**FuelPHP Application Development Blueprints**

FuelPHP is an open source framework that uses some of the most advanced features of PHP to allow you to build projects as easily and quickly as possible. It is gaining popularity with developers as you can build powerful, complex, and yet efficient applications with very little code.

This book will provide you with the skills you need to successfully create, improve, and publish functional and maintainable FuelPHP applications. We will go through a variety of topics such as MVC, ORM, scaffolding, packages, modules, and much more by building applications of increasing complexity. At the beginning of each chapter, we will specify the application we want to build, and then we will progressively implement it by learning new FuelPHP's features along the way. By the end of the book, you will be familiar with FuelPHP and will be able to implement new projects all by yourself.

**Who this book is written for**

This book is for intermediary to seasoned web developers who want to learn how to use the FuelPHP framework and build complex projects using it. You should be familiar with PHP, HTML, CSS, and JavaScript, but no prior knowledge about MVC frameworks is required.

**What you will learn from this book**

- Use FuelPHP's ORM to your advantage to execute complex operations in the database using simple and object-oriented code
- Implement clean and maintainable HMVC web applications using controllers, views, models, and presenters
- Create modules and packages in order to implement reusable code bundles
- Implement tasks to execute repetitive actions or cron jobs
- Use the oil console to quickly test code and ideas
- Create a powerful API-driven application in order to allow users and external services to use your data
- Debug and profile your applications for further improvements
- Be introduced to the Novius OS Content Management System based on FuelPHP

**Supercharge your projects by designing and implementing web applications with FuelPHP**
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Building Your First FuelPHP Application"
- A synopsis of the book’s content
- More information on FuelPHP Application Development Blueprints

About the Author

Sébastien Drouyer is a computer science research engineer from France. He has a master's degree in computer science from the National Institute of Applied Sciences of Lyon, one of the most prestigious engineering schools in France. He has been developing web applications since 2005 and has won various contests and awards from GitHub, NASA, and Intel. He has also been a member of the Novius OS core team (an open source content management system based on the FuelPHP framework) and published many additional open source projects. He has trained several teams on FuelPHP and is a conference speaker on the subject.

First of all, I would like to thank the FuelPHP core team and its community for improving this wonderful framework every day.

If there are only a countable number of errors in this book, then it is due to Aravind Udayashankara, Kenji Suzuki, Sági-Kazár Márk, Ivan Đurdevac, Craig Hooghiem, and John Alder who all did excellent reviews.

I would like to salute the amazing Novius OS core team and I wish them the best in their future endeavors.

I would also like to thank the Packt Publishing team, especially Melita Lobo, for their patience and professionalism.

Last, but not least, I would like to thank my very supportive family and friends. A special thanks to my mother and father for making me the person I am today; I wouldn't be here without you two.
FuelPHP Application Development Blueprints

The main idea behind FuelPHP Application Development Blueprints is to teach you FuelPHP's basic and advanced features by building various projects of increasing levels of complexity. It is very result-oriented; at the beginning of the chapters, we specify the application we want to build, and then we progressively implement it by learning how to use new FuelPHP features along the way. Our approach will therefore be very practical; a lot of concepts will be explained using code examples, or by integrating them directly into our projects. Thus, it is important to highlight that there will be a lot of code and you should be comfortable with reading and understanding PHP and HTML. As we will use them from time to time, having some knowledge about server/system administration and some foundation in JavaScript, jQuery, and CSS will be an added advantage.

Though this book is for intermediary to advanced web developers, any prior knowledge of the FuelPHP framework, or any other PHP framework, is not required. In order to understand this book, you don't have to know common concepts such as MVC, HMVC, or ORM. We take into account this shortcoming some of you might have, and important notions will be explained. We won't explain all of those in the first chapter though, as we want this to be as painless as possible; we will instead approach them when they become necessary for the project completion.

The ultimate purpose of FuelPHP Application Development Blueprints is to give you the ability to build any project using FuelPHP. By the end of this book, you certainly won't know every little detail of the framework, but you will hopefully have the necessary toolbox required to implement complex and maintainable projects.

What This Book Covers

Chapter 1, Building Your First FuelPHP Application, covers the very basics of the FuelPHP framework; how to install it, how to configure it, how it is organized, and its main components. Along the way, we will generate our first FuelPHP application using the oil utility and tweak some files, in order to illustrate how things work.

Chapter 2, Building a To-do List Application, focuses on FuelPHP's ORM and debugging features. We will illustrate these features using a lot of examples, and then implement a small to-do list application. We will also use some JavaScript and jQuery to send AJAX requests.

Chapter 3, Building a Blog Application, will teach you how to generate and tweak an administration interface easily, how to create your own modules and tasks, how to manage paginations easily, and how to use the Auth and Email packages. We will create a blog application implementing all these features.
Chapter 4, Creating and Using Packages, will approach the FuelPHP package system. This is a rather short chapter; we will first try to protect our website from spam bots by installing an existing package, and then create our own original solution by creating a new package.

Chapter 5, Building Your Own RESTful API, covers more advanced subjects such as building a JSON API, using language agnostic template engines, allowing user subscriptions, and implementing unit tests. To illustrate this, we will create a responsive micro blogging application featuring a public API.

Chapter 6, Building a Website Using Novius OS, will quickly introduce you to Novius OS, a FuelPHP-based Content Management System. Using such a system can greatly speed up the implementation of complex projects.
Building Your First FuelPHP Application

Throughout the book, we will use the FuelPHP framework to build different types of projects. The objective of this chapter is to make you familiar with the basics of the framework and create your first project as quickly as possible. We won’t create anything exceptional in this chapter and there will be very little coding, but we will go through the whole process from installing FuelPHP to publishing your project on a production server. You will learn the necessary basics for the other projects as well.

By the end of the chapter, you should know the following:

- A common development process of a FuelPHP application
- How to install FuelPHP (the latest or a specific version)
- The FuelPHP file system hierarchy
- Two different ways to configure Apache to access your application
- How to configure FuelPHP to connect to a database
- The oil command line and how to use it for scaffolding your application
- How does an application respond to a URL requested by a visitor
- What are the FuelPHP templates
- How to publish your project to a host

Since this book is intended for intermediate developers, we will assume that you have already installed Apache and MySQL on your system. Some prior knowledge of Git and Composer is an added advantage as you might need it, but you should be fine in this book if you are not familiar with these tools. However, for advanced applications that need collaboration between several developers mastering them is highly recommended.
In this chapter, we will go from installing the FuelPHP framework to having a functional – though limited – web application. As our objective here is to introduce the framework and create a sample application as quickly as possible, we won't address important topics such as the ORM, which will be addressed in Chapter 2, Building a To-do List Application.

**About FuelPHP**

Dan Horrigan started the FuelPHP framework in late 2010, and was later joined by Phil Sturgeon, Jelmer Schreuder, Harro Verton, Frank de Jonge, Steve West and Márk Sági-Kazár. The first stable version was released on July 31st, 2011 and this book is based on FuelPHP 1.7.2, the latest stable version available as of writing this book. With over 300 contributors, its community is large and active.

The core team is currently working on the second version of FuelPHP; several alpha versions of it have already been released.

If you want to know more about the FuelPHP team and the framework philosophy, I recommend you to read the About FuelPHP section of the official website at:

http://fuelphp.com/about

You can read the latest news about the framework on its official blog at:

http://fuelphp.com/blogs

The official documentation can be found at: http://fuelphp.com/docs/

If you have any questions about FuelPHP or encounter any issues, you can search the official forum (http://fuelphp.com/forums/) and start a new discussion if you don't find any answer. In a general manner, the official website (http://fuelphp.com) is an excellent resource.
Development process of a FuelPHP application

The development process of a FuelPHP application generally contains the steps shown in the following image:

- **Install FuelPHP**: Since we are using this framework, this first step is quite obvious.
- **Config (configuration)**: At the beginning, you will generally need to specify how to connect to the database and which package you will use. Later on, you might also need to create and use your own configuration files to improve the maintainability of your application.
- **Scaffold**: The oil command line of FuelPHP allows you to easily generate code files ready to be used. This step is not necessary, but we will often use this functionality in this book because it really speeds up the implementation of your application.
- **Dev (development)**: This is where you, as a developer, step in. You customize the generated code to get exactly what you want. When you want to add new features (for instance a new model), you go back to the scaffolding step.
- **Tests**: Functional and unit testing are important if you want large applications to stay maintainable. When bugs are discovered, you go back to the development step in order to fix them. Unlike the other steps, we won't approach this subject in this chapter for the sake of its conciseness. It will be addressed in *Chapter 5, Building Your Own RESTful API*.
- **Prod (production)**: Having a project working locally is nice, but the final objective is generally to publish it online. We will give you some directions about this step at the end of this chapter, but we won't get too much into the details, given the diversity of available hosting services.
Just to be clear, this is a very general guideline, and of course the order of the steps is not rigid. For instance, developers using the test-driven development process could merge the fourth and fifth steps, or a preproduction step could be added. The development process should only depend on each developer and institution's standards.

Installing the environment

The FuelPHP framework needs the following three components:

- **Web server**: The most common solution is Apache
- **PHP interpreter**: The 5.3.3 version or greater
- **Database**: We will use MySQL

FuelPHP works on Unix-like and Windows operating systems, but the installation and configuration procedures of these components will depend on the operating system used. In the following sections we will provide some directions to get you started in case you are not used to installing your development environment. Please note that these are very generic guidelines, so you might need to search the web for complimentary information. There are countless resources on the topic.

Windows

A complete and very popular solution is to install WAMP. This will install Apache, MySQL, and PHP, in other words everything you need to get started. It can be accessed at [http://www.wampserver.com/en/](http://www.wampserver.com/en/).

Mac

PHP and Apache are generally installed on the latest version of the OS, so you just have to install MySQL. To do this, you are recommended to read the official documentation at [http://dev.mysql.com/doc/refman/5.1/en/macosx-installation.html](http://dev.mysql.com/doc/refman/5.1/en/macosx-installation.html).

A very convenient solution for those who have the least system administration skills is to install MAMP, the equivalent of WAMP, but for the Mac operating system. It can be downloaded from [http://www.mamp.info/en/downloads/](http://www.mamp.info/en/downloads/).
Ubuntu
As this is the most popular Linux distribution, we will limit our instructions to Ubuntu.

You can install a complete environment by executing the following command lines:

```
# Apache, MySQL, PHP
sudo apt-get install lamp-server

# PHPMyAdmin allows you to handle the administration of MySQL DB
sudo apt-get install phpmyadmin

# Curl is useful for doing web requests
sudo apt-get install curl libcurl3 libcurl3-dev php5-curl

# Enabling the rewrite module as it is needed by FuelPHP
sudo a2enmod rewrite

# Restarting Apache to apply the new configuration
sudo service apache2 restart
```

Recommended modules and extensions
The Apache mod_rewrite module and some additional PHP extensions are also recommended, but not required:

http://fuelphp.com/docs/requirements.html (can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Basic | Requirements)

Getting the FuelPHP framework
As this book is being written, there are four common ways to download FuelPHP:

- Downloading and unzipping the compressed package which can be found on the FuelPHP website.
- Executing the FuelPHP quick command-line installer.
• Downloading and installing FuelPHP using Composer.
• Cloning the FuelPHP GitHub repository, it is a little bit more complicated but allows you to select exactly the version (or even the commit) you want to install.

These approaches are very well-documented on the website installation instructions page at http://fuelphp.com/docs/installation/instructions.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Installation | Instructions)

Installing FuelPHP 1.7.2
FuelPHP is always evolving and will continue to evolve even after this book is published. As we used FuelPHP 1.7.2 in this book, you might want to install the same version in order to prevent any conflict. You can do this by either downloading the appropriate ZIP file, cloning the 1.7/master branch of the GitHub repository, or using Composer.

Downloading the appropriate ZIP file
This is the simplest solution. You should be able to download it by requesting the URL http://fuelphp.com/files/download/28.

Alternatively, you can access all the compressed packages of important FuelPHP releases at http://fuelphp.com/docs/installation/download.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Installation | Download)

Using Composer
First, if you didn't do it yet, you need to install Composer. You can find out how by reading the official website at https://getcomposer.org/.

The installation instructions for major operating systems are given in the Getting Started guide. Please note that you can install Composer either globally or locally.

From now on, we will generally assume that you have installed Composer globally. If Composer is installed locally into your working directory, our instructions will work if you replace composer by php composer.phar.
In order to specifically install FuelPHP 1.7, you can simply execute the following command line (replace \texttt{TARGET} by the directory in which you want to download FuelPHP):

\begin{verbatim}
composer create-project fuel/fuel:dev-1.7/master TARGET
\end{verbatim}

**Updating FuelPHP**

If you have downloaded FuelPHP by cloning the GitHub repository, or if you simply want to update FuelPHP and its dependencies, you have to enter the following command line at the location you installed your instance of FuelPHP:

\begin{verbatim}
php composer.phar update
\end{verbatim}

As you can see, Composer is locally installed in the FuelPHP root directory.

**Installation directory and apache configuration**

Now that you know how to install FuelPHP in a given directory, we will give you the two main ways you can integrate the framework in your environment.

**The simplest way**

Assuming you have activated the \texttt{mod_rewrite} Apache module, the simplest way is to install FuelPHP in the root folder of your web server (generally the \texttt{/var/www} directory on Linux systems). If you install FuelPHP in the \texttt{DIR} directory of the root folder (\texttt{/var/www/DIR}), you will be able to access your project at the following URL:

\begin{verbatim}
http://localhost/DIR/public/
\end{verbatim}

However, be warned that FuelPHP has not been implemented to support this, and if you publish your project this way in the production server, it will introduce security issues you will have to handle. In such cases, you are recommended to use the second way we will explain in the upcoming section, although, for instance if you plan to use a shared host to publish your project, you might not have the choice. A complete and up-to-date documentation about this issue can be found in the FuelPHP installation instruction page at http://fuelphp.com/docs/installation/instructions.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Installation | Instructions)
By setting up a virtual host

Another way is to create a virtual host to access your application. You will need a little bit more Apache and system administration skills, but the benefit is that it is more secure and you will be able to choose your working directory. You will need to change two files:

- Your Apache virtual host file(s) in order to link a virtual host to your application
- Your system host file in order to redirect the wanted URL to your virtual host

In both cases, the files' location will be dependent on your operating system and the server environment you are using; therefore, you will have to figure out their location yourself (if you are using a common configuration, you won't have any problem to finding instructions on your preferred search engine).

In the following example, we will set up your system to call your application when requesting the `my.app` URL on your local environment (*nix system recommended).

Let's first edit the virtual host file(s). Add the following code at the end:

```html
<VirtualHost *:80>
    ServerName my.app
    DocumentRoot YOUR_APP_PATH/public
    SetEnv FUEL_ENV "development"
    <Directory YOUR_APP_PATH/public>
        DirectoryIndex index.php
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Then, open your system host file and add the following line at the end:

```
127.0.0.1 my.app
```

Depending on your environment, you might need to restart Apache after this. You can now access your website at: `http://my.app/`. 
Congratulations! You just have successfully installed the FuelPHP framework. The welcome page shows some recommended directions to continue your project.

**FuelPHP basics**
Now that we have installed a working version of FuelPHP, let's analyze, on a very basic level, how the framework works. We won't go into the details here; the idea is to only understand the necessary information to use the framework. In this section, you are recommended to follow and check our explanations on your installed instance; don't hesitate to explore files and folders, this will make you more comfortable when we will begin our project's implementation. In this section, we will approach the following:

- The FuelPHP file system hierarchy
- MVC, HMVC, and how it works on FuelPHP
- The oil utility
The FuelPHP file system hierarchy

Let's dive into the directory where we have installed FuelPHP. You might want to follow along using a file browser. As this book is being written, the current version of FuelPHP has the following directory hierarchy:

- /docs: contains an HTML version of the framework documentation
- /fuel, which contains:
  - /fuel/app: Everything related to your application. This is where you will work most of the time. We will look into this directory in the upcoming The app directory section.
  - /fuel/core: The core classes and configuration. You should not change anything inside it, unless of course you want to contribute to the FuelPHP core.
  - /fuel/packages: Packages are core extensions, they are bundles containing reusable classes and configuration files. Using the FuelPHP default configuration, this is the only directory where you can install packages (your own as well as from external sources). Notice that there are already five installed packages. We will use each of them in this book.
  - /vendor: This directory contains third-party packages and libraries that are generally not FuelPHP-specific.
- /public: This directory is accessible by external visitors. You want to put here files publicly available, as CSS or JS files for instance.

The app directory

As written earlier, the app directory is where you will work most of the time. Thus, you should be familiar with its hierarchy, which is given as follows:

- /cache: This directory is used to store cache files that improve your application's performance.
- /classes: Classes used by your application:
  - /classes/controller: Where you have to implement your controllers (see the MVC, HMVC, and how it works on FuelPHP section)
  - /classes/model: Where you have to implement your models (see the MVC, HMVC, and how it works on FuelPHP section)
  - /classes/presenter: Where you have to implement your presenters (see the MVC, HMVC, and how it works on FuelPHP section).
• /config: Every configuration file. Since some files are important, we will list them as well:
  ° /config/config.php: Defines important FuelPHP configuration items such as activated packages or security settings.
  ° /config/db.php: Defines database connection information.
  ° /config/routes.php: Defines the application’s routes (we will approach them later in this chapter).
  ° /config/development, config/production, config/staging, config/test: All configuration files in the config/ENV directory, ENV being the current environment, are merged with the ones in the config folder. For instance, if the FuelPHP environment is set to development (as it is by default), the config/development/db.php file will be recursively merged with the config/db.php file. In concrete terms, this means that configuration items defined in the config/ENV/db.php file overwrite those in the config/db.php file. We will illustrate this through an example in The oil utility and the oil console section.

• /lang: Contains the translation files.

• /logs: Contains the log files. The log file path depends on the day it is written. For instance, if you log a message on July 1, 2015, it will be saved in the file located in logs/2015/07/01.php.

• /migrations: Contains the migration files, which allow you to easily alter your database in a structured manner. For instance, if many people are working on the same project, or if there are many instances of the same project (development/production), they make the database change easier. We will often use them in the book.

• /modules: Contains your application’s modules. Each module can be described as a bundle of code that can respond to requests and be easily reused on other projects. We will create a module for the blog project in Chapter 3, Building a Blog Application.

• /tasks: Contains task files, which are classes that can be executed from the command line (for cron jobs for instance).

• /tests: Contains test files, which can be used to automatically test your application. We will approach them in Chapter 5, Building Your Own RESTful API, to test our application.
Building Your First FuelPHP Application

• /tmp: Contains temporary files.
• /vendor: This directory contains third-party libraries and packages only used by your application.
• /views: Contains the view files used by your application (see the MVC, HMVC, and how it works on FuelPHP section).

The packages
The fuel/packages directory contains five default packages that, when activated, can add interesting features to FuelPHP:

• The auth package provides a standardized interface for user authentication. We will use this package in Chapter 5, Building Your Own RESTful API.
• The email package provides an interface to send e-mails using different drivers. We will use this package in Chapter 3, Building a Blog Application.
• The oil package allows you to speed up your application's implementation by generating code files, launching tests and tasks, or providing a CLI PHP console. We will use this package in all chapters and we will explore its features in The oil utility and the oil console section.
• The orm: This package is an improvement of the FuelPHP's core models; it allows them to fetch complex queries and to define the relations between them. We will use this package in Chapter 2, Building a To-do List Application.
• The parser: This package allows your application to render view files in common template systems such as Twig or Smarty. We will use this package in Chapter 5, Building Your Own RESTful API.

We will also create our own package in Chapter 4, Creating and Using Packages.

Class name, paths, and coding standard
In FuelPHP, there are five constants that define the location of the most important directories as follows:

• APPPATH: The application directory (fuel/app)
• COREPATH: The core directory (fuel/core)
• PKGPATH: The packages directory (fuel/packages)
• DOCROOT: The public directory (public)
• VENDORPATH: The vendor directory (fuel/vendor)
You are recommended to read the official documentation about these constants at http://fuelphp.com/docs/general/constants.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | General | Constants)

Please keep in mind, that we will often use these constants in the book to shorten file paths.

An interesting point is that FuelPHP allows you to change quite easily the folder structure: for instance, you can change in the public/index.php file the value of the constants that we just introduced, or you can change the directory where FuelPHP will load modules by changing the module_paths key in the APPPATH/config/config.php configuration file.

You might also have noticed that class names are related to their own path, as given in the following:

- In the app directory, the classes/controller/welcome.php class is named Controller_Welcome
- The classes/model/welcome.php class is named Model_Welcome
- You can notice that classes are named the same way in the fuel/core directory

This result was not achieved by accident; FuelPHP follows by default the PSR-0 standard. You are recommended to read the official documentation about this standard at http://www.php-fig.org/psr/psr-0/.

**MVC, HMVC, and how it works on FuelPHP**

We will now look into one major aspect of the FuelPHP framework – the MVC and HMVC software architecture patterns.

**What is MVC?**

Model-view-controller (MVC) is a software architecture pattern that states that the code should be separated in three categories: models, views, and controllers.
For those who are not familiar with it, let's illustrate this through an example:

![Diagram of FuelPHP architecture]

Suppose a user tries to access your website. The following are some URLs he/she might request:

http://my.app/
http://my.app/welcome/
http://my.app/welcome/hello

Depending on the requested URL, your website is generally expected to return some HTML code and it also sometimes needs to update the database, for instance when you want to save the users' comments.

The returned HTML code is generated by the views, because this is what is received by the browser and indirectly seen by the user.

The database is generally updated through models. In concrete terms, instead of executing raw SQL code to access and update the database, the best practice is to use classes and instances to do so. Each class represents a model that is related to a specific table: for example, the car model would access the cars table. Each class' instance is a model instance linked to a specific row in a table: for example, your car's information can be saved as a car instance that will be linked to a specific row in the cars table. As we use classes instead of raw SQL code, the framework has already implemented frequently needed features such as reading, creating, saving, or deleting model's instances. A further advantage is that, as we used packaged and well-implemented methods to access our database, it can prevent most unintended security breaches that we can create when requesting the database using raw SQL.
The controllers allow the website to handle the user's request by selecting the correct view to send back (the response) and updating the database (through models) if necessary. Controllers handle a specific section of the website: for instance, the car controller will handle everything that is related to cars. Controllers are subdivided by actions that will handle specific features: for instance, the list action of the car controller will return a list of cars in HTML code. In practice, controllers are classes and actions are methods.

When the user requests a URL, the framework will select an action inside a controller to handle it. Those are generally chosen by convention; for instance, when requesting http://my.app/welcome/hello, the framework will choose the hello action inside the welcome controller. Sometimes, they can also be chosen using a routes configuration file that matches URLs to actions and controllers.

The views sometimes need to access models; for example, we need to access the car model's instances when we want to display a list of cars. However, views should never update models or the database; only the controllers and preferably models should do that.

Please note that additional code components as helpers or presenters can be added to ease the development process, but if you understood this section, you got the most important points.

How it works on FuelPHP
Let's illustrate how it works by testing our newly created website. We suppose that your application is available at the following URL:

http://my.app/

Actions and controllers
If you request a random URL, you will probably get a 404 exception. For instance:

http://my.app/should_display_404

But, if you request the following URL, you will display the same page as the home page:

http://my.app/welcome/index

If you request the following URL, you will display a different page:

http://my.app/welcome/hello
Let’s first explain how the last two requests worked. You can notice that both URLs contain the welcome word just after the base URL. You can also find this word in the file name fuel/app/classes/controller/welcome.php; it turns out that welcome is a controller. Now, open this file using your preferred text editor. You will then read the following:

```php
//...
class Controller_Welcome extends Controller
{
    //...
    public function action_index()
    {
        //...
    }
    //...
    public function action_hello()
    {
        //...
    }
}
```

You can notice the action_index and action_hello methods. These functions are called actions. Now, as you have probably guessed, when you request http://my.app/welcome/index, the action_index method will be called. In a more general manner, if you request http://my.app/CONTROLLER/ACTION, the action_ACTION method of the CONTROLLER controller will be called. Let’s test that. Edit the action_index function to add a simple echo at the beginning:

```php
public function action_index()
{
    echo 'Test 1 - Please never print anything inside an action';
    //...
}
```

Now, if you request http://my.app/welcome/index, you will read the printed content at the beginning of the web page. Though this is an easy way to test how things work, never print anything in your action or controller. When you print a message, you are already implementing the view entity; thus, printing something in the controller breaks the MVC pattern.
Views
But then how are the pages rendered? Let's analyze the only line of code in our index action:

```php
public function action_index()
{
    return Response::forge(View::forge('welcome/index'));
}
```

View::forge('welcome/index') returns a View object generated from the fuel/app/views/welcome/index.php view file. We will use this function a lot in this chapter and this book, and will cover all its parameters, but you can read its official documentation in the FuelPHP website:

http://fuelphp.com/docs/classes/view.html#/method_forge. (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | View)

Response::forge(View::forge('welcome/index')); returns a response object created from the View object. Additional parameters allow us to change headers or the page status. A response object contains all the necessary information that will be sent to the browser: the headers and the body (generally the HTML code). You are recommended to read the official documentation on the FuelPHP website at http://fuelphp.com/docs/classes/response.html#method_forge (It can be accessed through the FuelPHP website navigating to DOCS | TABLE OF CONTENTS | Core | Response)

Since the view is generated from the fuel/app/views/welcome/index.php file, open it to discover its content. You can notice that this is the same HTML code as the one displayed when requesting the URL. Just after `<h1>Welcome!</h1>`, add `<p>This is my first view change.</p>`. Now, if you refresh your browser, you will see this message appear under the Welcome! title.

Parameters
It is possible to indicate parameters, both to the actions and to the views. For instance, replace your index action by the following code:

```php
public function action_index($name = 'user', $id = 0)
{
    return Response::forge(
        View::forge(
            'welcome/index',
            array(
```
Building Your First FuelPHP Application

   'name' => $name,
   'id' => $id,

);}
}

And in the fuel/app/views/welcome/index.php view file, replace

   <h1>Welcome!</h1>

by

   <h1>Welcome <?php echo ($name.' (id: '.$id.')'); ?>!</h1>

Now, if you request the following URL:

http://my.app/welcome/index

the title will display Welcome user (id: 0)!

If you request the following URL:

http://my.app/welcome/index/Jane

the title will display Welcome Jane (id: 0)!

And if you request the following URL:

http://my.app/welcome/index/Jane/34

the title will display Welcome Jane (id: 34)!

You might have understood that if you request the following URL:

http://my.app/CONTROLLER/ACTION/PARAM_1/PARAM_2/PARAM3

The action _ACTION method of CONTROLLER will be called with the PARAM_1, PARAM_2, and PARAM_3 parameters. If there are less parameters defined in the URL than required in the method, either, if defined, the parameters take their default values (as illustrated previously), or, if no default value is defined, it will trigger a 404 error.

You can notice that we replaced

   View::forge('welcome/index')
By

```php
View::forge('welcome/index', array(
    'name' => $name,
    'id' => $id,
));
```

View parameters are sent by the second parameter of `View::forge` in an associative array. Here, the associative array has two keys, `name` and `id`, and their values are available inside the view file through the `$name` and `$id` variables.

In a more general manner, if you call the following:

```php
View::forge('YOUR_VIEW', array(
    'param_1' => 1,
    'param_2' => 2,
));
```

When the view file will be executed, parameters will be available through the `$param_1` and `$param_2` variables.

## Routes

Though what we previously observed explains how the standard cases operate

```
http://my.app/CONTROLLER/ACTION
```

we haven't explained why the two following URLs return content though no associated controller and action can be found:

```
http://my.app/
http://my.app/should_display_404
```

For understanding why we have to open the `fuel/app/config/routes.php` configuration file:

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

    'hello(:name)?' => array('welcome/hello', 'name' => 'hello'),
);`
```
Building Your First FuelPHP Application

You can first notice the following two special keys:

- 
  
  
  
  _root_: This defines which controller and action should be called when requesting the website root URL. Note that the value is welcome/index, you can now understand why http://my.app and http://my.app/welcome/index are returning the same content.

- 
  
  
  
  _404_: This defines which controller and action should be called when throwing a 404 error.

Beside specials keys, you can define the custom URLs you want to handle. Let's add a simple example at the end of the array:

```php
'my/welcome/page' => 'welcome/index',
```

Now, if you request the following URL:

http://my.app/my/welcome/page

it will display the same content as in the following URL:

http://my.app/welcome/index

You have probably noticed that there is also another key already defined: hello(/:name)?. The routing system is quite advanced, and to fully understand it you are recommended to take a look at the official documentation:

http://fuelphp.com/docs/general/routing.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Routing)

**Presenters**

You might have seen that the hello action doesn't use the View class to display its content, but instead it uses the Presenter class:

```php
public function action_hello()
{
    return Response::forge(Presenter::forge('welcome/hello'));
}
```
Let's analyze what is happening in this case. First, you can notice that, as for the views, a view file exists at the following path: `fuel/app/views/welcome/hello.php`. If you open this file, you will see that the code is the same as the one displayed when requesting the URL `http://my.app/welcome/hello`, except for one tiny difference. You can find the following code:

```php
<h1>Hello, <?php echo $name; ?>!
```

In a normal view, we would have to define the `name` parameter, except here we didn't. Though, when displaying the web page, this parameter seems to have a defined value (it displays `Hello, World!`). Where could it be defined then?

Probing a little further, you can find another file located at `fuel/app/classes/presenter/welcome/hello.php`. It contains the following:

```php
class Presenter_Welcome_Hello extends Presenter
{
    //...
    public function view()
    {
        $this->name = $this->request()->param('name', 'World');
    }
}
```

This file contains a Presenter class. The `view` function is called before rendering the view and it is here that the `name` parameter is set. It tries to get the name from the request parameter, `name`, but if it is not defined, the default value is `World`.

If you wonder how to change this parameter, refer to the routes. For instance, request the URL `http://my.app/hello/Jane`.

One could then wonder the use of Presenter classes, since we could change the previous code into a more classic view and controller approach.

Let's show its usefulness by an illustration. Suppose you have created an internal website managing the clients of your corporation. Each client is associated to a client category. In your creation, edition, and other forms, you thus display a selectable list of client categories. Each time you display the exact same selectable list, though you access it by using different controllers and actions. You can come up with three solutions:

- You can create a classic view for your selectable list, load the list of client categories inside each of your actions, and pass this list to each view until you reach the location where you want to display your list. The problem is that it would induce a lot of code repetition.
• You can create a classic view and load the list of clients inside this view. This way, you wouldn't have to pass along the necessary parameter. The problem is that you would break the MVC pattern by mixing models and views.

• You can create a Presenter class, load the list inside the Presenter class, use it inside the view file, and display the view file using Presenter::forge. This solution is the best because it doesn't mix views and models but still limits the code duplication.

What is HMVC?

FuelPHP is a Hierarchical Model-View-Controller (HMVC) framework, meaning that it allows you to request internal controllers from your application. In concrete terms, the following code:

```php
echo Request::forge('welcome/index')->execute();
```

will print exactly what the following URL would return:

http://my.app/welcome/index

Though we suggest you to use this feature in moderation, it can come handy when you want to implement and display widgets on several web pages.

You are recommended to read the following resources if you want to learn more about this pattern:


http://stackoverflow.com/questions/2263416/what-is-the-hmvc-pattern

The oil utility and the oil console

The oil utility is a very handy command-line tool. As the rails utility of Ruby on Rails, oil allows you to do the following:

• Easily generate code files: models, controllers, migrations, and entire scaffoldings

• Run tasks and migrations

• Easily install, update, or remove packages

• Test your code using PHPUnit test or a real-time console

• Even run a PHP-built-in web server hosting your FuelPHP application (for PHP >= 5.4)
Though we will use all these features, except the last one in this book, we recommend that you take a look at the official documentation at:

http://fuelphp.com/docs/packages/oil/intro.html

(It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Oil | Introduction)

In this section, we are going to use the oil console, which is an important tool if you want to test your website, or, as in this case, a FuelPHP feature.

First, open your command-line utility and go to the root of your website directory. Then, enter the following line:

```
php oil console
```

This will open the command-line interface oil provides. When you press Enter, something similar to the following should appear:

```
Fuel 1.7.2 - PHP 5.4.24 (cli) (Jan 19 2014 21:18:21) [Darwin]
>>> 
```

You can now type any PHP code and it will be executed. Let's start with something simple:

```
>>> $a = 2
```

If you press Enter, nothing will be printed, but the $a variable will be set to 2. Now, if you want to check a variable value, simply enter its name and then press Enter:

```
>>> $a
2
```
It also works for more complex variables:

```bash
>>> $a = array('a' => 'b', 'c' => 'd')
>>> $a
array (a' => 'b',
             'c' => 'd',
)
```

But be aware, that you might have trouble displaying complex objects.

Let's now test a FuelPHP feature. Earlier, when discussing the `app` directory structure, we explained that the configuration files in the `fuel/app/config` directory were merged with the ones with the same filenames in the `fuel/app/config/ENV` directory, `ENV` being FuelPHP's current environment. We will now test this behavior.

First, let's check FuelPHP's current environment:

```bash
>>> Fuel::$env
development
```

The environment should be set to `development`.

Now, create a PHP file located at `fuel/app/config/test.php` where you will write:

```php
<?php
return array(
    'this_is_the_root_config_file' => true,
);
```

Then create another PHP file located at `fuel/app/config/development/test.php` and write the following:

```php
<?php
return array(
    'this_is_the_dev_config_file' => true,
);
```

and an additional one located at `fuel/app/config/production/test.php`, where you will write the following:

```php
<?php
return array(
    'this_is_the_prod_config_file' => true,
);
```
Now, if you return to the command-line interface, you can load the test configuration file by writing the following:

```php
>>> $conf = Config::load('test', true)
```

You are recommended to read the Config::load official documentation for more information at:

http://fuelphp.com/docs/classes/config.html#method_load. (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | Config)

As explained before, the value returned will be a mix of the fuel/app/config/test.php and the fuel/app/config/development/test.php configuration files:

```php
>>> $conf
array (
    'this_is_the_root_config_file' => true,
    'this_is_the_dev_config_file' => true,
)
```

If we change the FuelPHP environment to production:

```php
Fuel::$env = 'production'; // only do that for testing purposes
```

And load again the test configuration file:

```php
>>> Config::load('test', true, true)
array (
    'this_is_the_root_config_file' => true,
    'this_is_the_prod_config_file' => true,
)
```

The merging will be done with the configuration file in the production folder.

You have probably noticed that we added a third parameter for Config::load. This parameter allows you to clear the configuration cache. If we didn't set it to true, the method would have returned the old configuration we loaded when we were in the development environment.

But what happens when the fuel/app/config/production/test.php and fuel/app/config/test.php configuration files contain the same key? The console can find the answer for us.
Building Your First FuelPHP Application

Change the content of the `fuel/app/config/test.php` configuration file to the following:

```php
<?php
return array(
    'complex_value' => array(
        'root' => true,
    ),
    'this_is_the_root_config_file' => true,
);
```

and change the content of the `fuel/app/config/production/test.php` configuration file to the following:

```php
<?php
return array(
    'complex_value' => array(
        'prod' => true,
    ),
    'this_is_the_root_config_file' => false,
    'this_is_the_prod_config_file' => true,
);
```

Let's now reload the test configuration files as follows:

```php
>>> Config::load('test', true, true)
array (  
    'complex_value' =>
        array (  
            'root' => true,
            'prod' => true,
        ),
    'this_is_the_root_config_file' => false,
    'this_is_the_prod_config_file' => true,
)
```

It is interesting to analyze how the preceding two configuration files have been merged:

- The `this_is_the_root_config_file` key shared by the two configuration files is associated in both cases to a simple value. In the resulting configuration, it is the value from the production file that prevails.
- The `complex_value` key is associated in both cases to an array. The two arrays seem to have been merged in the resulting configuration.
This is because the configuration files are not merged by the `array_merge` native PHP function, but instead by the `Arr::merge` FuelPHP function, which merges arrays recursively. You are recommended to take a look at its official documentation at http://fuelphp.com/docs/classes/arr.html#method_merge (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | Arr)

It should be clear now that the console is a great tool that allows you to test your application. It can also be used as a great complement to the documentation, as you can try FuelPHP methods and their parameters without changing any files in your application.

**Building your first application**

Now that we had a quick overview of the FuelPHP framework, let's build our first tiny application.

Suppose that you are a zoo manager and you want to keep track of the monkeys you are looking after. For each monkey, you want to save the following:

- Its name
- If it is still in the zoo
- Its height
- A description input where you can enter custom information

You want a very simple interface with the following five major features:

- You want to create a new monkey
- You want to edit existing ones
- You want to list all monkeys
- You want to view a detailed file for each monkey
- You want to delete monkeys from the system

The preceding five major features, very common in computer applications, are part of the **Create, Read, Update and Delete (CRUD)** basic operations. This is a perfect example to use the oil utility to generate a scaffold. Oil will quickly generate for us the controllers, models, views, and migrations to handle our monkeys. All we will have to do, then, is to refine the generated code and adapt it to our needs.
Database configuration

As we will store our monkeys into a MySQL database, it is time to configure FuelPHP to use our local database. If you open `fuel/app/config/db.php`, all you will see is an empty array, but, as we demonstrated it in the FuelPHP basics section, this configuration file is merged to `fuel/app/config/ENV/db.php`, ENV being the current FuelPHP’s environment, which in that case is development.

You should, therefore, open `fuel/app/config/development/db.php`:

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

This is the generated default configuration, which you should adapt to your local configuration, particularly the database name (currently set to `fuel_dev`), the username, and password. You must create the database of your project manually.

Scaffolding

Now that the database configuration is set, we will be able to generate a scaffold. We will use the generate feature of the oil utility.

Open the command-line utility and go to your website root directory. To generate a scaffold for a new model, you will need to enter the following line:

```
php oil generate scaffold/crud MODEL ATTR_1:TYPE_1 ATTR_2:TYPE_2 ...
```

where:

- **MODEL** is the model name
- **ATTR_1, ATTR_2**... are the model’s attribute names
- **TYPE_1, TYPE_2**... are attribute types
In our case, it should be as follows:

```php
oil generate scaffold/crud monkey name:string still_here:bool height:float description:text
```

Here we are telling oil to generate a scaffold for the `monkey` model with the following attributes:

- **name**: The name of the monkey. Its type is string and the associated MySQL column type will be VARCHAR(255).
- **still_here**: Whether or not the monkey is still in the facility. Its type is boolean and the associated MySQL column type will be TINYINT(1).
- **height**: Height of the monkey. Its type is float and the associated MySQL column type will be FLOAT.
- **description**: Description of the monkey. Its type is text and the associated MySQL column type will be TEXT.

You can do much more using the oil generate feature, such as generating models, controllers, migrations, tasks, packages, and so on. We will see some of these later in the book, but you are recommended to take a look at the official documentation at [http://fuelphp.com/docs/packages/oil/generate.html](http://fuelphp.com/docs/packages/oil/generate.html) (It can be accessed through the FuelPHP website by navigating to [DOCS | TABLE OF CONTENTS | Oil | Generate](http://fuelphp.com/docs/packages/oil/generate.html)).

When you press `Enter`, you will see the following lines appear:

- Creating migration: `APPPATH/migrations/001_create_monkeys.php`
- Creating model: `APPPATH/classes/model/monkey.php`
- Creating controller: `APPPATH/classes/controller/monkey.php`
- Creating view: `APPPATH/views/monkey/index.php`
- Creating view: `APPPATH/views/monkey/view.php`
- Creating view: `APPPATH/views/monkey/create.php`
- Creating view: `APPPATH/views/monkey/edit.php`
- Creating view: `APPPATH/views/monkey/_form.php`
- Creating view: `APPPATH/views/template.php`

Oil has generated for us nine files, which are as follows:

- A migration file, containing all the necessary information to create the model's associated table
- The model
Building Your First FuelPHP Application

- A controller
- Five view files and a template file

We will take a closer look at these files in the next sections.

You might have noticed that we used the `scaffold/crud` command, and, if you read the official documentation, we could have typed only `scaffold`. This is because two types of scaffold can be generated: `scaffold/crud`, which uses simple models, and `scaffold/orm` alias `scaffold`, which uses the `orm` models. Since using FuelPHP's native ORM was out of the scope of this chapter, and we didn't have to use complex model features such as relations, we chose to use `scaffold/crud`.

Migrating

One of the generated files was `APPPATH/migrations/001_create_monkeys.php`. It is a migration file and contains the required information to create our monkey table. Notice that the name is structured as `VER_NAME`, where `VER` is the version number and `NAME` is the name of the migration.

If you execute the following command line:

```
php oil refine migrate
```

All migration files that have not yet been executed will be executed from the oldest version to the latest version (001, 002, 003, and so on). Once all migration files are executed, oil will display the latest version number.

Once executed, if you take a look at your database, you will observe that not one but two tables have been created:

- `monkeys`: As expected, a table has been created to handle your monkeys. Notice that the table name is the plural version of the word we typed for generating the scaffold; such a transformation was internally done using the `Inflector::pluralize` method. The table will contain the specified columns (`name, still_here`), the `id` column, and also `created_at` and `updated_at`. These columns store the time an object was created and updated, and are added by default each time you generate your models. It is possible to not generate them with the `--no-timestamp` argument.
The `migration` table is automatically created the first time you execute migrations. It keeps track of the migrations that were executed. If you look into its content, you will see that it already contains one row: this is the migration you just executed. You can notice that the row does not only indicate the name of the migration, but also a type and a name. This is because migration files can be placed at many places such as modules or packages (see Chapter 3, Building a Blog Application).

It is important to note that the `migration` table is not the only location where FuelPHP keeps track of the already executed migrations. This information is also stored in `fuel/app/config/ENV/migrations.php`, `ENV` being FuelPHP's environment. If you decide to edit the `migration` table, you might want to also edit or delete this file, as it might prevent the execution of your migrations.

The `refine migrate` feature of oil allows you to have much more control on migrations than simply executing all the new ones. For instance, you can also revert to a previous version using the following command line:

```bash
donload
php oil refine migrate:down
```

Or revert to a specified version using the following command line:

```bash
donload
php oil refine migrate --version=3
```

Or even choose which modules or packages you want to update using the `--modules` or `--package` arguments. To have a complete overview, you are recommended to take a look at the official documentation at http://fuelphp.com/docs/general/migrations.html (it can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Migrations).

But how do migration files allow such complex manipulations? Let's open our migration file located at `APPPATH/migrations/001_create_monkeys.php` to find out. You should see the following:

```php
donload
<?php

namespace Fuel\Migrations;

class Create_monkeys
{
    public function up()
    {
```
\DBUtil::create_table('monkeys', array(  'id' => array(    'constraint' => 11,    'type' => 'int',    'auto_increment' => true,    'unsigned' => true  ),  'name' => array(    'constraint' => 255,    'type' => 'varchar'  ),  'still_here' => array(    'type' => 'bool'  ),  'height' => array(    'type' => 'float'  ),  'description' => array(    'type' => 'text'  ),  'created_at' => array(    'constraint' => 11,    'type' => 'int',    'null' => true  ),  'updated_at' => array(    'constraint' => 11,    'type' => 'int',    'null' => true  ),  ), array('id'));

public function down(){
    \DBUtil::drop_table('monkeys');
}
The file contains a class named `Create_monkeys` that has the following two methods:

- **up**: This method defines how to update your data structure. Note that this migration file creates the monkey table using the `DBUtil::create_table` method, but you could perfectly execute a handmade SQL request to do that. Though migrations are generally used to update your database, you can also use them to update custom data files or old configuration files.

  In some cases, if you want to implement your own migrations, you might find the idea of using your application’s methods (in models or helpers) attractive. Though it can allow you to limit your code duplication, it is not recommended. This is because, for compatibility reasons, the migration files are intended to stay in your application indefinitely, whereas your application’s code can evolve a lot. Therefore, by changing or deleting a method in your application, you might unexpectedly break some migration files (that use this method) without even noticing it, making the future installation of your application complicated.

- **down**: This method defines how to cancel all changes that were made by the `up` method. Suppose you realize that the feature was a mistake and you want to revert to an older version: this is when this method will be executed. In our case, the method simply deletes the monkey table.

  If the information contained in the table is important, it might be a good idea to instead move the table, for instance, to an archive database. A human mistake could have disastrous consequences otherwise.

The migration files are a powerful tool and their usefulness increase tenfold as the number of instances and the number of developers working on the same project rise. Using them from scratch is always a good decision.

### Using your application

Now that we have generated the code and migrated the database, our application is ready to be used. You might have noticed during the generation that a controller was created at `APP_PATH/classes/controller/monkey.php` and that the route configuration file was not changed, meaning that the controller must be accessible through the default URL.

Let’s request, then, the URL `http://my.app/monkey`.
As you can notice, this web page is intended to display the list of all monkeys, but since none have been added, the list is empty:

Then, let's add a new monkey by clicking on the Add new Monkey button. The following web page should appear:
You can enter your monkey's information here. There are, however, several inconsistencies:

- All fields are required, meaning that you can't leave any field empty, otherwise errors will be triggered preventing you from adding the monkey. This is not what we might want for the description field.
- Though you can enter anything you want in the **Height** field without triggering any error, if you enter anything other than a float, it will be replaced by 0. We might want to trigger an error in such a case.
- **Still here** can only have two values: 0 or 1 (false or true). Though the type of the associated database column is correct, the generated form uses a standard input where we might want a checkbox.

The form is certainly not perfect, but it is a great start. All we will have to do is refine the code a little bit.

Once you have added several monkeys, you can again take a look at the listing page as follows:

<table>
<thead>
<tr>
<th>Monkeys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing Monkeys</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Jane</td>
</tr>
<tr>
<td>Tony</td>
</tr>
</tbody>
</table>

Add new Monkey

Again, this is a great start, though we might want to refine it a little bit: display **Yes** and **No** instead of 1 and 0, respectively, for the **Still here** column, and remove the **Description** column because there might be too much text to display.

Each item on the list has three associated actions: **View**, **Edit**, and **Delete**.
Let's first click on **View**:

![Welcome!](image)

Again this is a great start, though we will also refine this web page.

You can return back to the listing by clicking on **Back** or edit the monkey by clicking on **Edit**. Accessed from either the listing page or the view page, it will display the same form as when creating a new monkey, except that the form will be prefilled of course.

Finally, if you click on **Delete**, a confirmation box will appear to prevent any miss clicking:

![Diagram](image)

**Refining the application**

Now that we took a look at our interface, let's refine our application so that it becomes more user-friendly. In this section, we will explore the files that have been generated by oil and try to adapt them to our needs.
Refining the monkey listing

During the previous section, two small issues bothered us for the monkey's listing:

- We wanted more explicit values than 0 and 1 for the Still here column
- We wanted to remove the Description column

We know that the list appears when requesting the following URL:

http://my.app/monkey

You have probably noticed that in this URL we indicated a controller, but no action. It is important to know that, by default and without any routing configuration involved, this URL is equivalent to http://my.app/monkey/index

So, in fact, we are calling the index action of the monkey controller. If we open the generated controller at APPPATH/classes/controller/monkey.php, we will read the following:

```php
<?php
class Controller_Monkey extends Controller_Template{
    //...
}
```

First, you can notice that Controller_Monkey extends Controller_Template instead of Controller, as we saw before in Controller_Welcome. Controller_Template is an extension of Controller that adds template support. The idea is that most of the time your web pages will have the same layout: the headers, footers, and menus generally stay the same, regardless of the web pages you are in. Templates allow you to achieve this by limiting the code duplication.

By default, Controller_Template is associated with the APPPATH/views/template.php template that was generated by oil. If you open this file, you will see that it generates the HTML code around the page content. You will also probably notice that it prints the $title and $content variables. We will find out how to set their values by exploring the index action. If you go back to the Monkey controller, the action_index method should contain the following:

```php
public function action_index()
{
    $data['monkeys'] = Model_Monkey::find_all();
    $this->template->title = "Monkeys";
    $this->template->content = View::forge('monkey/index', $data);
}
```
The first line stores all the monkeys' instances into the $data['monkeys'] variable. In a general manner, MODEL::find_all() returns all a model's instances, but it is definitely not the only method that retrieve instances. These methods will be discussed more thoroughly in Chapter 2, Building a To-do List Application.

The second and third lines set the $title and $content variables displayed in the template file. If you change the second line by $this->template->title = "My monkeys"; and then refresh the web page, you will see that its title has changed accordingly.

The third line sets the $content variable to a view instance that, from what we have observed in the previous sections, executes the view file located at APPPATH/views/monkey/index.php with the $monkey variable set to all monkeys' instances. Let's open this view file. You should see the following:

```php
<h2>Listing Monkeys</h2>
<br>
<?php if ($monkeys): ?>
<table class="table table-striped">
<thead>
<tr>
<th>Name</th>
<th>Still here</th>
<th>Height</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<?php foreach ($monkeys as $item): ?>
<tr>
<td><?php echo $item->name; ?></td>
<td><?php echo $item->still_here; ?></td>
<td><?php echo $item->height; ?></td>
<td><?php echo $item->description; ?></td>
<td>
  <?php /* Action buttons */ ?></td>
<?php endforeach; ?>
</tbody>
<?php else: ?></table>
<?php endif; ?>
<?php /* Add new Monkey button */ ?></p>

We have found where the table is displayed, so it is time to make our changes.

First, remove the Description column by removing the following:

```html
<th>Description</th>
```

and

```html
<td><?php echo $item->description; ?></td>
```

Then, let's refine how the Still here attribute is displayed by replacing the following:

```html
<td><?php echo $item->still_here; ?></td>
```

by

```html
<td><?php echo $item->still_here ? 'Yes' : 'No'; ?></td>
```

The Still here column should now display Yes and No instead of 1 and 0, respectively.

**Refining the monkey detailed view**

On the list, when clicking on an item's View link, a detailed view of the monkey appears. We would like to change two details here:

- As in the previous section, display more explicit values for the Still here attribute
- Currently, if you save a monkey with a multiline description, it is displayed on one line only

First, if you are on a detailed view page, you can notice that the URL is similar to http://my.app/monkey/view/1
This means we are calling the view action of the monkey controller with the first and only parameter set to 1. The view action is quite similar to the index action, as you can see in the following snippet:

```php
public function action_view($id = null)
{
    if (is_null($id)) Response::redirect('monkey');

    $data['monkey'] = Model_Monkey::find_by_pk($id);

    $this->template->title = "Monkey";
    $this->template->content = View::forge('monkey/view', $data);
}
```

The first line simply checks if the parameter of the action (associated to the $id variable) is actually set, and otherwise redirects the user (using the Response::redirect method) to the listing page.

The second line stores the monkey with ID $id into the $data['monkey'] variable. The find_by_pk (pk for primary key) method of a model finds one of its instances by its primary key. As we explained earlier, models' methods will be discussed more thoroughly in Chapter 2, Building a To-do List Application.

Just to be perfectly clear, requesting the URL http://my.app/monkey/view/ID will load the monkey instance with id = ID.

The third and fourth lines, as in the previous section, set the template variables. The template content is set to the view located at APPPATH/views/monkey/view.php.

```html
<h2>Viewing #<?php echo $monkey->id; ?></h2>

<p>
    <strong>Name:</strong> <?php echo $monkey->name; ?></p>

<p>
    <strong>Still here:</strong> <?php echo $monkey->still_here; ?></p>

<p>
    <strong>Height:</strong> <?php echo $monkey->height; ?></p>

<p>
    <strong>Description:</strong>
</p>
```
It is time to do some changes.

Replace:

```php
<?php echo $monkey->still_here; ?>
```

By:

```php
<?php echo $monkey->still_here ? 'Yes' : 'No'; ?>
```

And replace:

```php
<?php echo $monkey->description; ?>
```

By:

```php
<div><?php echo nl2br($monkey->description); ?></div>
```

### Allowing an empty description

One of the issues we pointed out previously, is that the description field is required, though we want to be able to enter an empty value.

First, open your browser and request the following URL:

http://my.app/monkey

Click on the Add a new Monkey button, and you can see you are redirected to http://my.app/monkey/create

If you take a look at the page source, you will find that the form’s action attribute is actually the same URL:

```html
<form class="form-horizontal" action="http://my.app/monkey/create" accept-charset="utf-8" method="post">
```

It means that whether we are opening the monkey’s creation form or submitting it, we will always call the create action of the monkey controller. We should then read how this action is implemented:

```php
public function action_create()
{
    if (Input::method() == 'POST')
    {
```
Building Your First FuelPHP Application

```php
$val = Model_Monkey::validate('create');

if ($val->run())
{
    // Saves the model (out of this chapter scope)
}
else
{
    Session::set_flash('error', $val->error());
}

$this->template->title = "Monkeys";
$this->template->content = View::forge('monkey/create');
```

As you can notice, the action is able to know whether or not it is accessed through a POST request by using `Input::method()`. You are recommended to take a look at the official documentation of the `Input` class at [http://fuelphp.com/docs/classes/input.html](http://fuelphp.com/docs/classes/input.html) (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | Input).

`Model_Monkey::validate('create')` returns an object that seems to define whether or not the object can be saved (depending on what `$val->run()` returns).

This is a method from the `Monkey` model, so we should look into it. Open `APP PATH/classes/model/monkey.php`:

```php
<?php
class Model_Monkey extends Model_Crud
{
    protected static $_table_name = 'monkeys';

    public static function validate($factory)
    {
        $val = Validation::forge($factory);
        $val->add_field('name', 'Name', 'required|max_length[255]');
        $val->add_field('still_here', 'Still Here', 'required');
        $val->add_field('height', 'Height', 'required');
        $val->add_field('description', 'Description', 'required');

        return $val;
    }
}
```
The file contains the `Model_Monkey` class that extends `Model_Crud` and allows us to handle the monkey instances.

First, you can notice the `$_table_name` static attribute that defines the table name where the objects are saved (here, all our monkeys are saved into the `monkeys` table).

And then there is the `validate` static method we are looking for. It returns a `Validation` object, that in our case will check that:

- The `name` attribute is not empty and its length is less than 255 characters
- `still_here`, `height`, and `description` are not empty

For more detail about this class, you are recommended to read the official documentation at [http://fuelphp.com/docs/classes/validation/validation.html](http://fuelphp.com/docs/classes/validation/validation.html) (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | Validation | Introduction)

In our case, simply comment or remove the following line:

```php
$val->add_field('description', 'Description', 'required');
```

Checking whether the height is a float

It is now easy to check if the height is a float. As we know that monkeys are generally not taller than 4 feet, we can even add a numerical constraint. In the `validate` method of `Model_Monkey`, replace the following line:

```php
$val->add_field('height', 'Height', 'required');
```

by

```php
$val->add_field('height', 'Height', 'required|numeric_between[0,6]');
```
Using a checkbox instead of an input for the still_here attribute

This change will be a bit more complex. First, still in the validate method of Model_Monkey, remove the following line as we won't need this validation:

```
$val->add_field('still_here', 'Still Here', 'required');
```

Now, if you go back to our create action in Controller_Monkey (located at APPPATH/classes/controller/monkey.php), you will see that the template content is set to the view located at APPPATH/views/monkey/create.php. If you look at the file content, it is pretty simple:

```
<h2>New Monkey</h2>
<br>
<?php echo render('monkey/_form'); ?>
<p><?php echo Html::anchor('monkey', 'Back'); ?></p>
```

For your information, the render method is an alias of View::render, and in this case equivalent to View::forge. This illustrates that it is possible to render views inside other views. It can be convenient to prevent code repetition; the view located at APPPATH/views/monkey/edit.php also renders the same view (monkey/_form), and this makes sense since the forms displayed are exactly the same, whether you create a new monkey or edit an existing one.

Since we want to edit the form to replace the still_here input by a checkbox, open the view located at APPPATH/views/monkey/_form.php and replace the following lines:

```
<?php
    echo Form::input('still_here', Input::post('still_here', isset($monkey) ? $monkey->still_here : '' ), array('class' => 'col-md-4 form-control', 'placeholder' => 'Still here'))
?>
```
By

```php
<?php
echo Form::checkbox(
    'still_here',
    1,
    Input::post(
        'still_here',
        isset($monkey) ? $monkey->still_here : true
    )
);
?>
```

In the code above, the first parameter is the name attribute of the checkbox. The second parameter is the value attribute of the checkbox. The third parameter determines whether the checkbox is checked or not. You can notice that, when we create a new monkey (and therefore no monkey is set), the checkbox will be checked by default. You are recommended to read the official documentation for more information about the Form class at http://fuelphp.com/docs/classes/form.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | Core | Form)

Finally, you are probably aware that the `still_here` POST attribute won't be defined if the checkbox is unchecked when submitting the form. Thus, we need to define a default value when retrieving the `still_here` POST attribute, not only in the create action but also in the edit action. In both the methods, replace the following:

```
Input::post('still_here')
```

by

```
Input::post('still_here', 0)
```

Our solution works, but, in most cases, hard-coding a default value is not a good idea. When indicating a default value, for a request parameter or a configuration item, the best is to define this value inside a centralized configuration file and load it from there. Always avoid hard-coding constants, even for default values.
Setting custom routes

Last but not least, we don't want to display FuelPHP's welcome screen when requesting the root URL, but instead the monkeys' listing. For doing that we will have to change the routes' configuration file located at APPPATH/config/routes.php.

Replace:

    '_root_' => 'welcome/index',

By:

    '_root_' => 'monkey/index',

When requesting:

http://my.app/

You should now see your monkey listing.

Removing useless routes and files

Now that our project is working as intended, it might be a good idea to clean it:

- Remove APPPATH/classes/controller/welcome.php as we don't need this controller anymore
- Remove the APPPATH/classes/presenter folder
- Remove the APPPATH/views/welcome folder
- And remove the _404_, _hello(/:name)?_, my/welcome/page keys from the routes' configuration file located at APPPATH/config/routes.php.

A few notes about deploying your application

Now that you have a working application, you might want to publish it on hosts. Handling this is quite easy, the longer part being sending the project's files (using FTP, Git, or any other tool depending on your hosting service), but there are a couple of things you should know.

First, you have to set your apache FUEL_ENV environment to production. An easy way to do that is to edit public/.htaccess and uncomment the second line:

    SetEnv FUEL_ENV production
Keep in mind, that in this case you will have two different files between your local environment and your production environment, so it will be prone to human error. You are recommended to read the official documentation at http://fuelphp.com/docs/general/environments.html (It can be accessed through the FuelPHP website by navigating to DOCS | TABLE OF CONTENTS | FuelPHP | Environments)

If you are using a shared hosting solution, keep in mind that, as explained in The simplest way section, you should take additional security precautions

**Summary**

In this chapter, we have seen the very basics of the FuelPHP framework and we have built our first project. We have learnt how to install FuelPHP, used the oil command line to generate code files and migrate our application, understood how routes work, and seen how models, views, presenters, and controllers interact with each other.

Though you are now able to create an application and implement basic features, you might not be yet ready for more complex projects. In the next chapter, we will improve your skills by using FuelPHP's Object Relational Mapper (ORM).
Where to buy this book

You can buy FuelPHP Application Development Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.