

Community Experience Distilled

Learning FuelPHP for Effective PHP Development

Use the flexible FuelPHP framework to quickly and effectively create PHP applications

Ross Tweedie

[PACKT] open source
PUBLISHING



Community Experience Distilled

Learning FuelPHP for Effective PHP Development

Use the flexible FuelPHP framework to quickly and effectively create PHP applications

Ross Tweedie

[PACKT] open source
PUBLISHING

Table of Contents

Learning FuelPHP for Effective PHP Development
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. What is FuelPHP?
    A brief history of FuelPHP
    Features of FuelPHP
        (H)MVC
        Modular and extendable
        Security
        Oil – the power of the command line
        ORM
        Base controller classes and model classes
        The authentication package
        Template parsers
        Documentation
    What to look forward to in Version 2.0
        Global scope and moving to dependency injection

        Singleton
        Multiton
        Dependency injection container
        Coding standards
        Package management and modularization
        Backwards compatibility
        Speed boosts
    Summary
2. Installation
    Preparing the development environment
        Apache
        PHP
        Database interactions
        Source control – introducing Git
            Further information about Git
    Getting and installing FuelPHP with curl and Oil
        Clone from GitHub
            Continuing the installation
        Setting up your project
        Using submodules to easily update the FuelPHP core and packages
        Committing your code
        Composer – the package manager
        Configuration
        Running in the production environment
        Performing migrations
    Summary
3. The Architecture
    Environments, constants, and configuration
        Server and Apache configuration
        The bootstrap PHP file
        Configuration
        Constants
    Models, views, and controllers
        Views and ViewModels
        Models
        Controllers

        Template
        Rest
        Hybrid
    Summary
4. Demo Application
    Getting started
    Creating a database table
        Entries
        Categories
        Users
        categories_entries
    Migrations and Oil
    Models
    Controllers
    Views
    Generating an administration system using Oil
    Summary
5. Packages
    What is a package?
    Recommended packages
        OAuth
        OAuth2
        Mandrill
        Sprockets
    Using the Auth package for user authentication
        SimpleAuth
        ORMAuth
        OPAuth
    Composer
    An introduction to building your own package
        Setting up a repository
        Working the package as a submodule
        Structuring your package
        Configuring your package
        Using your package
    Letting people know about your package
    Summary

6. Advanced Topics
    What modules are and how to use them
        Namespaces
        The module folder structure
        Using the module from the main application
    Tasks
    Routing
    Unit testing
        So what is unit testing?
        PHPUnit
        Running unit tests
        Creating unit tests
        Grouping unit tests
        Configuration and module testing
    Profiling
    Summary
7. Welcome to the Community
    FuelPHP and GitHub
    Getting help and support
    FuelPHP on social media
        Twitter
        Facebook
    Parting words
Index

Learning FuelPHP for Effective PHP Development

Learning FuelPHP for Effective PHP Development

Copyright © 2013 Packt Publishing

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

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

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

First published: November 2013

Production Reference: 1141113

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK..

ISBN 978-1-78216-036-6

www.packtpub.com

Cover Image by Suresh Mogre (<suresh.mogre.99@gmail.com>)

Credits

Author

Ross Tweedie

Reviewers

Adam Booth

Rémi Delhaye

Derrick Egersdorfer

Acquisition Editor

Kartikey Pandey

Kevin Colaco

Commissioning Editor

Shaon Basu

Technical Editor

Akashdeep Kundu

Copy Editors

Alisha Aranha

Roshni Banerjee

Tanvi Gaitonde

Sayanee Mukherjee

Project Coordinator

Amey Sawant

Proofreader

Linda Morris

Indexer

Hemangini Bari

Tejal Soni

Production Coordinator

Kirtee Shingan

Cover Work

Kirtee Shingan

About the Author

Ross Tweedie is a developer who specializes in creating applications using PHP and FuelPHP. He is passionate about using FuelPHP to test new ideas and concepts effectively. He is using open source technologies to enable people to collaborate with one another. His background is both in design and development, and he has experience in a wide range of industries.

I would like to thank my family for putting up with the writing sessions. I would also like to thank Stuart Cullum and Davide Casali for their encouragement for this book. I would also like to thank Maarten Jacobs for some of the great tools created and used during this book.

I would also like to thank the team at Packt Publishing including Amey Sawant, Shaon Basu, and Yogesh Dalvi for their help with this book.

About the Reviewers

Adam Booth is a devoted web developer living in Oklahoma City. He loves all kinds of technology and prefers to stay on the cutting edge of tech news. In his spare time he enjoys watching movies and working on side projects to learn new frameworks and languages. One of these side projects was to create a home automation system that included a beautiful dashboard used to deliver useful information such as weather forecasts for the week, movies playing in theaters, shopping lists, and calendar events happening in the future.

Adam has been using PHP for many years and FuelPHP has been his framework of choice for all new PHP projects he starts. He is excited about the future of web technologies and believes this is where the future of development will really thrive.

I would like to thank my wife, Mallory, for all of the support she has given me over the years. She is creative, resourceful, and one of the best debuggers I know. I'd also like to thank my parents, Don and Susan, who have always been inspirational and encouraging. As small business owners, they have shown me how hard work and perseverance pays off.

I hope you enjoy reading this book as much as I did. FuelPHP is a great framework with a community full of developers willing to help. The team members for FuelPHP are great at what they do and somehow still find the time to answer questions in the forums and other websites.

Rémi Delhaye is a 19-year-old French IT student. His hobbies are Football (U.S.), Muay Thaï, Music, and Computer Science. He is passionate about mobile and web development, and his career objective is to create his own international IT business.

He is also passionate about web development and loves working on fantastic projects!

As an IT Student, he worked in one web agency (Calais, France) as an Intern Web Developer and in another Incubation firm (start-up) as a Junior Web Developer (Symfony2) and Frontend Developer (Miami, FL).

He did his schooling from EPSI (Master Degree of IT Engineering). You can contact him at:

- **GitHub**: http://github.io/RemiDelhaye
- **Website**: http://remidelhaye.fr

Derrick Egersdorfer is a 34-year-old web developer, living in Durban, South Africa. He has over 9 years of experience of using PHP, MySQL, and various JavaScript frameworks. In 2010, Derrick partnered with two of his colleagues to form a web and graphic design studio. Since then, Derrick has been specializing in highly customized website solutions, such as Social Media Networks and Geographic Information Systems.

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.

Preface

FuelPHP is a great tool, focused on writing web applications in a simple and fast way that requires less PHP code. It allows you to quickly build prototypes using scaffolding and command-line tools, allowing you to concentrate on the fun part of trialing ideas and concepts.

Learning FuelPHP for Effective PHP Development contains several tutorials that will help you build a powerful and engaging application, and in the process, you will learn more about FuelPHP. This book gives the experience of installing and building a FuelPHP project in a step-by-step way.

The book starts by detailing the features of FuelPHP and then delves into the creation of a simple application. We then move on to scaffolding our application using the powerful FuelPHP Oil command-line tool. We conclude the book with an introduction to the community.

What this book covers

Chapter 1, *What is FuelPHP?*, gives you a quick introduction to FuelPHP and some of the changes expected in Fuel PHP Version 2.0.

Chapter 2, *Installation*, covers installing FuelPHP and setting up your development environment.

Chapter 3, *The Architecture*, gives you a basic overview of the FuelPHP architecture and will cover where to store your project code followed by the summary of this chapter.

Chapter 4, *Demo Application*, will cover the creation of a project with example code and a step-by-step guide. It will also cover creating the administration system, easy form creation, and using the HTML5 Boilerplate with templates.

Chapter 5, *Packages*, will introduce packages and highlight important ones for use on projects.

Chapter 6, *Advanced Topics*, will introduce more advanced topics including modules, routing, and unit testing.

Chapter 7, *Welcome to the Community*, covers getting help from the community.

What you need for this book

This book assumes you have a computer with Apache and PHP installed.

Who this book is for

This book is for PHP developers looking forward to learning more about using the FuelPHP framework for effective PHP development. It is assumed that the reader has a basic understanding of general PHP development.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " There is no need to detail the `id`, `created_at`, or `updated_at` fields, as these will be automatically generated."

A block of code is set as follows:

```
<?php
return array(
    'default' => array(
        'connection' => array(
            'dsn' =>
'mysql:host=localhost;dbname=journal_dev',
            'username' => 'journal_dev',
            'password' => 'journal_dev',
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
$ git checkout -b develop
$ git push origin develop
```

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: "On the Initializr website, choose the **Classic H5BP** option and then the **Responsive Bootstrap** 2.3.2 template option. Finally, click on **Download it!**".

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of

our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

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

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

Questions

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

Chapter 1. What is FuelPHP?

FuelPHP is a fresh approach to frameworks that strive to be more community-driven. There are already more than a hundred developers contributing and extending the source code. There are many more helping to provide some of the best documentation for a framework currently available.

Since it is community-driven, everyone is in an equal position to spot bugs, provide fixes, or add new features to the framework. This has led to the creation of features such as the new temporal **ORM** (**Object Relation Mapper**), which is a first for any PHP-based ORM. This also means that everyone can help build tools that make development easier, more straightforward, and quicker.

The framework is lightweight and allows developers to load only what they need. It's a configuration over convention approach. Instead of enforcing conventions, they act as recommendations and best practices. This allows new developers to jump onto a project and catch up to speed quicker. It also helps when we want to find extra team members for projects.

The topics covered in this chapter are:

- A brief history of FuelPHP
- Features of FuelPHP
- What to look forward to in Version 2

A brief history of FuelPHP

FuelPHP started out with the goal of adopting the best practices from other frameworks to form a thoroughly modern starting point, which makes full use of PHP Version 5.3 features, such as namespaces. It has little in the way of legacy and compatibility issues that can affect older frameworks.

The framework was started in the year 2010 by *Dan Horrigan*. He was joined by Phil Sturgeon, Jelmer Schreuder, Harro Verton, and Frank de Jonge. FuelPHP was a break from other frameworks such as CodeIgniter, which was basically still a PHP 4 framework. This break allowed for the creation of a more modern framework for PHP 5.3, and brings together decades of experience of other languages and frameworks, such as Ruby on Rails and Kohana. After a period of community development and testing, Version 1.0 of the FuelPHP framework was released in July 2011. This marked a version ready for use on production sites and the start of the growth of the community.

The community provides periodic releases (at the time of writing, it is up to Version 1.7) with a clear roadmap (http://fuelphp.com/roadmap) of features to be added. This also includes a good guide of progress made to date.

The development of FuelPHP is an open process and all the code is hosted on GitHub at https://github.com/fuel/fuel, and the main core packages can be found in other repositories on the Fuel GitHub account—a full list of these can be found at https://github.com/fuel/.

Features of FuelPHP

Using a Bespoke PHP or a custom-developed framework could give you a greater performance. FuelPHP provides many features, documentation, and a great community. The following sections describe some of the most useful features.

(H)MVC

Although FuelPHP is a **Model-View-Controller** (**MVC**) framework, it was built to support the HMVC variant of MVC. Chapter 3, *The Architecture*, covers the MVC design pattern in detail. **Hierarchical Model-View-Controller** (**HMVC**) is a way of separating logic and then reusing the controller logic in multiple places. This means that when a web page is generated using a theme or a template section, it can be split into multiple sections or widgets. Using this approach, it is possible to reuse components or functionality throughout a project or in multiple projects.

In addition to the usual MVC structure, FuelPHP allows the use of presentation modules (ViewModels). These are a powerful layer that sits between the controller and the views, allowing for a smaller controller while still separating the view logic from both the controller and the views. If this isn't enough, FuelPHP also supports a router-based approach where you can directly route to a closure. This then deals with the execution of the input URI.

Modular and extendable

The core of FuelPHP has been designed so that it can be extended without the need for changing any code in the core. It introduces the notion of packages, which are self-contained functionality that can be shared between projects and people. Like the core, in the new versions of FuelPHP, these can be installed via the **Composer** tool. Just like packages, functionality can also be divided into modules.

For example, a full user-authentication module can be created to handle user actions, such as registration. Modules can include both logic and views, and they can be shared between projects. The main difference between packages and modules is that packages can be extensions of the core functionality and they are not routable, while modules are routable. Both packages and modules are covered in Chapter 5, *Packages* and Chapter 6, *Advanced Topics*.

Security

Everyone wants their applications to be as secure as possible; to this end, FuelPHP handles some of the basics for you. Views in FuelPHP will encode all the output to ensure that it's secure and is capable of avoiding **Cross-site scripting** (**XSS**) attacks. This behavior can be overridden or can be cleaned by the included htmLawed library.

The framework also supports **Cross-site request forgery** (**CSRF**) prevention with tokens, input filtering, and the query builder, which tries to help in preventing SQL injection attacks. **PHPSecLib** is used to offer some of the security features in the framework.

Oil – the power of the command line

If you are familiar with CakePHP or the Zend framework or Ruby on Rails, then you will be comfortable with FuelPHP Oil. It is the command-line utility at the heart of FuelPHP—designed to speed up development and efficiency. It also helps with testing and debugging. Although not essential, it proves indispensable during development.

Oil provides a quick way for code generation, scaffolding, running database migrations, debugging, and cron-like tasks for background operations. It can also be used for custom tasks and background processes.

Oil is a package and can be found at https://github.com/fuel/oil.

ORM

FuelPHP also comes with an **Object Relation Mapper** (**ORM**) package that helps in working with various databases through an object-oriented approach. It is relatively lightweight and is not supposed to replace the more complex ORMs such as Doctrine or Propel.

The ORM also supports data relations such as:

- belongs-to
- has-one
- has-many
- many-to-many relationships

Another nice feature is cascading deletions; in this case, the ORM will delete all the data associated with a single entry.

The ORM package is available separately from FuelPHP and is hosted on GitHub at https://github.com/fuel/orm.

Base controller classes and model classes

FuelPHP includes several classes to give a head start on projects. These include controllers that help with templates, one for constructing RESTful APIs, and another that combines both templates and RESTful APIs.

On the model side, base classes include **CRUD** (**Create**, **Read**, **Update**, and **Delete**) operations. There is a model for soft deletion of records, one for nested sets, and lastly a temporal model. This is an easy way of keeping revisions of data.

The authentication package

The authentication framework gives a good basis for user authentication and login functionality. It can be extended using drivers for new authentication methods. Some of the basics such as

groups, basic ACL functions, and password hashing can be handled directly in the authentication framework.

Although the authentication package is included when installing FuelPHP, it can be upgraded separately to the rest of the application. The code can be obtained from https://github.com/fuel/auth.

Template parsers

The parser package makes it even easier to separate logic from views instead of embedding basic PHP into the views. FuelPHP supports many template languages, such as Twig, Markdown, Smarty, and **HTML Abstraction Markup Language** (**Haml**).

Documentation

Although not particularly a feature of the actual framework, the documentation for FuelPHP is one of the best available. It is kept up-to-date for each release and can be found at http://fuelphp.com/docs/.

What to look forward to in Version 2.0

Although this book focuses on FuelPHP 1.6 and newer, it is worth looking forward to the next major release of the framework. It brings significant improvements but also makes some changes to the way the framework functions.

Global scope and moving to dependency injection

One of the nice features of FuelPHP is the global scope that allows easy static syntax and instances when needed. One of the biggest changes in Version 2 is the move away from static syntax and instances. The framework used the **Multiton design pattern**, rather than the **Singleton design pattern**. Now, the majority of Multitons will be replaced with the **Dependency Injection Container** (**DiC**) design pattern, but this depends on the class in question.

The reason for the changes is to allow the unit testing of core files and to dynamically swap and/or extend our other classes depending upon the needs of the application. The move to dependency injection will allow all the core functionality to be tested in isolation.

Before detailing the next feature, let's run through the design patterns in more detail.

Singleton

Ensures that a class only has a single instance and it provides a global point of access to it. The thinking is that a single instance of a class or object can be more efficient, but it can add unnecessary

restrictions to classes that may be better served using a different design pattern.

Multiton

This is similar to the singleton pattern but expands upon it to include a way of managing a map of named instances as key-value pairs. So instead of having a single instance of a class or object, this design pattern ensures that there is a single instance for each key-value pair. Often the multiton is known as a registry of singletons.

Dependency injection container

This design pattern aims to remove hard coded dependencies and make is possible to change them either at run time or compile time.

One example is ensure that variables have default values but also allow for them to be overridden, also allow for other objects to be passed to class for manipulation.

It allows for mock objects to be used whilst testing functionality.

Coding standards

One of the far-reaching changes will be the difference in coding standards. FuelPHP Version 2.0 will now conform to both PSR-0 and PSR-1. This allows a more standard auto-loading mechanism and the ability to use Composer. Although Composer compatibility was introduced in Version 1.5, this move to PSR is for better consistency. It means that the method names will follow the "camelCase" method rather than the current "snake_case" method names. Although a simple change, this is likely to have a large effect on existing projects and APIs.

With a similar move of other PHP frameworks to a more standardized coding standard, there will be more opportunities to re-use functionality from other frameworks.

Package management and modularization

Package management for other languages such as Ruby and Ruby on Rails has made sharing pieces of code and functionality easy and common-place. The PHP world is much larger and this same sharing of functionality is not as common. **PHP Extension and Application Repository** (**PEAR**) was a precursor of most package managers. It is a framework and distribution system for re-usable PHP components. Although infinitely useful, it is not as widely supported by the more popular PHP frameworks.

Starting with FuelPHP 1.6 and leading into FuelPHP 2.0, dependency management will be possible through Composer (http://getcomposer.org). This deals with not only single packages, but also their dependencies. It allows projects to consistently set up with known versions of libraries required by each project. This helps not only with development, but also its testability of the project as well as its maintainability.

It also protests against API changes. The core of FuelPHP and other modules will be installed via Composer and there will be a gradual migration of some Version 1 packages.

Backwards compatibility

A legacy package will be released for FuelPHP that will provide aliases for the changed function names as part of the change in the coding standards. It will also allow the current use of static function calling to continue working, while allowing for a better ability to unit test the core functionality.

Speed boosts

Although initially slower during the initial alpha phases, Version 2.0 is shaping up to be faster than Version 1.0. Currently, the beta version (at the time of writing) is 7 percent faster while requiring 8 percent less memory. This might not sound much, but it can equate to a large saving if running a large website over multiple servers. These figures may get better in the final release of Version 2.0 after the remaining optimizations are complete.

Summary

We now know a little more about the history of FuelPHP and some of the useful features such as ORM, authentication, modules, (H)MVC, and Oil (the command-line interface).

We have also listed the following useful links, including the official API documentation (http://fuelphp.com/docs/) and the FuelPHP home page (http://fuelphp.com).

This chapter also touched upon some of the new features and changes due in Version 2.0 of FuelPHP.

In the next chapter, we will install FuelPHP along with an introduction to the different environments and configurations.

Chapter 2. Installation

In this chapter, we will run through some of the basics of installing FuelPHP. Even as an experienced PHP developer, some of the topics may be new. We will run through the basics of **Git** for source control and touch upon automated deployment using a Ruby tool called **Capistrano** in later chapters. Don't worry, although it's written in Ruby, it's really easy to use even if you have not used Ruby before.

Everyone has their own way of setting up their development environment—some prefer compiling Apache from source, while others prefer the simplicity of MAMP or WAMP. Whatever your environment of choice, FuelPHP is likely to be fairly quick and easy to set up.

In this chapter, we will cover the following topics:

- Preparing the development environment
- Source control with Git
- Installing FuelPHP and setting up your project
- Working with different environments and migrating database changes

Preparing the development environment

FuelPHP should work with any web server, and has been tested to work on Apache, IIS, and Nginx. It has also been built to work on Windows and *nix (Unix, Linux, and Mac) operating systems.

As far as this guide goes, the examples will be based on *nix and Mac, but the same steps will work on other operating systems such as Windows.

Apache

Other web servers are available, but for the purpose of this book we assume the use of Apache.

For the use of clean URLs such as `http://example.com/welcome/hello`, Apache will need the `mod_rewrite` module installed and enabled.

PHP

You're likely to already have PHP installed, especially with an interest in FuelPHP.

FuelPHP requires PHP Version 5.3 or greater. It also uses several PHP extensions:

- `fileinfo()`: This extension is used for uploading files and may require a manual installation on Windows
- `mbstring()`: This is used throughout the framework
- `mcrypt`: This is used for the core crypt functions
- `PHPSecLib`: This is used as an alternative should `mcrypt` not be found

There are many ways of setting up PHP, more information can be found at both `php.net` and http://www.phptherightway.com.

Database interactions

Interactions with databases in FuelPHP are handled by drivers; as such, it's possible to support a multitude of databases. Out of the box, FuelPHP supports MySQL (via MySQL, MySQLi drivers), MongoDB, Redis, and any databases that have **PHP Data Objects** (**PDO**) drivers.

Applications and sites written in FuelPHP can work perfectly without either a relational database or no SQL data store like Mongol DB, but in this book we will be using one to demonstrate some awesome features of FuelPHP.

MySQL is well supported on most platforms and is one of the most widely used database systems. You can visit

http://dev.mysql.com/downloads/mysql, which is a great place to start with instruction and installation programs.

Source control – introducing Git

Although not all projects will require the use of source control, it can certainly be handy to roll back to older versions of code, or work with a team of developers.

Git is an extremely powerful tool and is relatively easy to start using. Although not all projects require the use of source control systems such as Git or **Subversion**, they will all benefit from it. A key benefit is the ability to revert to a previous version of the code and think of it as a universal "undo" function. Instead of the "undo" function working on a single document, it works on the entire project. The core team, while developing and enhancing the FuelPHP framework, also uses them. If you're not comfortable with source control, the framework can be installed manually, but this book will assume you're using Git and source control.

If you would like to have more information about using Git, there is an online version of the manual available at the following link:

http://git-scm.com/book/

Git originates from the Linux world. So, compiling from source is the traditional way of installing it, but alternatives now exist. Ubuntu can install Git with the following `apt-get` command:

```
$ sudo apt-get install git-core
```

OS X users have a multitude of options for installing Git, including **MacPorts** (http://www.macports.org) and **Homebrew** (http://github.com/mxcl/homebrew):

- While installing via MacPorts, we use the following command:

```
$ sudo port install git-core +svn
```

- While using Homebrew, we use:

```
$ brew install git
```

Windows users have the `msysGit` installer available at the following link:

http://msysgit.github.io

Once Git is installed, it's recommended to configure it with your user details, as shown in the following commands:

```
$ git config --global user.name "Your name"
$ git config --global user.email "user@domain.com"
```

To make the output from Git more visual, it's recommended to enable the colors in the command line:

```
$ git config --global color.ui auto
```

Further information about Git

A great alternative for the Git client and available commands can be found at http://git-scm.com. GitHub also has a good guide for setting up Git, available at the following link:

https://help.github.com/articles/set-up-git

Tip

Extra reading material

If you are familiar with source control, the core concepts of Git will be very familiar. It may take some time to get used to the syntax. Here are some links that will prove to be useful for learning more about Git:

- http://git-scm.com/book/en/: An online guide to Git
- http://git-scm.com: A great collection of resources
- http://nvie.com/posts/a-successful-git-branching-model/: A great way of working with Git is by using a tool named **git-flow**, which helps to keep branching structured and in control

Getting and installing FuelPHP with curl and Oil

The easiest way of installing FuelPHP is through the use of `curl` (or `wget`) and a stripped down version of Oil, the FuelPHP command-line tool.

To install the quick installer, you can run the following command from the shell or a terminal window:

```
$ curl get.fuelphp.com/oil | sh*
```

This will ask for your password to install the new files to the `/usr/bin` directory.

After this is over, you only need to use `oil` rather than `php oil`, but both will work for the command-line iterations.

> ## Note
>
> If you have used a FuelPHP version older than 1.6 previously, you will need to reinstall FuelPHP to allow it to use the **Composer** tool.

To create a new project, simply run the following command:

```
$ oil create <project>
```

In this, `<project>` is the name of your project. This will create a folder of the project name in the current directory. All of your application code and packages will be created in the project folder.

Clone from GitHub

If you would prefer not to use curl, or would prefer to just clone the FuelPHP repository in the command line, navigate to the folder where you would like the files to sit. For example:

```
$ cd /Users/ross/Sites
$ git clone --recursive
git://github.com/fuel/fuel.git <project name>
```

This will create a folder called `<project name>` in your web server root. It will contain all the necessary FuelPHP files, including all the core packages.

Continuing the installation

In addition to the single command, or cloning from GitHub, it is also possible to download the files manually and install that way. Further information on this method can be found at http://fuelphp.com/docs/installation/instructions.html.

Note

If you're manually installing the files, for security reasons, it is recommended to move the fuel folder outside of the publicly accessible web folder directory. The default `.htaccess` file included with FuelPHP also stops the core files from being web-accessible.

While working on a project, the write permissions for certain application folders might get changed. These folders might include those for logs and caches causing the application to stop functioning. If this happens, Oil can be used to correct them. It can also be used to make them writable by the web server:

```
$ php oil refine install
    Made writable: APPPATH/cache
    Made writable: APPPATH/logs
    Made writable: APPPATH/tmp
    Made writable: APPPATH/config
```

Setting up your project

Now that you have installed FuelPHP, it's really easy to set up a new project. First, you will need to navigate to the folder where you would like to work from, for example, the `Sites` folder on Mac OS X. After this, run the following command:

```
php oil create <project name>
```

Then again, run:

```
$ cd ~/Sites/
$ php oil create book
```

This will install the core files and packages required to run FuelPHP. It will also set up Git submodules within the project. These can sometimes be tricky to work with, but FuelPHP uses them in an extremely versatile and powerful way. With submodules, you have fine-grained control over the versions of the packages you're using on the project. It also makes it really easy to upgrade or install security updates.

The structure created by FuelPHP is fairly straightforward:

```
/
   fuel/
      app/
      core/
      packages/
   public/
```

```
    .htaccess
    assets/
    index.php
oil
```

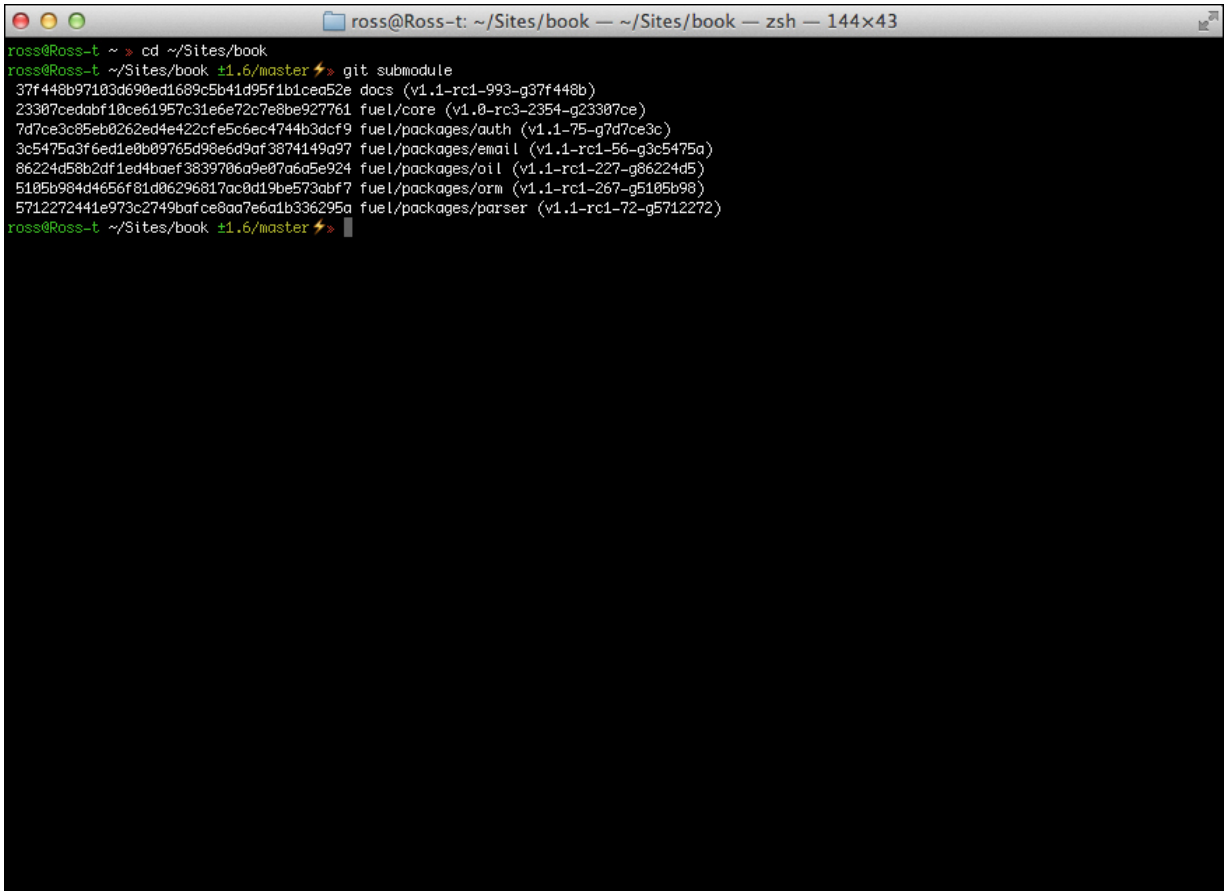
Files like CSS and JavaScript are placed in the `assets` folder in the public directory. Once you have installed some packages, the majority of the changes you make for your project will take place in the `fuel/app` folder. We will run through these with examples in the next few chapters.

Using submodules to easily update the FuelPHP core and packages

Submodules are a great way to work with multiple repositories in the project in a controlled manner. For example, it is possible to upgrade a version of the core FuelPHP framework while keeping other third-party packages on the old version. This makes it easier to test new functionality to ensure that it doesn't affect your project, or to highlight changes that you may need to make. In this section, we will run through some of the basics of using submodules, but if you would like to have more information I'd recommend looking at the *Submodules* section of the Git manual available at http://git-scm.com/book/en/Git-Tools-Submodules.

If you want to see which submodules are currently set for your project, navigate to the root of the project and then run the `git submodule` command, as shown:

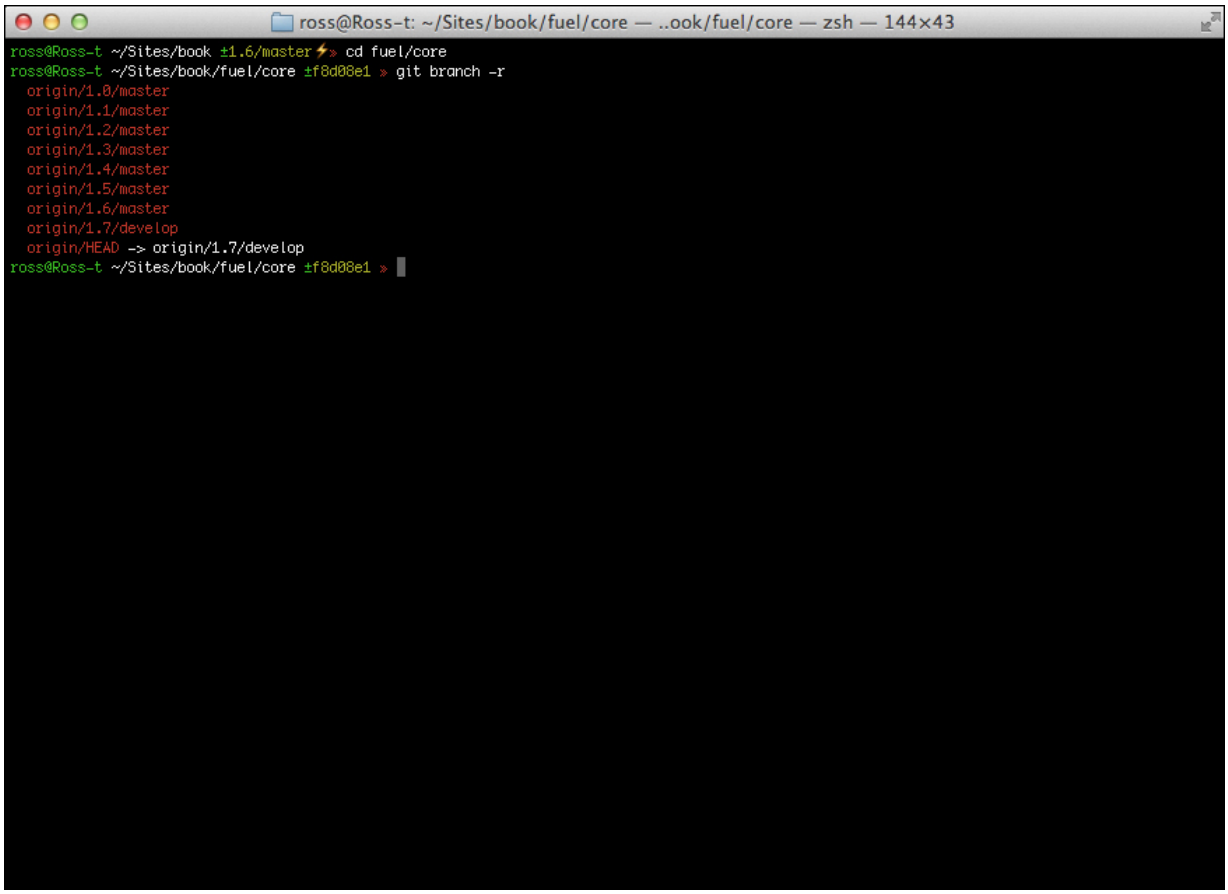
```
$ cd ~/Sites/book
$ git submodule
```



As you can see, there are six submodules set up and used for each FuelPHP project.

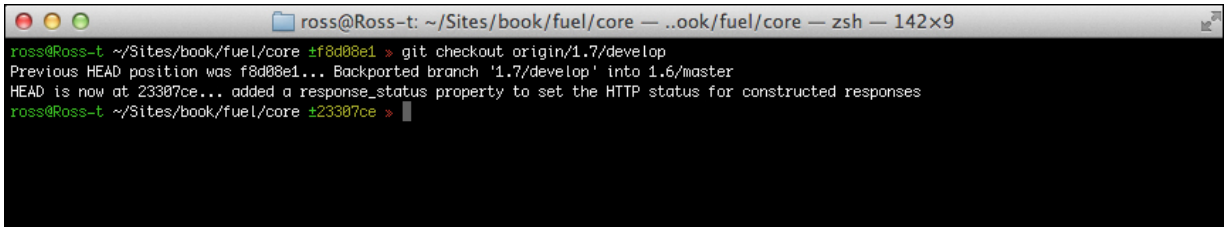
If you would like to check what other versions of submodules are available, navigate to the folder for the submodule and then run the `git branch -r` command, as shown:

```
$ cd fuel/core
$ git branch -r
```



We can then copy the code from one of the other branches to test new features, or to roll back to a previous version of the code. For example, let's see what happens when we use the development version of FuelPHP:

```
$ git checkout origin/1.7/develop
```



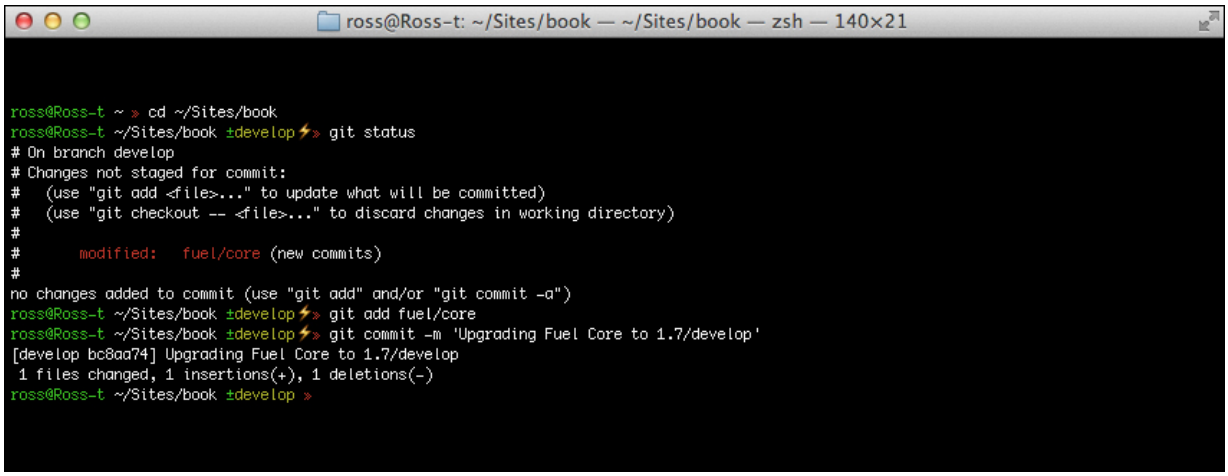
Each submodule acts like its own repository and doesn't take into account the main project repository. If you want the main project to take into account the submodule changes, simply commit all the

changes to the submodule, then navigate to the main project folder and commit the changes to the project repository, as shown:

```
$ cd ~/Sites/book
$ git status
$ git add fuel/core
$ git commit -m 'Upgrading Fuel Core to
1.7/develop'
```

Note

`fuel/core` is not the same as `fuel/core/`.



Committing your code

When the project is set up, the Git settings will want to send the code to the FuelPHP repository. So, the first thing to do is to change this so that it goes to your project. We will be demonstrating this using GitHub, which is the same place where FuelPHP is stored.

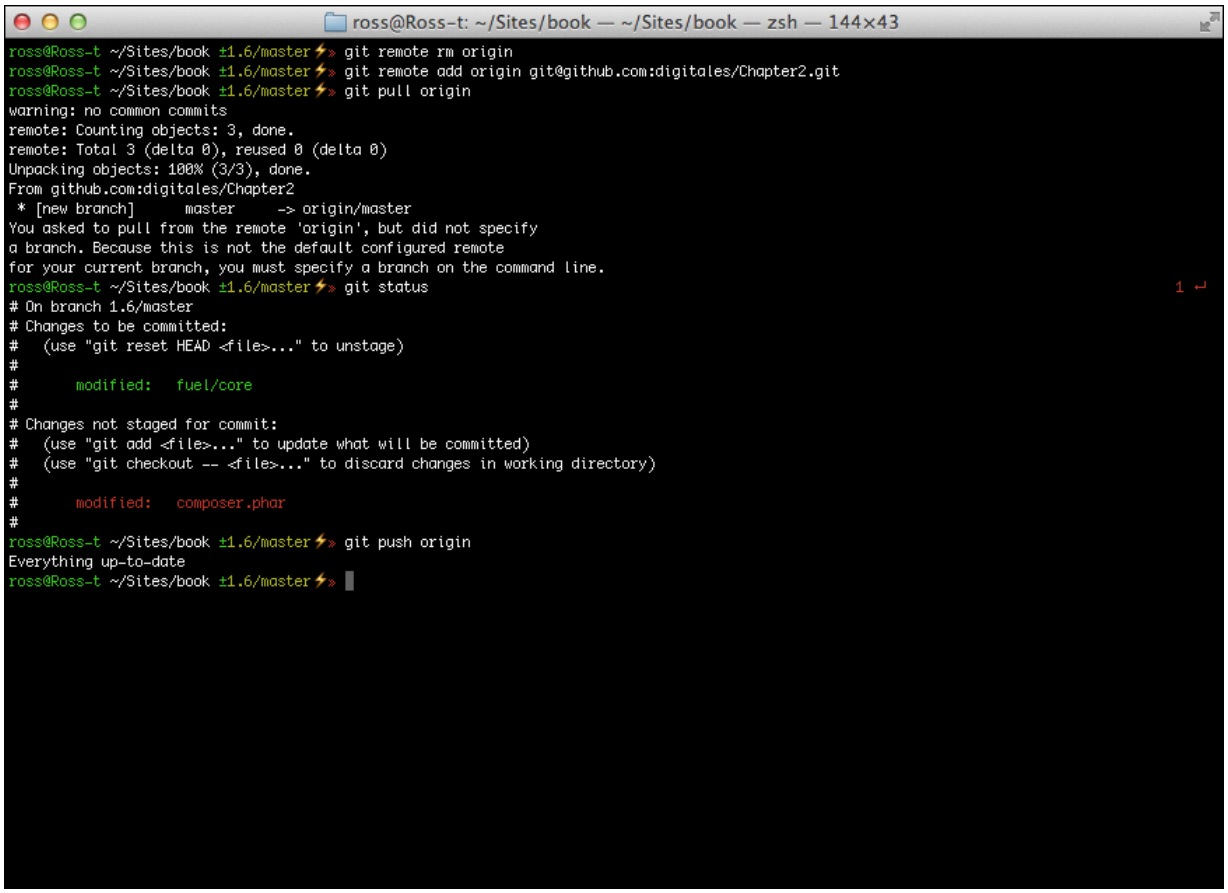
First, create an account on GitHub (https://github.com/new), and then follow the instructions to create a repository. Once you have created the repository, copy the repository address, for example, `git@github.com:digitales/Chapter2.git`; you will need this shortly.

Note

Bitbucket.org is a similar service, except it will allow you unlimited private repositories.

Once you have created your repository and copied the repository address, it's time to go back to your terminal. Navigate to your project directory in your terminal and then add a remote to the repository, for example:

```
$ cd ~/Sites/book
$ git remote rm origin
$ git remote add origin
git@github.com:digitales/Chapter2.git
$ git pull origin
$ git push origin master
```



Now that we've updated the origin, it's time to delve into a little about submodules, followed by configuration and some basics for the production environment configuration.

Tip

Downloading the example code

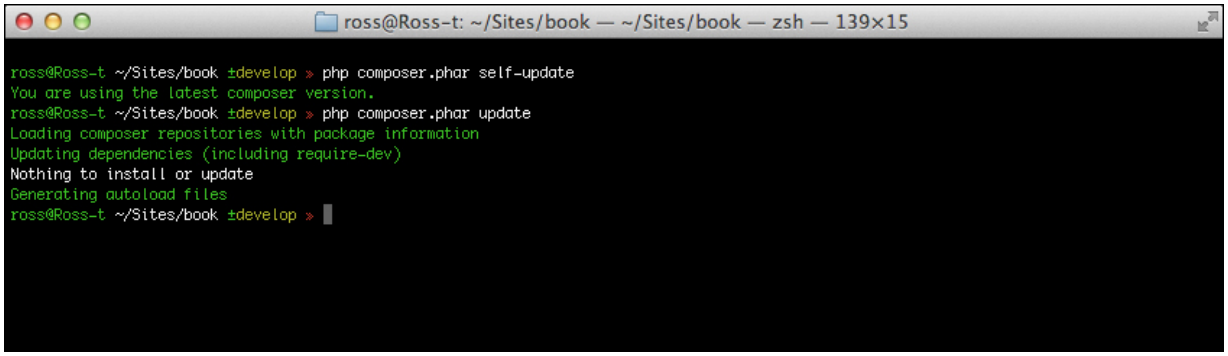
You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Composer – the package manager

With more recent versions of FuelPHP, the Composer package manager is used to dynamically pull dependencies from **Packagist**, Github, or from a custom defined location. It's controlled using the `composer.json` file, which you will find in the root folder of your project's FuelPHP installation.

Normally, you'd need to manually install Composer, but FuelPHP includes the `composer.phar` library so that you can run Composer directly:

```
$ php composer.phar self-update
$ php composer.phar update
```



> **Note**
>
> If you don't execute this step, FuelPHP will not start, as vital components of the framework are now being loaded by using Composer.

Configuration

FuelPHP works on a configuration over convention method, with best practices and guidelines to follow. All application- or project-specific code is stored in the `app/config` folder with the main configuration file consisting of `config.php`. It is worth mentioning that you can pick and choose which configurations to override. Any unspecified keys or values will be loaded from the core

configurations instead. This means that any changes to the default configurations will not be lost when upgrading versions of FuelPHP.

Running in the production environment

When you install FuelPHP, it defaults to thinking that it is in a development environment, but this can be changed quickly by setting the environment.

This can either be done in the virtual host (or similar) for the domain, or it can be done via the `.htaccess` file in the public folder for the application, using the following code:

```
Set FUEL_ENV production
```

By default, the environment will be set to a development environment in both the applications and in the command-line tasks. Running command-line tasks in the production environment will be covered in later chapters in this book.

Performing migrations

Migrations are a great way of ensuring that the database is consistent in different environments, or between team members. It provides a systematic way of updating the datastore structure. Gone are the days of manually running SQL statements on a database and then wondering if the correct database structure has been updated. At any point of developing a site, the database can be changed moving forward or rolled back to an older version of the database structure.

Examples of migrations will form part of the project later in this book.

Summary

In this chapter we have set up our development environment and have been introduced to the Git source control, along with some of its benefits. We have briefly examined tailoring a project to different environments and configuring the source control to take this into account with different branches. We have also installed FuelPHP.

In the next chapter, we will examine the FuelPHP architecture before building a demo application in Chapter 4, *Demo Application*.

Chapter 3. The Architecture

Before we start building anything with our fresh-installed version of FuelPHP, let's first take a look at some of the main aspects of the architecture. We will also cover some of the places where you should put your code for the project and then follow this with examples in the next chapter.

In this chapter, we will cover the following topics:

- Environments, constants, and configuration
- Apache configuration
- The FuelPHP bootstrap
- Models, views, and controllers

Environments, constants, and configuration

As any developer will tell you, it's never a good idea to edit files directly in the live production environment. To this end, FuelPHP has the notion of environments baked into its core. Environments allow for different configurations in each stage of the project's life cycle. FuelPHP supports four environment settings, which are as follows:

- `\Fuel::DEVELOPMENT`: This is the default environment setting and also where you'll start
- `\Fuel::TEST`: This is where you can run code with test data
- `\Fuel::STAGING`: This is where you get client approval and acceptance
- `\Fuel::PRODUCTION`: This is the live environment

If the environment is not set, the code will run in the development mode. Setting the environment can be done in several ways.

Server and Apache configuration

This section will teach you what probably is the easiest way of setting the environment variable, but only if you have access to the Apache configuration or virtual host file for the project domain. In the configuration or virtual host file, simply include the following code:

```
SetEnv FUEL_ENV production
```

> **Note**
>
> The `FUEL_ENV` code needs to be in uppercase as it's a PHP constant

The bootstrap PHP file

If you don't have access to set the server configuration and want to get it, an alternative is to set the environment in the application bootstrap. This can be done in the `bootstrap.php` file located in `fuel/app/`, using the following code:

```
Fuel::$env = ( isset( $_SERVER['FUEL_ENV'] )?
$_SERVER['FUEL_ENV'] : Fuel::PRODUCTION ;
```

Configuration

Once you have your environment configured, it's worth thinking about the configuration for the project. FuelPHP follows a hierarchy when auto-loading configurations. This really comes in handy for database settings, meaning that your production connections can be more secure for the production and staging environments. In fact, the team doesn't even need access to the database credentials for them to work on the project. The configuration directory structure can resemble the following code:

```
app/
    config/
        db.php
        development/
            db.php
        staging/
            db.php
        test/
            bd.php
        production/
            db.php
```

The settings in the environment directories are treated as more important than those in the higher directories giving fine-grained configuration controls with a fallback. This is great for testing and ensuring that each environment can be configured differently, for example, third-party API connection details.

The same hierarchy is followed for package configurations. In this case, the package configuration in the application configuration folder is treated with greater importance than the configuration in the package directory. This allows the default options in packages to be overridden with a configuration file in the `app/config` folder.

Constants

In relation to packages, the details of the path to their folders take the form of constants. The following are PHP constants set in the bootstrap file:

- `APPPATH`: The path to the application directory where your application code and directories sit
- `COREPATH`: The path to the core FuelPHP files
- `DOCROOT`: The public folder for assets and the `index.php` file
- `PKGPATH`: The package directory path
- `VENDORPATH`: The path to the composer root directory

Models, views, and controllers

No look at the architecture would be complete without mentioning the **model-view-controller** (**MVC**) design pattern. If you have written PHP before, or looked at any of the other numerous frameworks out there, you would have heard of the MVC pattern. It allows the logical separation of code. The controller handles the logic while the model ensures the consistency of data and also performs the interaction with the data store. The view presents the results of the controller and model to the user. So far, so good; why mention the MVC pattern? Well, FuelPHP introduces the ViewModel and some base classes to give you a head start on your project.

Views and ViewModels

Views are stored in the `views` folder within the `app` folder, for example, `fuel/app/views`.

They can be grouped together in subfolders and normally relate directly to a controller action, for example, `login.php`, which is located in `fuel/app/views/user/`, would relate to the `login` method within the `user` controller. It would also have a class name of `user/login`.

It is also worth noting that the views can use variables passed to it from the controller or ViewModel. They can also use any core FuelPHP classes and native PHP functions. To help with the separation of the logic from the presentation it's recommended to only use basic `if` statements and loops.

Security is always important and, as such, all variables passed to a view will be sanitized using the configured output filters—usually `Security::htmlentities()`. This behavior can be changed or disabled from within each controller, as can the default functions that run from the configuration files.

To reduce the memory footprint of FuelPHP apps, views are "lazy-loaded". This means that they will only be rendered when called for or echoed.

We mentioned earlier about ViewModels; these act as the glue between controllers and views. When an application starts to get complex, it can become difficult to decide whether or not a piece of code should go in the controller or view. The code may have little to do with the application logic and more to do with the presentation logic. This is where ViewModels come in use.

ViewModels come into play after the user input has occurred. They then retrieve the data needed for the views. They don't manipulate the data, but can interact with the data store before the view.

Models

The FuelPHP core model (`Model_Crud`) includes the basic CRUD (create, read, update, and delete) functions, making it quick and easy to start working with the data store. It is just a shortcut for working with tables and is not part of the ORM package. The naming conventions are the same for the `Model_Crud` and the `ORM` package; this makes it really easy to update to the ORM package for more complex data interactions. The ORM package includes many advanced pieces of functionalities, ranging from soft deletion to temporal historical record keeping.

Using the ORM package will require you to define more details on the data, such as the relationships between database tables. We'll touch upon some of these when building the example application in the next chapter.

Controllers

Controllers are where the more interesting parts of the application happen. Like other parts of the project, the code has a logical

location within the following folder structure. Controllers go in the `fuel/app/classes/controller` folder.

The router takes the URL, loads the required controller, and then passes the request to the corresponding action or method within the controller. For example, navigating to the `http://project.dev/home/index` URL would call the `index` method in the home controller. The structure of the controller would look something like the following:

```
class Controller_Home extends Controller
{
    public function action_index()
    {
        $data['css'] =
Asset::css(array('reset.css','site.css'));
        return Response::forge(
View::forge('home/index'));
    }
}
```

Note

Actions or methods within the controller are prefixed with action_, this is to help you use the reserved PHP method names, such as `list`, and still have a logic name for the controller methods. Other methods can be created, but without the `action_` prefix, they can only be called directly rather than through the URL.

If you are building an API or setting up an AJAX system, it is possible to separate out the methods. In the following example, the `action_example` would take GET requests while POST requests would go to the `post_example` method as follows:

```
class Controller_Home extends Controller
{
     public action_example(){
       // Add your code here
     }
```

```
        public post_example(){
          // Add your code here
        }
}
```

This functionality comes into play with the base controller templates, which you can use as a starting point when creating your application. The separation of the `action_` and `post_` methods will only work with the hybrid or rest base controllers. There are four base templates—base, template, rest, and hybrid.

Template

The template controller is an extension of the base controller mentioned previously, that has template support built-in using predefined `before()` and `after()` methods. It can be used to wrap your view in a layout with a header and footer. Your layout template should be stored in `fuel/app/views/template.php`.

Rest

Like the template controller, the rest controller is an extension of the base controller. Instead of built-in template support actions, it has RESTful support built into it, making it easier to build APIs.

Note

If, in your rest controller, you have added a `before()` or router method, you will need to call the parent method `parent::before()` for all of it to work.

Hybrid

The hybrid controller brings together the rest and template controllers.

Summary

In this chapter, we have run through some of the basic parts of the FuelPHP architecture. We have seen that FuelPHP gives a great starting point with extendable controllers and models. These provide a great head start to most projects and introduce some functionalities, which is a first for any PHP framework; for example, the temporal ORM model.

In the next chapter, we will start using this knowledge to create a demo application.

Chapter 4. Demo Application

Now that we are through with the introduction to the FuelPHP framework, it's time to build something. We will run through the Oil command-line tool migration, putting it all together in the example application. In this chapter we will be creating a simple blogging application much like WordPress.

In their own words, WordPress is becoming the operating system of the Web. According to founder *Matt Mullenweg*:

> *"WordPress now powers 18.9% of the Web, has over 46m downloads"*

Visit the following link for more information on the article:

http://thenextweb.com/insider/2013/07/27/wordpress-now-powers-18-9-of-the-web-has-over-46m-downloads-according-to-founder-matt-mullenweg/

Now, WordPress has a focus much larger than blogging or simple journals. Although a simple application or website, creating a simple blog will demonstrate a lot of the FuelPHP features—from database migrations, pivoting ideas, and code to full temporal models to store revisions of journal entries.

Before starting to code, let's first think about what to build and what the minimum viable product will be.

In this chapter, we will cover the following topics:

- Creating and running database migrations
- Using Oil to create models
- Using Oil to create controllers
- Installing and using the HTML5 Boilerplate

- Putting it all together to create an administration system with scaffolding
- Using the Oil command-line console

Getting started

Just like WordPress, we're going to start small and demonstrate some of the FuelPHP tools that make your life easier while developing our project with FuelPHP.

Before creating your project, first create a source control repository. In this example, we'll use GitHub (https://github.com), but alternatives such as **Bitbucket** (https://bitbucket.org) or **beanstalk** (http://beanstalkapp.com) may be more suitable for you.

1. To begin with, log in to your account and then create a new project/repository. For the purpose of this application we can choose the public option as it's free with GitHub. In this example, we'll call our project `journal`. Make sure to record the repository URL from the right–hand side of the page—you will need it shortly.
2. Now, let's create the project on our development machine, but first you need to navigate to your `home` folder using the following command line:

```
$ cd ~/
```

3. Then, navigate to the `Sites` folder using this command:

```
$ cd ~/Sites
```

4. If you don't have a `Sites` folder, let's create one:

```
$ mkdir ~/Sites
```

5. Once in your Sites folder, run the following Oil command:

```
$ php oil create journal
```

Note

Please note that in these examples, the $ represents the start of a new line in the terminal. When running the commands you should only need the text after the `$` for example `php oil create journal`.

If you are using Apache on your local development machine, the next step will be to create the virtual hosts (**vhosts**) for the new site and then modify the hosts file. You need to have an Apache configuration with a `sites-available` folder and a `sites-enabled` folder. Then, a small script from a friend will prove to be invaluable. It can be installed with the following command:

```
$ sudo cd /usr/local/bin && sudo git clone
git://github.com/maartenJacobs/quickhost.git
quickhost files && sudo mv quickhost files/* . &&
sudo rm -Rf quickhost_files
```

Note

This script is a set of PHP functions that set up the Apache vhosts and the hosts file on your computer. All of the actions are simple, but they allow you to get it set up quicker. The code can be viewed publicly at the following link:

https://github.com/maartenJacobs/quickhost

Or or the more information can be found at https://github.com/digitales/quickhost

Once the scripts are installed, we will be able to run the following set of commands to set up Apache so that we can use a local domain `http://journal.dev`:

```
$ cd ~/Sites/journal/public
$ sudo quickhost journal.dev
```

The reason for using `sudo` is to allow Apache to be gracefully restarted and to modify the hosts file. This will map the domain name to the localhost IP address (`127.0.0.1`). In this example, we have chosen the `.dev` **top-level domain** (**TLD**) to differentiate it from the production and the staging environments. We could have chosen `.local`, but this can cause conflicts with the Active Directory (http://en.wikipedia.org/wiki/Active_Directory) on some operating systems, especially on Mac OS X.

Note

This has been tested on Mac OS X and it should work on *nix, but may not work on Windows environments without changing the scripts.

We'll need a data store for the journal entries, so let's configure it now. Load the `db.php` configuration found at `fuel/app/config/development/db.php`.

As mentioned before, FuelPHP has the notion of environments, so we can add the development database configuration to source control and it won't affect the other environments. Therefore, your `db.php` file should look something like this:

```
<?php
return array(
    'default' => array(
        'connection' => array(
            'dsn' =>
'mysql:host=localhost;dbname=fue_dev',
            'username' => 'root',
            'password' => 'root',
        ),
        'profiling' => true,
```

```
    ),
);
```

Note

I tend to setup a separate database users for each project. This ensures that no project can touch the database of another project. In my db.php I'm using a dbname, username and password of journal_dev so you are welcome to change your version of the db.php file.

Although not recommended on the live environment, enabling `profiling` can help while developing the project. The example configuration has it enabled.

We have the code and the repository, now let's link them:

```
$ cd ~/Sites/journal
$ git remote rm origin
$ git remote add origin <repository url>
$ git pull origin
$ git add ./
$ git commit -m 'Initial commit'
$ git push origin master
```

Replace the `<repository URL>` above with your repository URL. Now, is also a good time to set up a few branches for the different environments: `develop`, `staging`, and `production`:

```
$ git checkout -b develop
$ git push origin develop
$ git checkout -b staging
$ git push origin staging
$ git checkout -b production
$ git push origin production
$ git checkout develop
```

Continuing on the setup theme, let's set the database tables.

Creating a database table

Before creating the tables in the database, let's first detail what data will be stored and which initial fields will be there in the tables.

For blogging, we will have blog posts or entries and these will have a published date. They will be categorized to enable grouping of similar themes. Each post will be associated with an author (user) and will have content and an excerpt. We can add free-form tagging later.

Entries

The following fields exist in the `Entries` table:

- `id` (int): This will be the primary identifier
- `name` (varchar)
- `slug` (varchar)
- `excerpt` (text)
- `content` (text)
- `published_at` (int)
- `created_at` (timestamp): This will be a timestamp of when the record was created
- `updated_at` (timestamp) - This will be a timestamp of when the record was last updated

Categories

The following fields exist in the `Categories` table:

- `id` (int): This will be the primary indentifier
- `name` (varchar)
- `slug` (varchar)
- `created_at` (timestamp): This will be a timestamp of when the record was created

- `updated_at` (timestamp): This will be a timestamp of when the record was last updated

Users

The following fields exist in the `Users` table:

- `id` (int): This will be the primary indentifier
- `name` (varchar)
- `username` (varchar)
- `password` (varchar)
- `email` (varchar)
- `last_login` (timestamp)
- `login_hash` (text)
- `profile_fields` (text)
- `created_at` (timestamp): This will be a timestamp of when the record was created
- `updated_at` (timestamp): This will be a timestamp of when the record was last updated

We also need a link table so that multiple entries can be categorized in the same categories.

categories_entries

The following fields exist in the `categories_entries` table:

- `id` (int): This will be the primary identifier
- `category_id` (int): This will be the primary identifier of the category
- `entry_id` (int): This will be the primary identifier of the entry
- `created_at` (timestamp): This will be a timestamp of when the record was created
- `updated_at` (timestamp): This will be a timestamp of when the record was last updated

You may have noticed that each table has an `id`, a `created_at` table, and an `updated_at` field. These are automatically added by the FuelPHP Oil tool and can prove to be useful when linking or forming relationships between data objects. They do add a minimal overhead, but this should not be of concern at this stage, as the extra storage required is minimal.

Let's create some migrations using the FuelPHP Oil tool:

```
$ php oil generate migration create_entries
name:string slug:string excerpt:text content:text
published_at:timestamp
```

This will create the migrations to assemble the posts database table. There is no need to detail the `id`, `created_at`, or `updated_at` fields as these will be automatically generated. If you don't need these extra time-based fields, you can append `--no-timestamp` at the end of the `generate` command.

Tip

We are running through migration examples here and we will use the proper entries table when generating the models later in this chapter.

Migrations and Oil

When quickly creating projects and getting the bare bones together, the FuelPHP Oil tool can handle some of the more repetitive tasks for you, allowing you to concentrate on other aspects of the project. This ranges from setting up the database tables (as demonstrated before) to creating models and controllers. The FuelPHP Oil tool can also be used to scaffold an administration system that is ready to be used to administer contents with a full user-authentication system. It'll provide the starting point; although the results won't always be pretty, it will be functional.

This allows you to test ideas quickly and iterate towards the perfect code. We'll run through the individual features of Oil to create models, controllers, and database migrations. This will give us more time to tweak and create a more polished project/journal.

One of the most useful features of Oil is the ability to migrate the database structure. These can be used to ensure the consistency between environments. When deploying code, it's possible to run migrations on the database as well.

Migrations can be used to rename a table, add and remove fields or even tables. Oil has the notion of magic migrations. These start with a keyword's prefix, making it easy to see what they will be doing:

```
$ php oil generate migration create_users
name:text
$ php oil generate migration
rename_table_users_to_people
$ php oil generate migration add_profile_to_people
profile:text
$ php oil generate migration
delete_profile_from_people profile:text
$ php oil generate migration drop_people
```

Tip

You need to be aware of the keywords when creating a migration. Keywords include `create_`, `rename_`, `add_`, `delete_`, `rename_`, and `drop_`.

Migrations are stored in `fuel/app/migrations` and a list of performed migrations is stored in `fuel/app/config/<environment>/migrations.php`.

Additionally, the migration's database table records the current state of the project. Running migrations is as a simple as typing the following command:

```
$ oil refine migrate
```

Tip

You can simply type `r` and `g` to run the refine and generate Oil actions.

If you want to migrate up or down, run the following commands:

```
$ oil r migrate:up
$ oil r migrate:down
```

These commands can be worked into a deployment script, so that migrations can automatically be run when the new code is added to different environments. This is particularly useful for avoiding errors on the production environment.

Before progressing to generating models and their database tables, let's drop the posts table we created previously:

```
$ php oil g migration drop_posts
```

```
$ php oil r migrate
```

Models

Now that we have a brief understanding of migrations and Oil, let's put Oil to use. We'll create a model along with their corresponding database table details with the following command:

```
$ php oil generate model entry name:string
slug:string excerpt:text content:text
published_at:timestamp
```

This will create both the model and migration. Let's open the newly created `entry.php` model, located at `fuel/app/classes/model/`:

```
<?php
class Model_Entry extends \Orm\Model
{
    protected static $_properties = array(
        'id',
        'name',
        'slug',
        'excerpt',
        'content',
        'published_at',
        'created_at',
        'updated_at',
    );

    protected static $_observers = array(
        'Orm\Observer_CreatedAt' => array(
            'events' => array('before_insert'),
            'mysql_timestamp' => false,
        ),
        'Orm\Observer_UpdatedAt' => array(
            'events' => array('before_save'),
            'mysql_timestamp' => false,
        ),
    );

    protected static $_table_name = 'entries'
}
```

You'll notice in the newly generated model that it is extending the ORM Model. For this to work, we'll need to include the ORM package in the autoloader. This is a good chance to introduce the FuelPHP autoloader configuration. Load the configuration file located at `fuel/app/config/config.php`. Look for the `'always load'` section and uncomment the `'always_load'` array, as shown in the following:

```
'always_load' => array(
    'packages' => array(
        'orm',
    )
),
```

When you install new packages, simply add them in the `always_load` array to automatically load them. Sometimes, a package is needed only in a certain method or class. In this instance, however, we can load the package only when needed. More information on this can be found at the following link:

http://fuelphp.com/docs/classes/package.html#method_load

Now, let us go back to the `entry` model. You'll notice the `$_properties` array, this will make it easier to call variables from the model layer. If you add a new database field, it'll need to be added to the `properties` array. The ORM package includes the notion of observers, these are actions that can execute methods or functions at different times. The ones used in the model are `before_insert` and `before_update`. These are currently being used to create the timestamps at the time of adding or updating entries.

The other observers are listed in the documentation; one that will be useful for use is `Observer_Slug`. This observer will handle turning the entry title to a URL safe and avoid the need to rewrite pre-existing functionality. Let's add this observer to our entry model by adding the following snippet to the `observers` array:

```
'Orm\Observer_Slug' => array(
    'events' => array( 'before_insert',
'before_update'),
```

```
        'source' => 'name',
        'property' => 'slug',

    ),
```

Now, when we save an entry, the slug will automatically be updated.

There are some other database tables we have to create. Let's do this now and then we can create relations between their models to make our lives easier:

```
$ php oil generate model category name:varchar
slug:varchar
$ php oil generate model categories_entries
category_id:int entry_id:int
```

For the `entry` model, add `Observer_Slug` to the new category model.

For the `user` table, we are going to use the Auth package, as it includes the user authentication functionality. To do this, we need to copy the default Auth package configuration to our application configuration.

```
$ cp
~/Sites/journal/fuel/packages/auth/config/auth.php
~/Sites/journal/fuel/app/config/auth.php
```

At this stage, we should change the salt value in the `auth.php` configuration file. Once this is done, we have to run the following command to create the necessary `auth` tables:

```
$ php oil refine migrate --packages=auth
```

Let's run the migration tool to make sure the database is updated:

```
$ php oil r migrate
```

Now that we have the database structure established, we can relate the models with each other. This is made possible by the ORM model, which natively supports the following relationships:

- `belongs_to`: This model has the primary key for the relation and it belongs to a single related object.
- `has_one`: Its primary key is stored in another table that belongs to this model. It has a single relation only.
- `has_many`: Its primary key is saved in multiple results of another model. Each other model will need to belong to this one. It can have many relations.
- `many_to_many`: This keeps track of the many-to-many relations using a linking table. This table keeps a record of the primary key pairs from the tables being linked. It can have, and belong, many models. For example, a blog post can have many categories and a category can be linked to many blog posts.

For the journal, we'll use the `belongs_to`, `has_one`, and `many_to_many` relations. Making full use of the relationships only needs a quick addition to the already-created models. Load the entry model in your favorite text editor. The model can be found at `fuel/app/classes/model/entry.php`.

```
<?php

class Model_Entry extends \Orm\Model
{
    protected static $_properties = array(
        'id',
        'name',
        'slug',
        'excerpt',
        'content',
        'published_at',
        'created_at',
        'updated_at',
    );
```

```
    protected static $_observers = array(
        'Orm\Observer_CreatedAt' => array(
            'events' => array('before_insert'),
            'mysql_timestamp' => false,
        ),
        'Orm\Observer_UpdatedAt' => array(
            'events' => array('before_update'),
            'mysql_timestamp' => false,
        ),
        'Orm\Observer_Slug' => array(
            'events' => array('before_insert'),
            'source' => 'name',
            'property' => 'slug',
        ),
    );
    protected static $_table_name = 'entries';
}
```

Add the following code to the model just before closing the curly brace:

```
 protected static $_belongs_to = array(
            'category' => array(
                'key_from' => 'id',
                'model_to' =>
'Model_Category_Entry',
                'key_to' => 'entry_id',
                'cascade_save' => false,
                'cascade_delete' => false,
            ),
        );

    protected static $_has_many = array(
        'categories' => array(
            'key_from' => 'id',
            'model_to' => 'Model_Category_Entry',
            'key_to' => 'entry_id',
            'cascade_save' => false,
            'cascade_delete' => false,
        )
    );
```

We don't always need to include the full `$_belongs_to` declaration. In most situations, we can simply use the following:

```
protected static $_belongs_to = array('post');
```

With these additions, we are configuring the ORM model to treat the relationship between entries and categories as a many-to-many relationship. Instead of directly linking categories to entries, we are using the `category_entry` link table. This table is named using the singular version of the tables to be linked in an alphabetical order; in this case, the joining of `categories` and `entries` gives us a table name `category_entry`. The reason we want to use the link table is because we want to have the ability to assign multiple categories to each entry, so that the categories can be reused without requiring duplication. For example, if we retrieve an entry, the list of categories would be obtained in the following manner:

```
$entry = Model_Entry::find(1)

$categories =$entry->categories;
```

In this example, we are looking for an entry with its ID set as `1`. If this is found, the full entry object will be returned, otherwise it will return false.

In addition to the many-to-many relationship, let's link the users to the entries that they authored.

Firstly, we need to add the `user_id` to the entries table:

```
$ php oil g migration add_user_id_to_entry
user_id:int
$ php oil r migrate
```

We need to add `user_id` to the entry model `$_properties`:

```
protected static $_properties = array(
        'id',
        'name',
        'slug',
```

```
            'excerpt',
            'content',
            'published_at',
            'created_at',
            'updated_at',
        );
```

Since the relationship maps from `user_id` to the primary `id` key in the `user` table, we can just define the user as a part of the `$_belongs_to` array. Everything else will fall into place once the reciprocal relation is added to the `Model_User` model:

```
$_belongs_to = array(
    'user',
    'category' => ...
);
```

Talking about the `Model_User`, it needs to be created:

```
$ php oil g model user username:string
password:string group:string email:string
last_login:int profile_fields:text --no-migration
```

Although most of the user functionality are done from within the Auth package, adding the user model is necessary for the `Model_Entry` relation with the user data. Therefore, load the `user` model `user.php` located at `fuel/app/classes/model/` and add the following array:

```
protected static $_has_many = array('entry');
```

The `category` and `category_entry` models also need the relations creating. To do this, navigate to the `category.php` file located at `fuel/app/class/mode/`:

```
<?php

class Model_Category extends \Orm\Model
{
    protected static $_properties = array(
```

```
            'id',
            'name',
            'slug',
            'created_at',
            'updated_at',
        );

        protected static $_observers = array(
            'Orm\Observer_CreatedAt' => array(
                'events' => array('before_insert'),
                'mysql_timestamp' => false,
            ),
            'Orm\Observer_UpdatedAt' => array(
                'events' => array('before_update'),
                'mysql_timestamp' => false,
            ),
        );
        protected static $_table_name = 'categories';

        protected static $_belongs_to = array(
                'entry' => array(
                    'key_from' => 'id',
                    'model_to' =>
    'Model_Category_Entry',
                    'key_to' => 'category_id',
                    'cascade_save' => false,
                    'cascade_delete' => false,
                ),
            );

        protected static $_has_many = array(
            'entries' => array(
                'key_from' => 'id',
                'model_to' => 'Model_Category_Entry',
                'key_to' => 'category_id',
                'cascade_save' => false,
                'cascade_delete' => false,
            )
        );
    }
```

The following is the listing for the `entry.php` file located at `fuel/app/classes/model/category/`:

```
    <?php
```

```
class Model_Category_Entry extends \Orm\Model
{
    protected static $_properties = array(
        'id',
        'category_id',
        'entry_id',
        'created_at',
        'updated_at',
    );

    protected static $_observers = array(
        'Orm\Observer_CreatedAt' => array(
            'events' => array('before_insert'),
            'mysql_timestamp' => false,
        ),
        'Orm\Observer_UpdatedAt' => array(
            'events' => array('before_update'),
            'mysql_timestamp' => false,
        ),
    );
    protected static $_table_name =
'category_entries';

    protected static $_belongs_to = array(
        'category',
        'entry',
    );
}
```

Now that we have the models and the migrations, it would be a good idea to add them in the source control. Once that's done, let's start creating the controller to make use of the models and then the views (including forms).

Controllers

Controllers, like the models, can be created using Oil. The main controllers will be for managing entries and categories. These will come in two flavors, the publicly viewable site controllers and the administration system for the entries and categories tables.

Our controllers probably won't be needed to handle RESTful requests, at least initially. So, we should extend the `Controller_Template` and as luck would have it, that is the way that the Oil tool is set up to run. When we create a controller with Oil, it'll be created to extend the template controller. We only need to think about what actions and methods we need for doing this. As we will primarily be displaying the entries and categories information for the blog, let's start with the `index` and `view` actions:

```
$ php oil g controller entry index view
```

This will create the controller, a template file, and the views needed for the methods—`index` and `view`.

Let us have a look at `entry.php`, which is located at `fuel/app/classes/controller/`:

```
<?php

class Controller_Entry extends Controller_Template
{
    public function action_index()
    {
        $data['entries'] =
Model_Entry::find('all');
        $this->template->title = "Entries";
        $this->template->content =
View::forge('entry/index', $data);

    }
```

```
    public function action_view($id = null)
    {
        is_null($id) and
Response::redirect('entry');

        if ( ! $data['entry'] =
Model_Entry::find($id))
        {
            Session::set_flash('error', 'Could not
find entry #'.$id);
            Response::redirect('entry');
        }
        $this->template->title = "Entry";
        $this->template->content =
View::forge('entry/view', $data);

    }
}
```

There are a few interesting things to notice in the controller generated. The first is that each method that relates directly to a URL is prefixed with `action_`. This makes it easier to see which methods relate directly to a URL and which methods cover other functionality within the controller

You will also notice that the active view is being passed to the view in the `$data` array. This is used to pass variables to the view, allowing the logic to be done prior to view rendering. This leaves the views purely for presentation, as they should be. The keys in the `data` array will be translated to variables in the view, for example, `$data['subnav']` will be callable via the `$subnav` variable in the view.

The `Controller_Template` class also has a couple of other methods—`before()` and `after()`—which are particularly useful. The `before()` method can be used for user authentication for the administration system, and this will be demonstrated later in this chapter. We will be using it for the administration system to ensure that only authenticated users can access the administration system. When using the `before()` method in your controller, make sure you call the `parent::before()` method, for example:

```
class Controller_Entry extends Controller_Template
{
    // Essential, don't forget this method.
    public function before()
    {
        parent::before();


        // Your code
    }
    .....
}
```

Views

When creating the controller with the Oil tool, a template and views are also created. Let's have a look at one of the views, load `fuel/app/views/entry/index.php`.

You will notice the lack of opening and closing HTML body in the view. The test of the presentation elements are handled by the `template.php` file found in the root of the views folder. Before looking at `template.php`, let's discuss some of the parts of the `index.php` file:

```
Arr::get( $subnav, "index");
```

As shown in the preceding line of code, it is using the core FuelPHP `Arr` class, which is a set of helper functions for working with arrays. In this instance, the `get` method is used. This allows you to check for a given key in an array, it then returns false if the key can't be found. The view is using this functionality to output the `'active'` style class for the active page / view.

The second core class being used is the `Html::anchor()`. This class provides a large selection of HTML tags, and it ensures that all tags used are valid with the `Doctype` declaration. We are using the `anchor` method to output a link to the view. The first value is the link, while the second is the display title (the bit in between the `a` tags). One reason for using this helper method is to ensure that the links work no matter what the URL of the environment is, or whether the app is installed in a sub folder or subdomain. Now, let's have a quick look at the `template.php` file found at `fuel/app/views/template.php`.

By default, the template uses the Twitter bootstrap to give a quick starting point for your projects. FuelPHP expects a CSS and JS folder within the public folder of your project. Loading CSS and JavaScript files is as easy as using the following code in your template:

```
<?php echo Asset::css('bootstrap.css');?>
<?php echo Asset::js('bootstrap.js'); ?>
```

As with frameworks such as Ruby on Rails, FuelPHP implements a flash session for passing variables and values between page loads. By default FuelPHP has two flash sessions, named `success` and `error` by default. You can of course add as many flash sessions to FuelPHP as you need. The following is an example:

```
<?php Session::set_flash('warning', 'Warning
Message') ?>
```

The following examples are both ways by which the flash sessions are output in the template:

```
<?php if (Session::get_flash('success')): ?>
    <div class="alert alert-success">
        <strong>Success</strong>
        <p>
            <?php echo implode('</p><p>',
e((array) Session::get_flash('success'))); ?>
        </p>
    </div>
<?php endif; ?>
```

The rest of `template.php` is a fairly straightforward HTML. Creating websites that respond to the screen size of the visitor is proving to be very a popular way of supporting mobile devices. So we should start thinking about the mobile-sized screen first before adding layouts for the desktop. To help with this, we are going to make use of the HTML5 Boilerplate (http://html5boilerplate.com). We're going to customize the version of the Boilerplate and this can be done via the http://www.initializr.com/ companion site. To do this carry out the following steps:

1. On the Initializr website, there should be 3 options: Classic H5BP, Responsive, and Bootstrap. Choose the **Classic H5BP**

option (the gray button) and then the **Responsive Bootstrap** 2.3.2 template option. Finally click on **Download it!**.
2. Copy the resulting downloaded file to the public folder of your journal application. Copy the `js`, `css`, and `img` folders into your assets folder.
3. Now, open the `index.html` file in your text editor and edit it. The next step is a case of taking elements from the `index.html` file and adding them to your `template.php` file.

   Here is an example of what you will end up with:

```
<!DOCTYPE html>
<!--[if lt IE 7]>      <html class="no-js lt-
ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>         <html class="no-js lt-
ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>         <html class="no-js lt-
ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js">
<!--<![endif]-->
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible"
echo Asset::css('bootstrap.min.css'); ?>
content="IE=edge,chrome=1">
        <title><?php echo $title; ?></title>
        <meta name="description" content="">
        <meta name="viewport"
content="width=device-width">
        <?php
        <style>
            body {
                padding-top: 60px;
                padding-bottom: 40px;
            }
        </style>
        <?php echo Asset::css('bootstrap-
responsive.min.css'); ?>
        <?php echo Asset::css('main.css'); ?>
        <?php
Asset::add_path('assets/js/vendor/', 'js'); ?>
        <?php echo Asset::js('modernizr-
2.6.2.min.js'); ?>
    </head>
```

```
    <body>
        <!--[if lt IE 7]>
            <p class="chromeframe">You are
using an <strong>outdated</strong> browser.
Please <a
href="http://browsehappy.com/">upgrade your
browser</a> or <a
href="http://www.google.com/chromeframe/?
redirect=true">activate Google Chrome
Frame</a> to improve your experience.</p>
        <![endif]-->
        <div class="navbar navbar-inverse
navbar-fixed-top">
            <div class="navbar-inner">
                <div class="container">
                    <a class="btn btn-navbar"
data-toggle="collapse" data-target=".nav-
collapse">
                        <span class="icon-
bar"></span>
                        <span class="icon-
bar"></span>
                        <span class="icon-
bar"></span>
                    </a>
                    <a class="brand"
href="/">Journal</a>
                    <div class="nav-collapse
collapse">
                        <ul class="nav">
                          <li class="active">
<a href="/">Home</a></li>
                          <li><a
href="/entry">Entries</a></li>
                          <li><a
href="/category">Categories</a></li>
                        </ul>
                    </div><!--/.nav-collapse -
->
                </div>
            </div>
        </div>
        <div class="container">
            <h1><?php echo $title; ?></h1>
            <?php if
(Session::get_flash('success')): ?>
                <div class="alert alert-
success">
```

```
                    <strong>Success</strong>
                    <p>
                    <?php echo implode('</p>
<p>', e((array)
Session::get_flash('success'))); ?>
                    </p>
                </div>
            <?php endif; ?>
            <?php if
(Session::get_flash('error')): ?>
                <div class="alert alert-
error">
                    <strong>Error</strong>
                    <p>
                    <?php echo implode('</p>
<p>', e((array) Session::get_flash('error')));
?>
                    </p>
                </div>
            <?php endif; ?>
            <hr>
            <div class="span12">
                <?php echo $content; ?>
            </div>

            <footer>
                <p>&copy; Journal <?php echo
date( 'Y' ); ?></p>
            </footer>

        </div> <!-- /container -->

        <script
src="//ajax.googleapis.com/ajax/libs/jquery/1.
10.1/jquery.min.js"></script>
        <script>window.jQuery ||
document.write('<script src="js/vendor/jquery-
1.10.1.min.js"><\/script>')</script>
        <?php echo
Asset::js('bootstrap.min.js'); ?>
        <?php echo Asset::js('plugins.js'); ?>
        <?php echo Asset::js('main.js'); ?>
    </body>
</html>
```

Tip

After you are finished with the `index.html` file, make sure to delete it from your public folder; otherwise you may have problems loading the journal website.

Most of the `template.php` file has been mentioned earlier, except for the `Asset::set_path()`. The HTML5 Boilerplate includes the vendor folder for the `modernizr` JavaScript file. By default, the Asset loader will look into the `css` and `js` folders located at `public/assets`, but it won't look at the folders within those directories. We are using the `set_path()` method to include the vendor folder in the JavaScript loading. The second variable in `set_path()` lets the Asset loader know that the new path is a JavaScript folder.

Now, we have a chance to check the site in the browser, for example, `http://journal.dev/entry` or `http://localhost/Site/journal/entry` or `http://127.0.0.1/Site/journal/entry`, depending upon your development machine configuration.

At the moment, the first web page that visitors will see is the FuelPHP welcome page. We should change this so that the entry listing page is the first one the visitors see. Doing this is quick and it can easily be done with routes in FuelPHP. Load the `routes.php` configuration file located at `fuel/app/config/`.

```
<?php

return array(
    '_root_'  => 'welcome/index',  // The default
route
    '_404_'   => 'welcome/404',    // The main 404
route
    'hello(/:name)?' => array('welcome/hello',
'name' => 'hello'),
);
```

You can change the welcome page to the entry listing page by changing `'welcome/index'` to `'entry/index'`.

Back in your browser, navigate to the top level of the journal site, for example, `http://journal.dev` or `http://localhost/Site/journal` or `http://127.0.0.1/Site/journal`. You will notice that your entry index page is showing.

So far, we have covered the basic parts of the controllers and views, and we installed the HTML5 Boilerplate. Now, we should look at the administration of the entries and categories.

Before looking at the administration system, there is one thing left to do on the public site—categories. Let's generate the category index and viewing controller:

```
$ php oil g controller category index view
```

You're probably thinking that it's all fine and dandy to be able to display entries and categories, but how do we actually deal with the administration system? The next section will teach us how to do this.

Generating an administration system using Oil

We're going to use the Oil tool to quickly build an administration system for entries and categories.

Along with creating migrations and controllers, Oil can also be used to scaffold the functionality for you. This comes in two flavors: one for the frontend, like we've already seen, and the other is the administration system with full controls.

Before continuing, we need to rename the `category` and `entry` models. This is because they will be recreated as part of the administration system scaffolding. We rename them so that we have a copy of the previous relations that we set up in those models:

```
$ mv
~/Sites/journal/fuel/app/classes/model/entry.php
~/Sites/journal/fuel/app/classes/model/entry.bak.p
hp
$ mv
~/Sites/journal/fuel/app/classes/model/category.ph
p
~/Sites/journal/fuel/app/classes/model/category.ph
p
```

Now, let's run the administration scaffold Oil command:

```
$ php oil g admin entry name:varchar slug:varchar
excerpt:text content:text published_at:int
user_id:int
$ php oil g admin category name:varchar
slug:varchar -force
```

The `-force` element is needed to rerun the shared files in the administration system so that the category files are created.

Oil has created a few files for you, but, unfortunately, it does not know about the category and entry relationship or the `Observer_Slug` Property. As such, we need to relate back to the `Entry and Category` models, we will also need to add back `Observer_Slug`. This process will be straightforward with the recently renamed entry and category models.

While editing the models, we need to remove or comment out the following:

```
$val->add_field( 'slug', 'Slug',
'required|max_length[255]');
```

We also need to do the same with the references in the administration entry and category controllers. `Observer_Slug` that is defined in the models will handle the slug.

Therefore, you need to check lines 32, 66, and 86 of `category.php` located at `fuel/app/classes/controller/admin/;`, and also check the lines 32, 69, and 92 of `entry.php` located at the same folder.

As a part of the scaffolding process, a couple of forms have been created and are shared between the creation and editing of both entry and category views. Load `fuel/app/views/admin/category/_form.php` and `fuel/app/views/admin/entry/_form.php`. In both forms, comment out the references to slug as it is not needed.

In your browser, navigate to `http://journal.dev/admin` or `http://localhost/Site/journal/admin` or `http://127.0.0.1/Site/journal/admin` and you will be redirected to a login form. We haven't set up a user account yet, but Oil comes to the rescue.

Load your terminal and run:

```
$ php oil console
```

This starts an interactive console, which can make use of the core FuelPHP and package code. In this instance, we need to call the Auth package to create a new user.

```
>>> Auth::create_user('admin', 'password',
'email@domain.com', 100 );
1
>>> exit;
```

This command will output the `user_id` of the newly created account. `100` represents the highest default user privilege and role in the Auth package. To exit the console session, simply type `exit` and run it.

Now, you will be able to log in to your newly created administration system. From there, you will be able to add new categories and entries.

Summary

This chapter has given a brief introduction to the project structure and how the FuelPHP Oil command-line tool can help you get set up very quickly.

Oil acts as a scaffolding structure, but you will still need to fill in the gaps. For example, it's possible to retrieve the linked entries and categories from the blog, but we still need to build the necessary administration controls for setting up with the relationships.

In the next chapter, we will look at packages in more detail and then create our own package, which can be shared with the community.

Chapter 5. Packages

In the previous chapter, we used packages to make our lives easier. We used Oil to quickly get up and running, as well as the Auth package that helped with user authentication.

In this chapter, we will cover the following topics:

- What is a package?
- Recommended packages
- Working with the Auth package for user authentication
- What is Composer and how to use it?
- An introduction to building your own package

What is a package?

As developers, we often have code that we use on multiple projects. It could be as simple as string manipulation, but we would be lost without the code we are familiar with.

This is where packages come in useful. They give a great way of organizing, re-using, and sharing your code. Packages can contain all sorts of code, such as models, configurations, and even third-party libraries.

Due to the class structure of FuelPHP and other packages, it's possible to extend other packages and to the FuelPHP core. This is all possible without changing the core files—making upgrades easier and more straightforward.

Although there is a lot that packages can do, there are some things that packages can't do. For example, they can't map to a URL; this is the role of the application or project code. If there is a functionality that you'll re-use in multiple projects and which also requires URL

access, it is recommended to use modules. These will be introduced in the next chapter.

Running through some recommended packages is a great way to demonstrate what is possible when using packages.

Recommended packages

Although your requirements may differ, in this section we'll introduce a few packages that often come in useful.

OAuth

The internet nowadays is filled with big players like Facebook and Twitter, each one has a different structure and use; but they all support a version of **OAuth** for user authentication. This means that many projects may require the use of OAuth for user authentication and single sign-ins.

One highly-recommended package is a port from the Kohana PHP framework (http://kohanaframework.org). It can be found at https://github.com/fuel-packages/fuel-oauth.

This package handles the authentication with third-party services such as:

- Dropbox
- Flickr
- LinkedIn
- Tumblr
- Twitter
- Vimeo

When using this package, you'll need to copy the configuration file to your application configuration directory. This will allow you to add your consumer key and secret for the third party systems.

OAuth2

You would have noticed that Facebook was missing from the available interactions for the OAuth package. This was mainly due to

Facebook using the **OAuth2** standard (http://oauth.net/2). Although it does a similar job to the first package, you may find the OAuth2 package useful. This package can be found at https://github.com/fuel-packages/fuel-oauth2 and supports the following third-party services:

- Facebook
- Foursquare
- GitHub
- Google
- PayPal
- Instagram
- Soundcloud
- Windows Live
- YouTube

As with the previous package, you will need to create the controller and application code to make use of the authentication data from OAuth.

Mandrill

Sometimes, we need to send e-mails. Although it is certainly easier to use the default `mail()` PHP function, this method is not always reliable. Sometimes, the `mail()` function will silently fail. This is where third-party systems come in handy. There are several large e-mail services, such as **Campaign Monitor** and **MailChimp**. These tend to be e-mail campaign and mailing lists. Luckily for us, MailChimp have opened their infrastructure in the form of **Mandrill** (http://mandrill.com).

The Fuel Mandrill package is an unofficial one and can be obtained from https://github.com/Izikd/fuel-mandrill. It is a package wrapper for the official Mandrill API library, so you can be confident that it will work fine.

Sprockets

Frontend development and HTML is starting to become more structured with compilers such as LESS, Sass, and Compass. Usually, we need to compile the assets for serving on the Web with external libraries or tools. The Sprockets package is inspired by the Ruby on Rails asset pipeline (http://guides.rubyonrails.org/asset_pipeline.html). It handles this compiling for using ports of Sass, LESS, and CoffeeScript compilers. More information and installation instructions are available at https://github.com/vesselinv/fuel-sprockets.

Using the Auth package for user authentication

Although we touched upon the Auth package in Chapter 4, *Demo Application*, let's examine the package in detail.

The `Auth` package is not part of the FuelPHP core, it provides a standardized interface for the user authentication in FuelPHP. This allows developers to write their own drivers and easily integrate a new driver to work with the old code, but still keep the methods consistent.

The `Auth` package includes three basic drivers: login, group, and **ACL** (**Access Control List**). It is worth noting that login drivers can handle multiple login drivers at the same time. For example, users could log in using a username and password pair, or authenticate via a third-party system, such as Twitter.

Three of the included drivers are **SimpleAuth**, **ORMAuth**, and **OPAuth**.

SimpleAuth

This is a simple authentication system that uses a database table to store user information. It stores information regarding groups, roles, and ACLs in a configuration file. It was used as part of the administration system. As we saw in Chapter 4, *Demo Application*, SimpleAuth includes the migrations necessary to create the database tables that store the user information. Simply configuring your application to use SimpleAuth and then running the following command will create the tables:

```
oil r migrate --package=auth
```

More information on configuring SimpleAuth, along with usage examples can be found at the following link:

http://fuelphp.com/docs/packages/auth/simpleauth/intro.html

ORMAuth

This is similar to SimpleAuth in many aspects except that it stores all the configuration in the database rather than in a configuration file.

The user controls and roles can be more fine grained and assigned to individual users. Unlike SimpleAuth, the user attributes are stored in a metadata table rather than a serialized array in the user table. Another nice feature is that a history of user login times is kept.

To enable the ORMAuth package, you will need to add `ORMAuth` to the `always load` code section in the `app/config/config.php` file.

More information can be found at the following link:

http://fuelphp.com/docs/packages/auth/ormauth/usage.html

OPAuth

This is the most complex of the three provided drivers. It supports authentication against OAuth and OpenID providers. It also supports single sign-in; so, when a user is logged into a third-party site, such as Twitter, OPAuth will be able to detect the session and log in the user transparently.

More information on the OPAuth driver can be obtained at the following link:

http://fuelphp.com/docs/packages/auth/opauth/intro.html

As you saw in the scaffolding of the journal administration system, the authentication methods will need to be implemented in our

application. The Auth methods are well documented and named as follows:

- `Auth::check()`: This method checks if the user is authenticated and returns a Boolean true or false
- `Auth::remember_me()`: This method creates a "remember me" cookie
- `Auth::dont_remember_me()`: This method deletes the "remember me" cookie
- `Auth::logout()`: This method logs the user out
- `Auth::create_user( array())`: This method registers a user

Every web application is different, but at least the FuelPHP Auth package gives a good starting point for user authentication. It's usually good enough for the majority of uses.

Composer

There are currently a lot of ways to manage code and install third-party functionality. The Ruby world has the Gem packaging system. As mentioned in Chapter 1 *, What is FuelPHP?*, the framework is adopting the PHP coding and interoperability standards. Part of this is the ability to use code from other frameworks without rewriting them to a FuelPHP package.

During the life span of a project, the packages may change with new functionality and security fixes. Like Bundler for Ruby on Rails, PHP has a dependency manager called Composer.

Composer allows you to declare which versions of libraries to install in your project, and it will install them for you. This is great when developing and testing, as you know exactly what code is installed. It also allows you to source control any changes to those libraries.

Although it is still early days for FuelPHP packages and Composer, there are a few packages available at the following link:

https://packagist.org/search/?q=fuel-

To add more dependencies to your project, simply change the `composer.json` file found at `~/Sites/journal/composer.json`.

Once you have found a package you would like to use with Composer, it is a case of adding something like the following code in your `composer.json` file:

```
"require" : {
    "monolog/monolog": "1.2.*"
    }
```

This will ensure that you have the latest point release of the Monolog 1.2 package.

An introduction to building your own package

As you have seen so far, packages are very useful, besides they can also be simple to create. We will run through some of the basics in this section while creating a text manipulation package.

Setting up a repository

The first thing to do is set up a repository. Like in the previous chapter, we will use GitHub.

We will be creating a package called **Journal String**; this will have a repository name of `journal-string`. Ordinarily, it is recommend to use something like Fuel String, but since this is a simple example, it's not necessary to include Fuel within the title of the package.

We will be using the new repository as a submodule within the journal project, so make sure you make a note of the repository address, which will be something like `git@github.com:digitales/journal-string.git`.

Working the package as a submodule

We need to add this new submodule into our project, so it's time to load a console/terminal window. In the terminal, navigate to the top level of the journal project and then add the submodule. Make sure that it is cloned to the packages directory. In this example, we are checking out the submodule into a directory called string, rather than `journal-string`; this is to save typing and make it easier to autoload.

```
$ cd ~/Sites/journal
$ git submodule add
git@github.com:digitales/journal-text.git
```

```
fuel/packages/string
$ cd fuel/packages/string
```

The last command takes us into the `string` package directory. Git submodules act as a completely separate repository enclosed within the main project repository. This means that any changes made to the `journal-text` package will need to be committed to its own repository, and then the main repository will need to be updated.

Structuring your package

Since you will be sharing the package with your team or yourself or with the community, in the future, the structure of the directories and files is important. It will help you with a quick familiarization when working with the package. The recommended directory structure is given as follows:

```
/packages
    /package
        /bootstrap.php
        /classes
            /class1.php
            /second-class.php
        /config
            /packageconfig.php
```

Every package should include a `bootstrap.php` file in the top directory of the package. This file can be used to add a namespace for the package and to add the package classes for better performance and auto-loading purposes. If you want to override the core classes, you will need to add the package namespace to the core namespace, for example:

```
Autoloader::add_core_namespace( 'String', true );
```

Let's create our package structure, as shown in the following hierarchy:

```
packages/
    /journal/
        /classes/
            string.php
        /config/
            string.php
        bootstrap.php
        readme.md
```

In this example, we will create some basic two-way encryption for a string of text and give it the namespace `String`. We will have a configuration file to allow us to salt the encrypted string. So, let's first create the example configuration file. Load the `string.php` file located at `/journal/config/`, and add the following code snippet:

```
<?php
/**
 * NOTICE:
 *
 * If you need to make modifications to the
default configuration, copy
 * this file to your app/config folder, and make
them in there.
 *
 * This will allow you to upgrade without losing
your custom config.
 */
return array(
    'active' => Fuel::$env,
    'development' => array(
        'salt' => 'put_your_salt_here',
    ),
    'production' => array(
        'salt' => 'put_your_salt_here',
    ),
);
```

In the `string.php` file, there are a couple of example environments. In the configuration file, we are also dynamically setting the environment with `Fuel::$env`. This will be used within the main string class to load the configuration for the correct environment. Then, the

correct values will be assigned to a static class variable called `$_config`.

In the `encode` and `decode` functions, we are making use of the FuelPHP **Crypt** functionality. We are also including some uppercase and lowercase string manipulation functions for demonstration purposes later. Without further adieu, an example of the string manipulation class is given as follows:

```
<?php
/**
 * String manipulation Package
 *
 * This is a simple set of methods for string
manipulation
 *
 * @license    MIT License
 */
namespace String;
```

This class will include a `StringExeption` exception that extends the FuelPHP `Exception` class, allowing us to customize the exceptions when we need to:

```
class StringException extends \Exception {}

class String {
    protected static $config;

    /**
     * @var  object  PHPSecLib hash object
     */
    protected static $hasher;
```

The following function is used to get the `string.php` file from within the application configuration directory:

```
    protected static function get_config()
    {
        if ( !static::$config ):
            $config = \Config::load('string',
```

```
true);
            static::$config = $config[
$config['active'] ];
        endif;
        return static::$config;
    }


    /**
     * Encode a string using the core encode
function
     *
     *  This will retrieve the salt and use that
for the encoding.
     *
     *  @param string $string The string to be
encoded
     *  @param null | string $salt If the salt is
null, the config will be used instead.
     *  @return string
     */
    public static function encode( $string, $salt
= null )
    {
        if ( ! $salt) {
             $config = self::get_config();
    $salt = $config['salt'];
        }
        return \Crypt::encode( $string, $salt );
    }
```

The `encode()` function will encode the string and return the encrypted string. Next, we will decode the string in the `decode()` function:

```
    /**
     * Decode a string using the core decode
function
     *
     *  This will retrieve the salt and use that
for the decoding
     *
     *  @param string $string The string to be
decoded
     *  @param null | string $salt If the salt is
null, the config will be used instead.
```

```
     *  @return string
     */
    public static function decode( $string, $salt
= null )
    {
        if ( ! $salt) {
            $config = self::get_config();
            $salt = $config['salt'];
        }
        return \Crypt::decode ( $string, $salt );
    }
```

Now, let's introduce a quick **password hashing** method. It is worth noting that in PHP 5.5 we can use the new password hasher:

```
    /**
     * Default password hash method
     *
     * @param   string
     * @return  string
     */
    public static function
hash_password($password)
    {
        $config = self::get_config();
            $salt = $config['salt'];
        return base64_encode(self::hasher()-
>pbkdf2($password, $config['salt'], 10000, 32));
    }



    /**
     * Returns the hash object and creates it if
necessary
     *
     * @return  PHPSecLib\Crypt_Hash
     */
    public static function hasher()
    {
        if ( !static::$hasher ):
            if ( !
class_exists('PHPSecLib\\Crypt_Hash', false))
            {
                import('phpseclib/Crypt/Hash',
```

```
'vendor');
            }
            $hasher = new \PHPSecLib\Crypt_Hash();
            return $hasher;
        endif;

        return static::$hasher;
    }
```

In the next few functions, we will perform some straightforward text string manipulations. First, let's change the string to be all lowercase:

```
    /**
     * Convert the string to lowercase.
     *
     * @param   string  $str       required
     * @param   string  $encoding  default UTF-8
     * @return  string
     */
    public static function lower($str, $encoding =
null)
    {
        $encoding = \Fuel::$encoding;


        if ( function_exists('mb_strtolower') ){
            return mb_strtolower($str, $encoding);
        } else {
            return strtolower($str);
        }


    }
```

Now, let's convert the string to uppercase:

```
    /**
     * Covert the string to uppercase.
     *
     * @param   string  $str       required
     * @param   string  $encoding  default UTF-8
     * @return  string
     */
```

```
    public static function upper($str, $encoding =
null)
    {
        $encoding or $encoding = \Fuel::$encoding;

        if ( function_exists('mb_strtoupper') {
            return mb_strtoupper($str, $encoding);
        } else {
            return strtoupper($str);
        }


    }
```

Now, let's make the first character lowercase:

```
    /**
     * lcfirst
     *
     * Does not strtoupper first
     *
     * @param   string  $str       required
     * @param   string  $encoding  default UTF-8
     * @return  string
     */
    public static function lcfirst($str, $encoding
= null)
    {
        $encoding or $encoding = \Fuel::$encoding;

        if(function_exists('mb_strtolower')){
            return mb_strtolower(mb_substr($str,
0, 1, $encoding), $encoding).
                mb_substr($str, 1, mb_strlen($str,
$encoding), $encoding);
        }else{
            return lcfirst($str);
        }

    }
```

Now, let's make the first character uppercase:

```
    /**
     * ucfirst
```

```
     *
     * Does not strtolower first
     *
     * @param    string $str       required
     * @param    string $encoding  default UTF-8
     * @return   string
     */
    public static function ucfirst($str, $encoding
= null)
    {
        $encoding or $encoding = \Fuel::$encoding;

        if(function_exists('mb_strtoupper')_{
            return mb_strtoupper(mb_substr($str,
0, 1, $encoding), $encoding).
                mb_substr($str, 1, mb_strlen($str,
$encoding), $encoding);
        }else{
            return ucfirst($str);
        }

    }
```

Now, let's make the first character of each word uppercase:

```
    /**
     * ucwords
     *
     * First strtolower then ucwords
     *
     * ucwords normally doesn't strtolower first
     * but MB_CASE_TITLE does, so ucwords now too
     *
     * @param   string   $str       required
     * @param   string   $encoding  default UTF-8
     * @return  string
     */
    public static function ucwords($str, $encoding
= null)
    {
        $encoding or $encoding = \Fuel::$encoding;

        if ( function_exists('mb_convert_case') ){
            return mb_convert_case($str,
MB_CASE_TITLE, $encoding);
        } else {
```

```
                return ucwords(strtolower($str));
            }

        }
    }
```

Now that we have the basic functionality sorted, the application code will need to be capable of accessing it. For this, we can use the `bootstrap.php` file in the top level of the journal package directory.

Load the `bootstrap.php` file located at `packages/journal-string/`:

```
<?php
Autoloader::add_core_namespace('String');

Autoloader::add_classes(array(
    'String\\String' =>
__DIR__.'/classes/string.php',
    'String\\StringException' =>
__DIR__.'/classes/string.php'
));
```

Then, we will be able to call the functionality using something like the following:

```
String::decode( $the_string );
```

Configuring your package

The first thing to do when using the package will be to create a project-specific version of the package configuration. To do this, run the following in your terminal:

```
$ cp ~/Sites/journal/fuel/packages/journal-
string/config/string.php
~/Sites/journal/fuel/app/config/string.php
```

We need to add some custom `salt` text strings which will be used as keys in the newly copied `string.php` configuration:

```
return array(
    'active' => Fuel::$env,
    'development' => array(
        'salt' => '(my awesome salt)',
    ),
    'production' => array(
        'salt' => '(my awesome salt)',
    ),
);
```

Using your package

Now that you have configured the package and created the string functions, it's time to demonstrate using the new package. First, let's add the `String` package to our `config.php` applications file:

As we added a core namespace `String`, we can call our string functions using the following method:

```
$encoded_string = String::encode( 'something to
encode');
$decoded_string = String::decode();
echo $decoded_string;
```

You can test the functionality in your controllers and then display the results in the views. Creating packages is a simple process and one that you should get familiar with. Once you have created your package, you may want to share it with others.

Letting people know about your package

So, you've created your package, it's time to publish it. First, it is a good idea to check that the functions are all commented and you have documented how to use the package in the `Readme.md` file (or a `Readme.txt` file). If you are using GitHub for your code, they provide a quick way of creating a webpage to publicize your package or project. When creating a page with GitHub, they will use the Readme file as a starting point and then let you customize any of the information you may like to include about your package. More information can be seen at https://help.github.com/categories/20/articles.

Once you are happy that the code is ready to be shared, send a tweet to the FuelPHP Twitter account (https://twitter.com/FuelPHP). They will often "retweet" your message to their followers. In addition to this, you can share your package link on the FuelPHP forums at http://fuelphp.com/forums/categories/codeshare.

Summary

In this chapter, we have covered some of the package basics, along with examples of useful packages that can make our development life easier. With a collection of dependable packages, we can concentrate on the fun job of creating the application and delivering what the client wants. We have created a package, configured it, and demonstrated its use.

In the next chapter we will cover some of the more advanced topics, including functionality portability, unit testing, and profiling within FuelPHP.

Chapter 6. Advanced Topics

In the previous chapters, we have run through the basics of creating a project in FuelPHP. We've touched upon how useful packages can be, but they can't be matched directly to a URL. In this chapter, we will introduce modules, which, unlike packages, can be accessed directly from a URL. We will also run through some of the more advanced topics related to FuelPHP.

The topics covered in this chapter are as follows:

- Modules—what they are and how to use them
- Tasks
- Routing
- Unit testing
- Profiling

What modules are and how to use them

In the previous chapter, we touched upon the fact that packages don't relate directly to a URL; they need controllers and views in the application to do this. Modules, on the other hand, are a group of MVC elements that can act independently from the project application. They allow for the encapsulation and reusability of your code, so that you can share a module between projects without needing to write an application code to fully utilize the functionality.

Modules are expected to reside within the modules folder within the application. If you would prefer to store them elsewhere, this path and folder can be changed in the application configuration. It's recommended to use modules on larger projects that consist of a large code base as they can help keep the code in order.

Modules can be used independently and they don't need access to the global application code. A route can be created to allow for a module to be directly accessed via a URL. Since modules can include views and controllers, they can be thought of as mini applications in their own right.

By default, modules are expected to reside in the modules folder within the application; for example, `[rootOfProject]/fuel/app/modules`. This can be changed in your application `config.php` file—it appears in the section named `module_paths`. If you need to change the path to the modules, you will need to ensure that the end of the path includes the DS global variable. For example:

```
'module_paths' => array(
    APPPATH.'journal_modules'.DS, // path to
application modules
    APPPATH.'..'.DS.'globalmods'.DS, // path to
our global modules
),
```

Namespaces

Historically, naming classes and functions is a difficult job. We often prefer class names to have meaning and indicate what the class does; this can lead to conflicts between our code and that of a third-party code that we may also be using on the project. In PHP Version 5.3, we have been able to (optionally) use namespaces to avoid these conflicts, in FuelPHP all modules are required to have their own namespaces to avoid these kinds of conflicts. The namespaces must be named identically to the folder name for the module. For example, a module in the `Examplemodule` folder will have a controller like the following one:

```
<?php
/**
 * Module controller in the Examplemodule module
 */
 namespace Examplemodule;
```

```
class Controller_Example
{
    // Add your code here
}
```

In addition to the namespace and controller names, the module folder name also influences the name in the URL if you want to route to the controllers in the module. So, choosing the right name is important; but even without the perfect name, we can still change the URL routes from within the main application.

The module folder structure

Speaking of the folder name of the module, we should treat the module as a self-contained application, complete with an expected folder structure. The following is the expected folder structure for any module:

```
/classes
    /controller
    /model
    /view
/config
/lang
/tasks
/views
```

As you will notice, the folder structure is similar to the full application folder structure.

Using the module from the main application

Sometimes, we may want to use some of the functions in the modules from the main application. In these cases, we need to

autoload the class before we can reference the module in the code. This can be done in the following two ways:

- The first is to autoload the module in the application `config.php` file, like we did with the example packages

  In the `'always_load'` array in the `config.php` file, you will notice a section for modules:

```
'always_load' => array(
      'modules' => array('examplemodule'),
```

- An alternative way of loading the module is to autoload it only when we need to, in the classes that require the module functionality.

  This can be done using the following code:

```
Module::load('examplemodule');
```

Once you have loaded the module, you can call the functions from the module in the following way:

```
\Examplemodule\Exampleclass::examplemethod(
'params' );
```

As modules tend to be self-contained applications, each has a specific use and they are not as widely open sourced as packages are.

Tasks

Sometimes, we will want background processes, periodic tasks, or maintenance tasks to occur. This is where FuelPHP tasks come in handy. They can be run via the command-line tool, or set up as a periodic task on Windows, or **cron** job on Mac OSX and *nix. They can call upon modules and other classes just like controllers can.

Tasks should be placed in the `fuel/app/tasks` folder and, by default, only a `run()` method needs to be defined within the class. If you need other methods, these can be added in the usual way for PHP classes.

The tasks are called using the Oil `refine` command. FuelPHP comes with an example task called `robots` and will be included in the `fuel/app/tasks` folder.

To call the main method of the robot task, you can run the following command:

```
$ php oil refine robot
```

The `run()` method in the `robot` task has a variable defined, allowing you to pass a string via the `oil refine` command. Typing the following command line will change the message in the task:

```
$ php oil refine robots "Kill all mice"
```

If you look at the `robots.php` file, you will notice a second method called `protect()`—this can be called using the following code:

```
$ php oil r robots:protect
```

Tasks in FuelPHP are very useful for periodic actions and are very similar to controllers, making them straightforward to write. They also have the advantage of having the same access to the core FuelPHP functionality.

Routing

As with other frameworks, FuelPHP has fairly extensive routing capabilities. In this section, we will run through the basics.

Firstly, there are a couple of reserved routes; they are: `_root_` and `_404_`. The `_root_` key is used when there is no URL specified; for example, the home page or root page. The second (`_404_`) is for when the requested content controller or view can't be found.

The routes exist in the `config` folder of the application in a file called `routes.php`. Let's load the `routes.php` file from the following path comprising the following code:

```
[rootOfProject]/fuel/app/config/routes.php
```

```
<?php
return array(
    '_root_'  => 'welcome/index',  // The default
route
    '_404_'   => 'welcome/404',    // The main 404
route
);
```

As you can see from the `routes` configuration file, the routes are stored as an array. The key on the left is matched to the URL and then the items on the right are executed by FuelPHP. This is fairly straightforward, but can lend itself to complex URL and keyword matching.

The simplest routes are the ones that match a URL string directly to a controller and an action. These can look as follows:

```
return array(
    'about'  => 'welcome/about',  // The action
method in the welcome controller
    'contact'   => 'about/contact',    // The
```

```
    contact method in the about controller
    );
```

Our applications would tend to be fairly dynamic in nature, so specifying all the possible routes in the application would be a tedious job. This is where more advanced routing comes in handy. For this, we can use keywords and basic regular expressions to match strings in the URL and translate them to the controller method. These expressions make use of keywords preceded by a colon as follows:

- `:any`: This keyword matches anything from that point on.
- `:segment`: This keyword matches a single segment in the URL. The segment can be anything. This can be useful for language strings in the URL.
- `:num`: This keyword matches numeric values in the URL.
- `:alpha`: This keyword matches any alpha characters.
- `:alnum`: This keyword matches any alphanumeric characters.

Using the following code, the router will match *any* journal entry and then send the entry name to the `entry` method in the `journal` controller:

```
'journal/(:any)' => 'journal/entry/$1',
```

The code in the following example will allow any preceding segment to be used for URLs like `/en/contact` and will send the language flag as a variable to the `contact` method in the site controller. The final URL would be something like `/site/contact/en`:

```
'(:segment)/contact' => 'site/contact/$1',
```

As developers, we often try to make our code more readable by choosing clear and descriptive variable names. The same can be done with more advanced routing in FuelPHP, as it allows you to use named parameters in the routes. These named segments can then be accessed from within your methods or actions. For example:

```
return array(
    'journal/:year/:month/:day/:id' =>
'journal/entry',
);
```

In this route, `/journal/2013/11/5/name` would be routed to the entry method within the journal controller. In the entry method, we can get the named segments in the following manner:

```
$this->param('year');
$this->param('month');
$this->param('day');
$this->param('id');
```

FuelPHP uses regex for the named segments to work within the route. Each segment counts as a back reference, for example, the `$1` and `$2` regex placeholders—we often use these when making use of regular expressions in PHP. Back references are a regex term and more information can be found at the following website:

http://www.regular-expressions.info/brackets.html#usebackrefinregex

In a route of `:name/(\d{2}`, the digit (`d{2}`)would be found using the variable `$2` and the variable `$1` would return the value of the `:name` segment.

We mentioned the RESTful controller template in the previous chapters. These can be used in conjunction with verb-based routing to direct requests to the correct methods in the RESTful controllers. This allows the route to a certain URL to be routed through to different methods and controllers in order to fit with the functionality.

For example:

- A `POST` request to `/journal` could be routed to the `create` method in the journal controller
- A `GET` request to `/journal` could be routed to the `index` method in the journal controller

These requests all follow the recommended use of the HTTP verbs to perform actions in the application and the route in the `routes.php` file would look something like the following:

```
return array(
    'journal' => array(
        array('GET', new Route('journal/index')),
        array('POST', new
Route('journal/create')),
    )
);
```

We could do something similar for the `PUT` and `DELETE` verbs and use regex and named parameters to make it easier to get the relevant information from the URL.

If you are working with user profile information and data, we would look to use HTTPS or secure connections; again, routing in FuelPHP supports this. The following example would only load the route if the request is sent via HTTPS, rather than just HTTP:

```
return array(
    'user/(:any)' => array(
            array( 'GET', new Route(
'user/view/$1' ), true )
        ),
);
```

The third parameter ensures that the named route is only used when HTTPS is used.

During development, we often rearrange the structure of the application to reflect the changing functionality. One feature of routes in FuelPHP, that aim to make routing easier, is called named routes and reverse routing. With this feature, instead of editing all of our views, we can simply change the named route in the main `routes.php` file. For this to work, we need to use the name of the route in our views. In the following example, we change `'admin/app/dashboard'` to `'admin/dashboard'`:

```
return array(
    'admin/app/dashboard' =>
array('admin/dashboard', 'name' =>
'admin_dashboard'),
);
```

In our views that need to link to the dashboard, we would use the following `anchor` code:

```
echo Html::anchor(Router::get('admin_dashboard'),
'Dashboard');
```

Note

This only works for application code and will not work for module routes.

Unit testing

No modern framework would be complete without the ability to test the application code and functionality. FuelPHP has been built with this in mind and, thus, includes tests and test cases based on the PHPUnit testing framework.

So what is unit testing?

Unit tests are automated tests written to check whether or not the units of functionality (methods and functions) are working as expected. The tests typically test whether or not, for a given input, the output is correct, treating the functions as a black box to ensure that the internal logic works.

Since unit testing is automated, it's easy to ensure that the recent code changes don't break other functionality. It also allows for the use of a continuous integration server such as Jenkins (http://jenkins-ci.org). A continuous integration server will automatically deploy your code for you once the code passes the unit testing, allowing you to concentrate on the actual code.

PHPUnit

There are several unit testing tools available in PHP, but the de facto standard is PHPUnit by Sebastian Bergmann. This is the testing framework supported by FuelPHP and is run using the Oil command-line tool. But, before using it, you will need to ensure that PHPUnit is installed. For the most up-to-date installation instructions, I'd recommend browsing the official documentation available at the following link:

http://www.phpunit.de/manual/current/en/installation.html

Running unit tests

Unit tests are run using the FuelPHP Oil command-line tool and can be run using `php oil test`:

```
$ php oil test
$ Tests Running...This may take a few moments.
$ PHPUnit 3.7.21 by Sebastian Bergmann.
$
$ Configuration read from
/home/user/sites/journal/fuel/core/phpunit.xml

$
.................................................
63 / 251 ( 25%)
$
.................................................
126 / 251 ( 50%)
$
.................................................
189 / 251 ( 75%)
$
.................................................
$ Time: 6 seconds, Memory: 22.25Mb
$
$ OK (251 tests, 206 assertions)
```

Creating unit tests

The tests are located in the application in the `fuel/app/tests` folder and will read any tests in its subfolders. The tests files should follow a similar structure to the classes they are testing. So, if you were testing the category model (`/fuel/app/classes/model/category.php`) it would have a test file located at `fuel/app/tests/model/category.php`. The test cases should extend the `TestCase` class, which is an extension of the `PHPUnit_Framework_TestCase` class. This means that you will be able to use the usual PHPUnit assertions and methods in your tests.

The name of the class should be prefixed with `Test_`. Similarly, the category test should be named `Test_Model_Category`. The class should look something like the following:

```
<?php
class Test_Model_Category extends TestCase
{
    public function test_category()
    {
        // Add your code here
    }
}
```

The official documentation lists many assertions and recommended ways of writing unit tests. The documentation can be found at the following link:

http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html

Grouping unit tests

As your application grows, sometimes, running all the unit tests can be time consuming; as such, we can group tests together and then only test certain groups. This is done with the `--group=` command at the end of the usual `test` command. So, running the `User` group of tests can be done using the following code:

```
$ php oil test --group=User
```

The grouping is done in a `docbloc` comment in each testing class and each test case can be assigned to multiple groups. The following code snippet would assign the category model `Test` to `Blog` and `App` groups:

```
<?php
/**
 * @group Blog
```

```
 * @group App
 */
 class Test_Model_Category extends TestCase
{
    public function test_category()
    {
       // Add your code here
    }
}
```

Configuration and module testing

FuelPHP uses configuration in the `phpunit.xml` file, which is included in the `fuel/core/` folder. To customize the configuration, we need to copy this file to our application and then make the changes there. FuelPHP will load the application `phpunit.xml` in favor of the core version.

As mentioned earlier, modules are applications in their own right—like unit testing. The unit tests should be included in a `test` folder within the top level of each module. For FuelPHP to run the tests in modules, it needs to know that they exist. This is done by including the modules folder in the `phpunit.xml` file. Once you have made a copy of the core FuelPHP `phpunit.xml` file, you can add the following code snippet:

```
<testsuite name="modules">
    <directory
suffix=".php">../app/modules/*/tests</directory>
</testsuite>
```

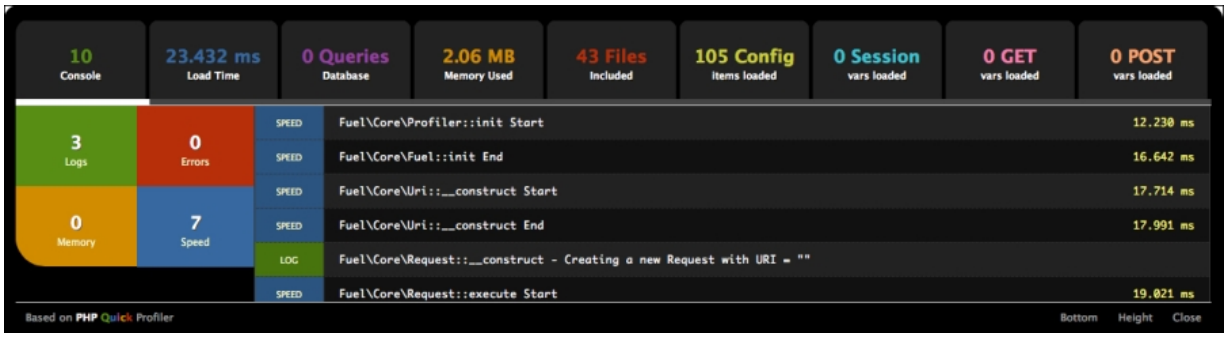
Now that you know a little about unit testing in FuelPHP, let's cover another aspect of development—the ability to profile your application.

Profiling

FuelPHP includes a **profiler** which is based on **PHP Quick Profiler**. This allows you to profile and debug your code, all without needing to write extra functions within the application. The profiler can be switched on and off via the application `config.php` file. To enable the profiler, simply change the `'profiling'` variable to `true`, and set it to `false` to disable the profiler.

The profiler also includes a database-profiling tool, but, due to the resources required, it is disabled by default. The database profiler will need to be enabled corresponding to the environment used so that the development environment can have it enabled without affecting other environments. It can be enabled in the environment's `db.php` file using the `true` value, as shown in the following code:

```
'profiling' => true,
```



The profiler has a tabbed interface and consists of the following tabs:

- **Console**: This is the default tab giving information about errors, log entries, and memory usage, along with execution timings
- **Load Time**: This tab shows the request load time
- **Database**: This tab shows the execution time and number of database queries executed
- **Memory Used**: This is the peak memory used by the page load
- **Included**: This tab shows the list of files loaded with their filenames and sizes

- **items loaded**: This tab show the final configuration variables at the end of the page load
- **vars loaded**: This tab shows the contents of the session at the end of the page load
- **GET**: This tab shows the `$_GET` array contents
- **POST**: This tab shows the `$_POST` array contents

The profiler provides a lot of information that can help you in optimizing your application.

Summary

In this chapter, we have introduced modules and how we can use them to quickly build our applications. We have touched upon unit testing and how to enable the profiler to optimize our code.

We have configured some basic routes and detailed where any changes will need to be placed. Named routes are a very powerful tool that can be used to counter changing project and client requirements. They allow us to reduce the number of changes that need to be made.

Tasks are a great way of building background or period functionality in our application and are really easy to run with the FuelPHP Oil command-line utility.

In the next chapter, we will introduce some ways to publicize our applications, packages, or modules within the FuelPHP community.

Chapter 7. Welcome to the Community

As you have seen from the preceding chapters, FuelPHP provides a lot of features and is constantly evolving with new features and ways of working. It deals with time-consuming tasks, allowing us to concentrate on the more fun aspects of coding. Thanks to a large community of developers, we have a large selection of packages and modules to give us a head start on projects and laying the foundations for great applications and projects.

FuelPHP and GitHub

The main website is http://fuelphp.com, which is where the main documentation resides. As the framework is open source, the real hub of activity is the GitHub pages, which can be found at https://github.com/fuel/.

Getting help and support

Sometimes, we need to get a little extra help that isn't always covered in the documentation. For this, there are the main FuelPHP forums at http://fuelphp.com/forums/. These are usually a great place to start when you need a little extra help. The forums are split into several groups, such as General, Packages, and Development. It's a great place to publicize your new packages or simply share what you have created.

In addition to the forums, you will find an active **Internet Relay Chat** (**IRC**) channel, `#fuelphp`.

FuelPHP on social media

Social media makes it quicker to spread the message to the community and FuelPHP uses both Twitter and Facebook for this purpose.

Twitter

The core FuelPHP team is fairly active on Twitter and can be found at https://twitter.com/FuelPHP. This is a great way of publicizing applications you've built, as they will often retweet you to their followers.

Facebook

Although not as active as the FuelPHP forums or Twitter, the FuelPHP Facebook page at https://www.facebook.com/pages/FuelPHP/120778634657454 is monitored and all questions do get answered.

Parting words

I hope you have found this introductory book useful for getting into the world of FuelPHP. I always believe that the best way to learn a new language or framework is to use it for a project or an idea.

The FuelPHP Oil tool is a great way for quickly prototyping ideas to test or demonstrate. Think about the minimum viable product that can be built to test the idea. This will give you the momentum to get it started.

Don't forget to reach out if you want to know more or get stuck with anything in this book.

Index

A

- administration system
    - generating, Oil tool used / Generating an administration system using Oil
- Apache
    - about / Preparing the development environment, Apache
- authentication package, FuelPHP
    - about / The authentication package
- Auth package
    - about / Using the Auth package for user authentication
    - used, for user authentication / Using the Auth package for user authentication

B

- backwards compatibility, FuelPHP Version 2.0
    - about / Backwards compatibility
- base controllers / Base controller classes and model classes
- beanstalk
    - URL / Getting started
- Bitbucket

    - URL / Getting started
- Bitbucket.org
    - about / Committing your code
- bootstrap PHP file
    - about / The bootstrap PHP file

C

- Campaign Monitor / Mandrill
- Categories table
    - fields / Categories
- categories_entries table
    - fields / categories_entries
- coding standards, FuelPHP Version 2.0
    - about / Coding standards
- composer
    - about / Composer
- Composer package manager
    - about / Composer – the package manager
- Composer tool
    - about / Modular and extendable, Getting and installing FuelPHP with curl and Oil
- configuration, FuelPHP

    - about / Configuration
- controllers
    - about / Controllers, Controllers
    - template controller / Template
    - rest controller / Rest
    - hybrid controller / Hybrid
    - creating, Oil used / Controllers
- Cross-site request forgery (CSRF)
    - about / Security
- Cross-site scripting (XSS) attacks
    - avoiding / Security
- Crypt functionality / Structuring your package
- curl
    - used, for installing FuelPHP / Getting and installing FuelPHP with curl and Oil
- custom package
    - building / An introduction to building your own package
    - repository, setting up / Setting up a repository
    - working, as submodule / Working the package as a submodule
    - structuring / Structuring your package
    - configuring / Configuring your package
    - using / Using your package
    - publishing / Letting people know about your package

D

- database interactions, FuelPHP / Database interactions
- database migrations

    - creating / Migrations and Oil
    - running / Migrations and Oil
- database table
    - creating / Creating a database table
- Dependency Injection Container (DiC) design pattern / Global scope and moving to dependency injection
- development environment
    - preparing / Preparing the development environment
- documentation, FuelPHP
    - about / Documentation

E

- encode() function / Structuring your package
- Entries table
    - fields / Entries
- environment settings, FuelPHP
    - Server and Apache configuration / Server and Apache configuration
    - bootstrap PHP file / The bootstrap PHP file

F

- Facebook
    - URL / Facebook
- fileinfo() / PHP
- FuelPHP

    - history / A brief history of FuelPHP
    - features / Features of FuelPHP
    - database interactions / Database interactions
    - installing, curl used / Getting and installing FuelPHP with curl and Oil
    - installing, Oil used / Getting and installing FuelPHP with curl and Oil
    - project, setting up / Setting up your project
    - code, comitting / Committing your code
    - configurations / Configuration, Configuration
    - running, in production environment / Running in the production environment
    - migrations, performing / Performing migrations
    - environment settings / Environments, constants, and configuration
    - constants / Constants
    - MVC pattern / Models, views, and controllers
    - about / Getting started
    - package / What is a package?
    - modules / What modules are and how to use them
    - tasks / Tasks
    - routing / Routing
    - unit testing / Unit testing
    - profiling / Profiling
    - URL / FuelPHP and GitHub
    - on social media / FuelPHP on social media
    - on Twitter / Twitter
    - on Facebook / Facebook
- FuelPHP, features
    - HMVC / (H)MVC
    - modules / Modular and extendable
    - extensions / Modular and extendable
    - security / Security
    - Oil / Oil – the power of the command line
    - ORM / ORM
    - base controller / Base controller classes and model classes

    - model classes / Base controller classes and model classes
    - authentication package / The authentication package
    - template parsers / Template parsers
    - documentation / Documentation
- FuelPHP core
    - updating, submodules used / Using submodules to easily update the FuelPHP core and packages
- FuelPHP forums
    - URL / Getting help and support
- FuelPHP packages
    - updating, submodules used / Using submodules to easily update the FuelPHP core and packages
- FuelPHP repository
    - cloning / Clone from GitHub
- FuelPHP tools
    - demonstrating / Getting started
- FuelPHP Version 2.0
    - about / What to look forward to in Version 2.0
    - features / What to look forward to in Version 2.0
    - global scope / Global scope and moving to dependency injection
    - Dependency Injection Container (DiC) design pattern / Global scope and moving to dependency injection
    - coding standards / Coding standards
    - package management / Package management and modularization
    - modularization / Package management and modularization
    - backwards compatibility / Backwards compatibility

    - speed boosts / Speed boosts

G

- Git
    - about / Source control – introducing Git
    - URL, for info / Further information about Git
- Github
    - URL / FuelPHP and GitHub
- GitHub
    - URL / Getting started
- git submodule command
    - about / Using submodules to easily update the FuelPHP core and packages

H

- HMVC
    - about / (H)MVC
- Homebrew
    - URL / Source control – introducing Git
- hybrid controller
    - about / Hybrid

I

- IIS
  - about / Preparing the development environment
- Internet Relay Chat (IRC) channel / Getting help and support

J

- Jenkins
  - URL / So what is unit testing?

K

- Kohana PHP framework
  - URL / OAuth

M

- MacPorts
  - URL / Source control – introducing Git
- MailChimp / Mandrill
- Mandrill
  - about / Mandrill
  - URL / Mandrill
- mbstring() / PHP
- mcrypt / PHP
- migrations
  - performing / Performing migrations
  - about / Migrations and Oil

- model classes / Base controller classes and model classes
- models
    - about / Models, Models
    - creating, Oil used / Models
- modularization, FuelPHP Version 2.0
    - about / Package management and modularization
- modules
    - about / What modules are and how to use them
    - namespaces / Namespaces
    - folder structure / The module folder structure
    - using, from main application / Using the module from the main application
- mod_rewrite module
    - about / Apache
- Multiton design pattern / Global scope and moving to dependency injection
- MVC pattern
    - about / Models, views, and controllers
    - Views / Views and ViewModels
    - ViewModels / Views and ViewModels
    - Models / Models
    - Controllers / Controllers
- MySQL
    - about / Database interactions

N

- Nginx
  - about / Preparing the development environment

O

- OAuth
  - about / OAuth
  - authentication, handling with third-party services / OAuth
- OAuth2
  - about / OAuth2
  - URL / OAuth2
  - supported third-party services / OAuth2
- Oil
  - about / Oil – the power of the command line
  - URL / Oil – the power of the command line
  - used, for installing FuelPHP / Getting and installing FuelPHP with curl and Oil
  - used, for creating models / Models
  - used, for creating controllers / Controllers
- Oil tool
  - about / Migrations and Oil
  - used, for generating administration system / Generating an administration system using Oil
- OPAuth
  - about / OPAuth
  - reference link / OPAuth
- ORM

    - about / ORM
    - data relations / ORM
    - URL / ORM
- ORMAuth
    - about / ORMAuth
    - reference link / ORMAuth

P

- package
    - about / What is a package?
    - recommended packages / Recommended packages
    - OAuth / OAuth
    - OAuth2 / OAuth2
    - Mandrill / Mandrill
    - Sprockets / Sprockets
- package management, FuelPHP Version 2.0
    - about / Package management and modularization
- Packagist
    - about / Composer – the package manager
- parser package, FuelPHP
    - about / Template parsers
- password hashing method / Structuring your package
- PHP
    - about / PHP
- PHP constants

    - about / Constants
    - APPPATH / Constants
    - COREPATH / Constants
    - DOCROOT / Constants
    - PKGPATH / Constants
    - VENDORPATH / Constants
- PHP Data Objects (PDO) drivers / Database interactions
- PHP Extension and Application Repository (PEAR)
    - about / Package management and modularization
- PHP Quick Profiler
    - about / Profiling
- PHPSecLib
    - about / Security

  / PHP
- PHPUnit
    - about / PHPUnit
- profiler
    - about / Profiling
    - tabbed interface / Profiling

R

- rest controller
    - about / Rest
- routing
    - about / Routing

    - _404_ / Routing
    - _root_ key / Routing
    - routes configuration file / Routing

S

- Server and Apache configuration
    - about / Server and Apache configuration
- SimpleAuth
    - about / SimpleAuth
    - reference link / SimpleAuth
- Singleton design pattern / Global scope and moving to dependency injection
- social media
    - about / FuelPHP on social media
- speed boosts, FuelPHP Version 2.0
    - about / Speed boosts
- Sprockets
    - about / Sprockets
- StringExeption exception / Structuring your package
- submodules
    - about / Using submodules to easily update the FuelPHP core and packages
    - used, for updating FuelPHP core / Using submodules to easily update the FuelPHP core and packages
    - used, for updating FuelPHP packages / Using submodules to easily update the FuelPHP core and packages

- Subversion
    - about / Source control – introducing Git

T

- tasks
    - about / Tasks
- template controller
    - about / Template
- top-level domain (TLD) / Getting started
- Twitter
    - URL / Twitter

U

- unit tests
    - about / So what is unit testing?
    - PHPUnit / PHPUnit
    - running / Running unit tests
    - creating / Creating unit tests
    - grouping / Grouping unit tests
    - configuration / Configuration and module testing
    - module testing / Configuration and module testing
- Users table
    - fields / Users

V

- ViewModel
    - about / Views and ViewModels
- views
    - about / Views
- Views
    - about / Views and ViewModels
- virtual hosts (vhosts) / Getting started