permalinks slug, custom post type
changing 134
phpMyAdmin
using, to simplify code creation 203, 204
plugin
applying, to host on wordpress.org 285, 286
banner image, providing 289, 290
data, removing on deletion 80-82
directory, checking 289
new options, adding 77
readme file, creating 282-284
uploading, subversion used 286-288
writing, object-oriented PHP used 70-72
plugin actions
used, for adding output content to page headers 44-47

plugin configuration data

processing 92-94
storing 92-94

Plugin Description fields 285**plugin file**

admin code, splitting from 115, 116
creating 40-42

plugin filters

used, for adding text after each items content
54, 55
used for inserting link statistics tracking code,
in page body 56-58
used, for modifying page title 50, 52
used, for updating page title 164-166

plugin initialization

default user settings, creating 74-76
language file, loading 277, 279

plugin output

formatting, by loading stylesheet 69, 70

plugin removal

custom tables, removing on 205-207

Plugins management page 132, 270**plugins_url function 50****plugins_url utility function 50****plugin upgrade**

custom tables structure, updating on 207,
209

Poedit

used, for translating text strings 275-277

pop-up dialog

display controlling, shortcodes used 237-240
displaying, built-in ThickBox plugin used
234-236
displaying, on selected pages 237

post editor

custom field section, hiding 175, 176
extending, for direct file upload permission
177-182
extra fields adding, custom meta boxes used
168-171

Post/Page Source meta box 170**posts_per_page query argument 149****post type. *See also* custom post type****post type**

custom post types, user-submitted content
saving in 187-190
meta box, adding 171

previous_post_links function 146**printing variable contents**

troubleshooting 59-62

print_r function 61**process_ch10hw_options function 272****Programmer's Notepad**

URL 29

R**readme file**

creating, for plugin 282, 284

recordOutboundLink Javascript function 58**records**

from custom table, deleting 218, 221
in custom table, inserting 213-217
in custom table, updating 213-217

register_activation_hook function 76**register_post_type function 133, 153****register_taxonomy function 152****register_widget function 255****remote web development environment**

creating 11

remove_meta_box function 176**render_widget, nb_book_reviews function
259****Revert to this revision menu item 27****S****Save all changes button 133****Save Changes button 134****saved options**

confirmation message, displaying on 95-97

Save Menu button 61**save_post action 171****Screem**

URL 29

search function

implementing, to retrieve custom table data
224-227

section

new section, adding to custom post type
editor 135-138

SELECT * command 212**Send Post button 285**

Set featured image link 132

settings API

used, for rendering admin page content 100-106

settings_fields function 105

settings menu

administration page menu item, creating 82-84

hook priority, settings 85

shortcode

creating, steps 63, 64

custom database table data, displaying 222

custom post type data, displaying 146-149

new enclosing shortcode, creating 67, 68

new shortcode, creating 63

new shortcode, creating with parameters 65, 66

output, modifying for translation 273, 274

used, for displaying pop-up dialog display 237-240

Show log menu item 35

show_tagcloud 153

show_ui option 153

single admin page

multiple sets of user settings, managing 122-128

single custom post type items

displaying, custom templates used 138-142

site_url() function 50

sprintf function 274

Stable tag field 288

strip_tags function 261

str_replace function 58

stylesheet

data, storing, in user settings 117-122

loading, to format plugin output 69, 70

Sublime Text 2

URL 29

Submit button 91

Subversion. *See* **SVN**

subversion file, statuses

added 21

conflicted 22

deleted 21

ignored 22

modified 21

non-versioned 21

normal 21

Subversion repository

changes, committing to 22-24

files, checking out from 19-21

files, updating to latest version 24

interacting, from NetBeans interface 34, 35

local Subversion repository, creating 15, 16

local Subversion repository, initial files importing 17

subversion file, statuses 21

SVN

about 15

used, for uploading plugin 286-288

T

taxonomy 150

template_include filter hook 142

template_redirect action hook 190

text

adding after each items content, plugin filters used 54, 55

text area settings field

rendering 106

TextMate

MateURL 29

text strings

translating, Poedit used 275-277

TextWrangler

URL 29

the_content() function 142

theme templates

custom post data, displaying 172-174

the_title() function 142

ThickBox plugin

built-in ThickBox plugin, used for displaying pop-up dialog box 234-236

TipTip plugin

used, for adding tooltips to admin page form fields 243-245

tool tips

adding to admin page form fields, TipTip plugin used 243-245

TortoiseSVN

configuring, to use external diff viewer 26

TortoiseSVN menu 288

TortoiseSVN site

URL 15

translate function 270

translation

admin page code, preparing for 270-273

advanced functions 279

file, updating 279

shortcode output, modifying for 273, 274

troubleshooting

coding errors 59-62

printing variable contents 59-62

U

UAC 7

uncommitted file changes

reverting to 25, 26

update method 259

update_post_meta 171

User Access Control. *See* UAC

user capability 84

user files

into custom tables, data importing from 227, 230

user forms

captcha, implementing on 194, 195, 196, 197

user settings

accessing, from action hooks 107, 108

accessing, from filter hooks 107, 108

default user settings, applying for translation 269, 270

multiple sets, managing from user admin page 122-128

storing, arrays used 78, 79

stylesheet data, storing in 117-122

user-submitted content

in custom post types, saving 187-191

moderating 191

V

Versions

URL 17

View Book Review button 133

W

WampServer

URL 10

web server

benefits 6

installing, on computer 6-10

web server packages

URL 11

widget display function

implementing 261

implementing, steps for 263

widget method 256, 263

widgets

about 253

archives widget 253

configuration options, displaying 256-258

configuration options, validating 259, 261

creating, steps for 254, 255

custom dashboard widget, adding 264, 266

dashboard widget 253

display function, implementing 261

links widget 253

working 255, 256

widgets_init action hook 255

widget_title function 259

WordPress

about 267

language configuration, changing 268

widget, creating 254-256

WordPress Codex

URL 47, 133

WordPress header

creating 40-42

wordpress.org

hosting on, by applying for plugin 285, 286

WordPress path utility function

using, to load external files 48-50

using, to load external images 48-50

WordPress plugin

creating 40-42

WordPress plugin creation module

installing, in NetBeans 43, 44

WordPress source code

hooks, searching for 48

WordPress Version 2.2

widgets 253

WordPress web pages

jQuery, loading 232, 233

wp_add_dashboard_widget function 266

wp_count_posts utility function 266

wpdb class 202

WP_DEBUG_DISPLAY 63

WP_DEBUG_LOG 63

wp_die function 190

wp_dropdown_categories function 163

wp_enqueue_script function 233

wp_insert_post function 190

wp_localize_script function 279

wp_mail function 191, 193

wp_nonce_field function 91, 186, 189

WP_Query class 149

WP_Query object 263

wp_redirect function 190

wp_redirect function 95

wp_title function 166

wp_upload_dir() function 50

wp_verify_nonce function 190

WP_Widget class 255

X

XAMPP website

URL 7



Thank you for buying WordPress Plugin Development Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



WordPress 3 Site Blueprints

ISBN: 978-1-847199-36-2 Paperback: 300 pages

Ready-made plans for 9 different professional WordPress sites

1. Everything you need to build a varied collection of feature-rich customized WordPress websites for yourself
2. Transform a static website into a dynamic WordPress blog
3. In-depth coverage of several WordPress themes and plugins
4. Packed with screenshots and step-by-step instructions to help you complete each site



WordPress 3 Complete

ISBN: 978-1-84951-410-1 Paperback: 344 pages

Create your own complete website or blog from scratch with WordPress

1. Learn everything you need for creating your own feature-rich website or blog from scratch
2. Clear and practical explanations of all aspects of WordPress
3. In-depth coverage of installation, themes, plugins, and syndication
4. Explore WordPress as a fully functional content management system

Please check www.PacktPub.com for information on our titles

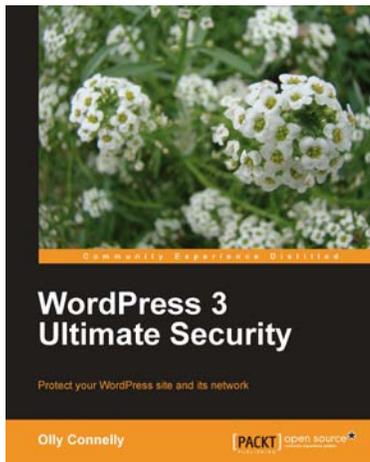


WordPress for Education

ISBN: 978-1-84951-820-8 Paperback: 144 pages

Create interactive and engaging e-learning websites with WordPress

1. Develop effective e-learning websites that will engage your students
2. Extend the potential of a classroom website with WordPress plugins
3. Create an interactive social network and course management system to enhance student and instructor communication



WordPress 3 Ultimate Security

ISBN: 978-1-84951-210-7 Paperback: 408 pages

Protect your WordPress site and its network

1. Know the risks, think like a hacker, use their toolkit, find problems first – and kick attacks into touch
2. Lock down your entire network from the local PC and web connection to the server and WordPress itself
3. Find out how to back up and secure your content and, when it's scraped, know what to do to enforce your copyright
4. Understand disaster recovery and use the best-of-breed tools, code, modules, techniques, and plugins to insure against attacks

Please check www.PacktPub.com for information on our titles