Community Experience Distilled

Getting Started with Phalcon

Design, implement, and deliver superior web applications using the most popular PHP framework available

Stephan A. Miller

[PACKT] open source*
PUBLISHING
community experience distilled

Getting Started with Phalcon

Design, implement, and deliver superior web applications using the most popular PHP framework available

Stephan A. Miller

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Phalcon

First published: January 2014

Production Reference: 1170114

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

ISBN 978-1-78328-767-3

`www.packtpub.com`

Cover Image by Manu Gangadhar (`manug30@gmail.com`)

Credits

Author
Stephan A. Miller

Reviewers
Vladimir Kolesnikov
Calin Rada
Ivan Vorontsov

Acquisition Editors
Richard Harvey
Antony Lowe

Lead Technical Editor
Sharvari Tawde

Technical Editors
Rosmy George
Ankita Thakur

Copy Editors
Deepa Nambiar
Alfida Paiva

Project Coordinator
Ankita Goenka

Proofreader
Paul Hindle

Indexer
Rekha Nair

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Stephan A. Miller is an SEO specialist, writer, and software developer from Kansas City, MO. He has expertise in operating systems, a few databases, and a handful of programming languages. He is currently working as a software contractor. He is also the author of *Piwik Web Analytics Essentials* published by *Packt Publishing*. He also blogs when he has the time at `http://stephanmiller.com`.

I would like to thank the developers of Phalcon for fielding any questions I had, and for helping me get this book finished. I would also like to thank my co-workers at Penton who learned Phalcon with me on the job.

About the Reviewers

Vladimir Kolesnikov has been developing server and web applications for over 20 years. He is the Vice President of Development at Goldbar Enterprises, LLC. Over his career, he has been involved in designing and writing very complex and high-performance software.

His passion for writing fast, robust, and optimized software introduced him to Phalcon, and he soon joined the development team. He mainly concentrates on Phalcon's stability and performance.

In his free time, he enjoys the company of his beloved wife, writes a technical blog at `http://blog.sjinks.pro/`, contributes to open source projects, and studies foreign languages.

Calin Rada is a full-stack developer with over nine years of experience, developing the server, network, and hosting environment; data modeling; business logic; the API layer, Action Layer, or MVC; user interface; and user experience, and understanding what the customer and business needs are. He is continuously hungry about learning new things and working with new technologies.

Currently, he occupies the position of IT Director at SOLOMO365 (`www.solomo365.com`). He is also the creator of SharePathy (`www.sharepathy.com`), the social discovery network built on top of the Phalcon framework.

Ivan Vorontsov admires the process of building applications and the power of open source. He loves developing client/server applications and is always looking for a reason to develop a useful program. His first application was Anti-cheat for the game World Of Warcraft coded with Delphi. He is also interested in game engines such as NeoAxis and Unity 3D. Currently, he is actively developing applications in the field of web-based technologies using modern PHP frameworks such as Phalcon, Symfony2, and ZF2. His favorite motto is: "Everything is possible!"

Table of Contents

Preface

The developers of Phalcon made it their mission to develop the fastest and most efficient PHP framework around, and they have done a good job of completing that mission. With the speed of C programming, simplicity of PHP, and the structure of a framework, Phalcon streamlines web application development.

Getting Started with Phalcon is an introduction to using Phalcon to develop web applications. You will learn by building a blog application using Phalcon Developer Tools and web tools to flesh out the CRUD skeleton for the application quickly, and then modifying it to add features.

As features are added to the blog, you will learn how to use other Phalcon functionalities such as the Volt template engine, view helpers, PHQL, validation, encryption, cookies, sessions, and event management.

What this book covers

Chapter 1, *Installing Phalcon*, covers installing the Phalcon PHP extension on Linux, Windows, or Mac, and configuring PHP, Apache, or Nginx.

Chapter 2, *Setting Up a Phalcon Project*, covers setting up a project in Phalcon manually using Phalcon Developer Tools or Phalcon web tools.

Chapter 3, *Using Phalcon Models, Views, and Controllers*, covers the MVC structure of Phalcon.

Chapter 4, *Handling Data in Phalcon*, covers Phalcon Models in depth, PHQL, session data, and filtering and sanitizing data.

Chapter 5, *Using Phalcon's Features*, covers even more features in Phalcon, including hashing passwords, controlling user access, setting cookies, and using view partials, logging, and view helpers.

Who this book is for

This book is intended for PHP developers who want to learn how to use the Phalcon PHP framework. Some knowledge of PHP but no prior knowledge of MVC frameworks is required.

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, path names, dummy URLs, user input, and Twitter handles are shown as follows: "So, we have to locate the `php.ini` file and edit it."

A block of code is set as follows:

```
phalconBlog/
  app/
    config/
    controllers/
    library/
    logs/
    models/
    plugins/
    views/
      index/
      layouts/
  public/
    css/
    files/
    img/
    js/
    temp/
```

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

```
sudo apt-get install git
sudo apt-get php5-devphp-mysqlgcc
```

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: "So, search for the word **Compiler** in your browser window."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can also find all the code used in the book by visting GitHub page `http://eristoddle.github.io/phalconBlog/`.

Errata

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

Piracy

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

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

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

Questions

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

1
Installing Phalcon

This chapter will walk you through installing Phalcon on your chosen platform. The installation process differs a little bit depending on which operating system you happen to be using. Since Phalcon is a PHP extension written in C, installation is a little bit more involved than downloading an archive of PHP files and uploading them to your server.

But installation should still take you less than an hour. Depending on your system and experience with PHP extensions, it could take only minutes. At the end of this chapter, you will have a high performance PHP framework that will simplify the process of developing your application and help you get your project up and running faster.

In this chapter, you will learn how to:

- Meet Phalcon's system requirements
- Find the correct Phalcon DLL for use on a Windows machine
- Install the Phalcon DLL on a Windows machine
- Install Phalcon on Linux
- Install Phalcon on Mac
- Install Phalcon on FreeBSD
- Edit your `php.ini` to use Phalcon
- Configure Apache for Phalcon

Phalcon system requirements

You will of course need a web server in order to use Phalcon. Apache is the most common, but if you have an Nginx or Cherokee web server, that will work too. You can even use PHP's built-in web server as your Phalcon development server if you wish. For the purposes of this book, we will assume you have Apache installed.

Along with that, you will need PHP installed. While Phalcon will work with any version of PHP higher than Version 5.3.1, it is recommended that you use PHP 5.3.11 or greater. Versions of PHP before 5.3.11 still have security flaws and memory leaks that were fixed in the later versions.

But that is about all you need. Phalcon is not really very picky. There are a few other requirements you may need to meet on each specific operating system, and we will cover them in the individual installation processes.

Installing Phalcon on Windows

In this section, we will walk through installing Phalcon on Windows. Even if you aren't installing the extension on Windows, please read through this section. Many of the steps involved, such as locating your `php.ini`, apply to all operating systems.

The Phalcon extension for Windows comes compiled as a DLL. However, you must locate the correct DLL for your version of PHP. The first requirement when installing Phalcon in Windows is a PHP binary compiled with Windows Visual Studio 9.0. It is possible to have a Windows PHP installation compiled with another version of Visual Studio. If you have one of these other versions, you will have to install the correct version of PHP if you want Phalcon to work on your system.

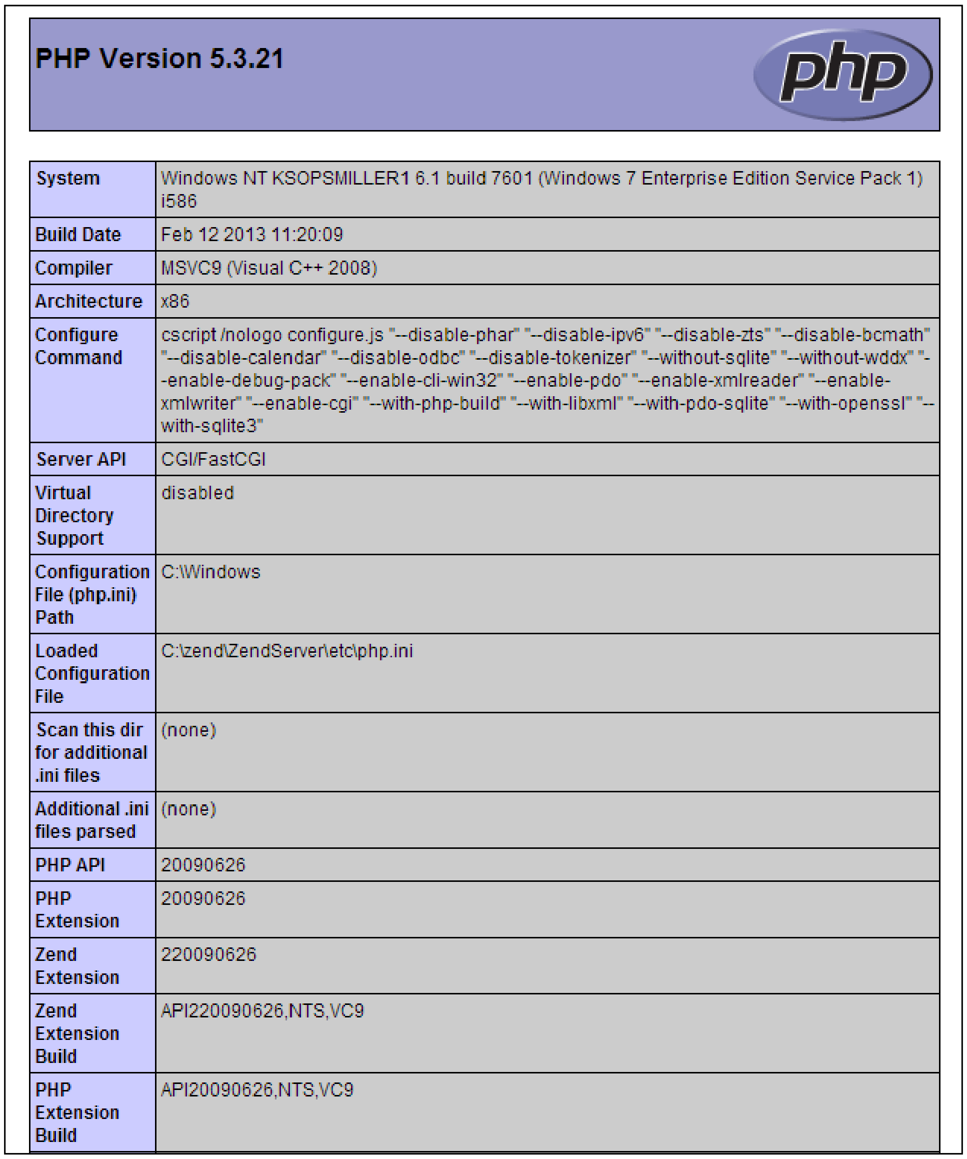
So, let's take a look at your PHP installation to discover just which version of PHP you have installed. First, you need to locate the document root of your web server. Or, in other words, you need to locate the folder on your computer that corresponds to the page you get when you browse to `http://localhost` or the folder on your server that maps to the home page of `http://yourdomain.com`. We are going to create a PHP file that will tell us everything we need to know to pick the right version of the PhalconPHP DLL and drop it in that folder.

So, open up your text editor or IDE and create a new file. Name it `info.php`. In that file, insert the following code:

```
<?php
  phpinfo();
?>
```

Save the file to your document root. Where your document root is depends upon which web server stack you have installed on your machine. It could be Apache directly from the official site, such as XAMPP, Wamp, or AMPPS, or one of the other versions of Apache, PHP, and MySQL stacks available. Your best bet is to visit the site where you downloaded your chosen version of Apache and read the documentation.

Once you have saved the file, browse to `http://localhost/info.php` and you should see a page like the following screenshot:

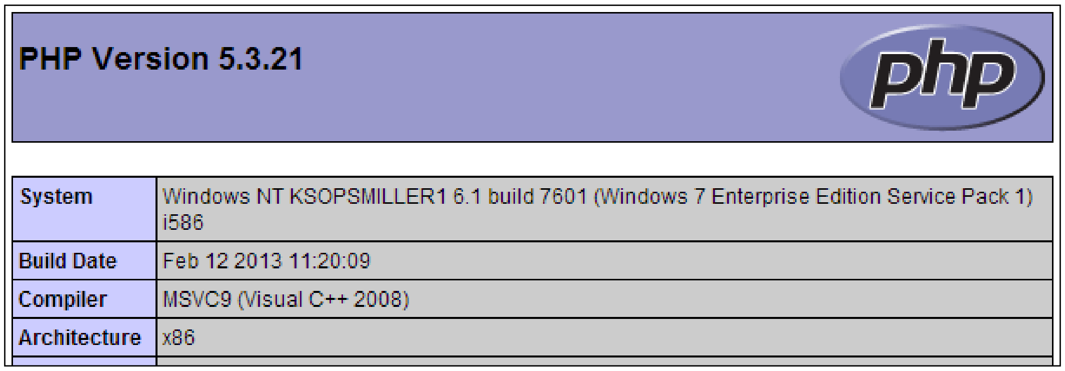


PHP Version 5.3.21

System	Windows NT KSOPSMILLER1 6.1 build 7601 (Windows 7 Enterprise Edition Service Pack 1) i586
Build Date	Feb 12 2013 11:20:09
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86
Configure Command	cscript /nologo configure.js "--disable-phar" "--disable-ipv6" "--disable-zts" "--disable-bcmath" "--disable-calendar" "--disable-odbc" "--disable-tokenizer" "--without-sqlite" "--without-wddx" "--enable-debug-pack" "--enable-cli-win32" "--enable-pdo" "--enable-xmlreader" "--enable-xmlwriter" "--enable-cgi" "--with-php-build" "--with-libxml" "--with-pdo-sqlite" "--with-openssl" "--with-sqlite3"
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	C:\Windows
Loaded Configuration File	C:\zend\ZendServer\etc\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20090626
PHP Extension	20090626
Zend Extension	220090626
Zend Extension Build	API220090626,NTS,VC9
PHP Extension Build	API20090626,NTS,VC9

You will notice the PHP version at the top of the page. We will be paying attention to the first two numbers separated by a dot. They will either be **5.3** as shown in the previous screenshot or 5.4. If they are 5.2 or less, you will need to update your PHP installation.

Next, you want to make sure that your version of PHP was compiled with Microsoft Visual Studio 9.0. So, search for the word **Compiler** in your browser window. The command to open the search box on most browsers is *Ctrl* + *F*. It is usually near the top of the page as shown in the following screenshot. If your PHP was compiled with Visual Studio 9.0, you should see **MSVC9 (Visual C++ 2008)**, as shown in the following screenshot:



PHP Version 5.3.21

System	Windows NT KSOPSMILLER1 6.1 build 7601 (Windows 7 Enterprise Edition Service Pack 1) i586
Build Date	Feb 12 2013 11:20:09
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86

Below the **Compiler**, you will see the **Architecture** listing. It will say **x86** or **x64**. In the previous screenshot, PHP was compiled for the **x86** architecture.

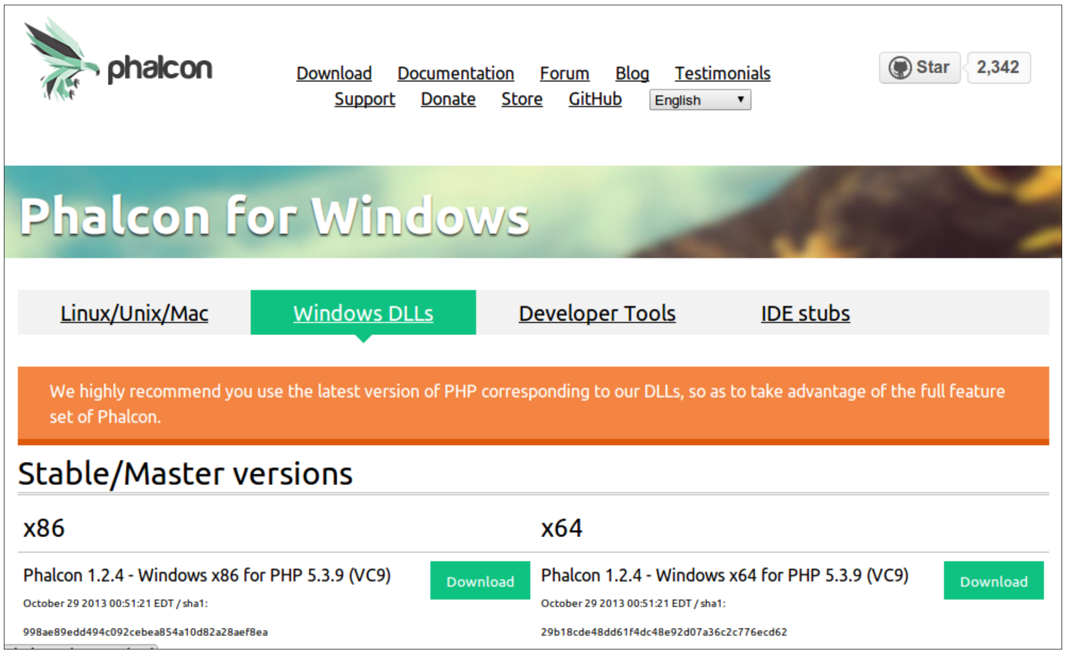
Next, we need to see if you have a thread-safe version of PHP or not. So, search for PHP Extension Build in your browser window. Next to this, you will see the build version of your PHP installation. If you see an NTS at the end of this number, then you do not have a thread-safe version of PHP.

Now that we have these pieces of information, we can go to `http://phalconphp.com/en/download/windows`.

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. You can also find all the code used in the book by visting GitHub page `http://eristoddle.github.io/phalconBlog/`.

Here, we will see the page as shown in the following screenshot, which lists multiple versions of the Phalcon PHP extension:



We highly recommend you use the latest version of PHP corresponding to our DLLs, so as to take advantage of the full feature set of Phalcon.

Stable/Master versions

x86

Phalcon 1.2.4 - Windows x86 for PHP 5.3.9 (VC9)

October 29 2013 00:51:21 EDT / sha1:

998ae89edd494c092cebea854a10d82a28aef8ea

x64

Phalcon 1.2.4 - Windows x64 for PHP 5.3.9 (VC9)

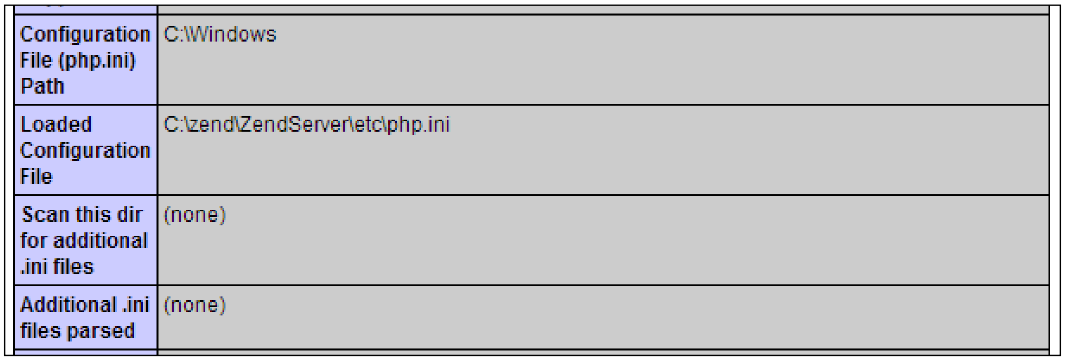
October 29 2013 00:51:21 EDT / sha1:

29b18cde48dd61f4dc48e92d07a36c2c776ecd62

It may look daunting, but we just did the research that will enable us to choose the correct version. We know whether we have a thread-safe version or not. We know whether we have an x86 or x64 architecture. And we know whether we have PHP 5.3 or 5.4. Don't worry about the last number in the PHP version. We just need to download the extension that fits our configuration. The Phalcon extension for PHP 5.3.9 will work for all versions of PHP 5.3 and the extension for PHP 5.4.0 will work for all versions of PHP 5.4.

After downloading the extension, unzip it. Inside the zip file will be a DLL. Next, we need to move this DLL into the folder where your other PHP extensions exist. If you have XAMPP installed, this folder will most likely be `c:\xampp\php\ext`, and with Wamp, that would be `c:\wamp\bin\php_version_number\ext`, where `php_version_number` is replaced with the version number of PHP you have installed. For other versions of web server stacks, you will have to visit the documentation to find the PHP extension folder.

Now we have the extension where it needs to be, but we aren't done yet. We need to have PHP load the extension when it runs. So we have to locate the `php.ini` file and edit it. But guess what, we can use our `info.php` file to tell us where this file is. So load it again in your browser and search for **Loaded Configuration File**. Listed beside this entry is its location, as shown in the following screenshot:

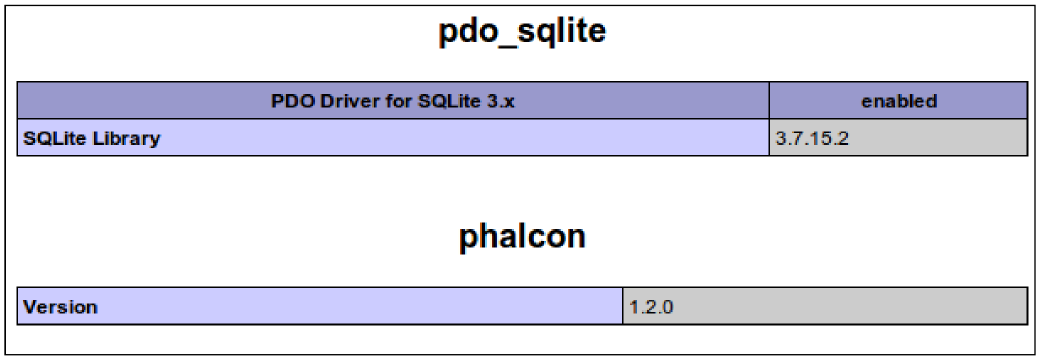


Configuration File (php.ini) Path	C:\Windows
Loaded Configuration File	C:\zend\ZendServer\etc\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)

Once you locate this file, open it with your text editor or IDE and scroll to the very bottom. This file can be quite long. At the very bottom of this file, you will want to add the following line to tell PHP to load Phalcon:

```
extension=php_phalcon.dll
```

Save the file. Some server stacks like XAMPP have desktop software to stop and start Apache, while others like Apache from the official site only have a tray icon to control stopping and starting of the server. Or, you can just restart Apache from the command line. Restart Apache now and Phalcon is installed. But let's check to make sure. Load `http://localhost/info.php` in your browser again and search for Phalcon and you should be greeted by a page as shown in the following screenshot:



pdo_sqlite

PDO Driver for SQLite 3.x	enabled
SQLite Library	3.7.15.2

phalcon

Version	1.2.0

Installing Phalcon on Linux

Linux is the most common operating system for a web server. Phalcon does require you to have root access to your web server. If you are on a shared server, you won't get root access, but you may be able to talk to your hosting provider to install Phalcon for you. If you have a VPS, cloud server, or dedicated server, you most likely have root access. If not, you can ask your hosting provider to give you access.

It is beyond the scope of this book to walk you through installing Phalcon on each and every Linux installation, but there are prebuilt versions of the Phalcon extension for some versions of Linux.

For Debian Linux, you will find a repo at the following link:

`http://debrepo.frbit.com/`

For Arch Linux, you can go to the following link to download a PKGBUILD:

`http://aur.archlinux.org/packages.php?ID=61950`

And for OpenSUSE, you can find a package at the following link:

`http://software.opensuse.org/package/php5-phalcon`

For all other versions of Linux, we will have to compile Phalcon ourselves. This is not hard, and if you have been using Linux for a while, you are probably used to compiling your own software by now. First, we have to be sure we have the following software installed:

- Git
- GCC Compiler
- PHP development resources

If you don't have these installed, you can install them via the command line.

For Ubuntu, the command line is as follows:

```
sudo apt-get install git
sudo apt-get php5-devphp-mysqlgcc
```

For Fedora, CentOS, and RHEL, the command line is as follows:

```
sudo yum install git
sudo yum install php-develphp-mysqlnd ccc libtool
```

For OpenSUSE, the command line is as follows:

```
yast2 -igit
yast2 -iphp5-pear php5-develphp5-mysqlgcc
```

For Solaris, the command line is as follows:

```
pkginstall git
pkg install gcc-45 php-53 apache-php53
```

Once you have met all the requirements, open up a terminal window and type the following commands to download the Git repository and compile the Phalcon extension:

```
git clone git://github.com/phalcon/phalcon.git
cd cphalcon/build
sudo ./install
```

On some Linux installations, the compilation may fail because `libpcre3-dev` is missing. You can fix this issue by installing this package with your Linux package manager.

You just compiled your first PHP extension. Now we need to add the reference to our extension to the `php.ini` file. We can find it by creating the `info.php` file as discussed in the *Installing Phalcon on Windows* section of this chapter, loading it in our browser, and searching the resulting page for **Loaded Configuration File**. Locate the file listed here in your file-system and add the following line to the very bottom of the file:

```
    extension=phalcon.so
```

Now save the file. If you do happen to be installing Phalcon for PHP 5.4 on a Debian type Linux, the procedure for installing a new PHP extension has changed. It is no longer a best practice to edit the main `php.ini` file. In the same folder where you found the `php.ini` file, usually with the `/etc/apache5/` path, you will find a `mods-available` folder. It is in this folder that you will be creating a custom `.ini` file just for Phalcon, and you will call it `phalcon.ini`. This file will be empty because we aren't adding any custom configuration settings to Phalcon. Then, to enable the extension, we just run the following in the command line:

```
php5enmod phalcon
```

Now restart your web server. The command to restart Apache differs depending on the type of Linux. For Debian and Ubuntu type Linuxes, you can type the following command:

```
sudo service apache2 restart
And for Red Hat, CentOS or Fedora, you would type:
sudo service httpd restart
```

Installing Phalcon on Mac

To install Phalcon on Mac, the steps are basically the same as installing it on Linux. You must compile the extension from the source code. But first there are some requirements. You need to have Git, php5-devel, and Xcode with the command line tools installed. Xcode is available for free from the official Mac site, but is not installed by default.

Also, Xcode's command line tools are not installed with Xcode by default. In order to make sure to install these tools, open Xcode's **Preference** panel, choose the **Download** tab, and click on the **Install** button located next to the **Command Line Tools** listing.

To install Git, just point your web browser to `http://git-scm.com/download/mac`, download the file, and install it. You will also need to install php5-devel if you don't already have it on your system. How you install this depends on the package manager you have installed on your Mac. There are also some prebuilt php5-devel binaries available for the Mac. A search on Google for "install Mac php5-devel" should point you in the right direction.

Now that we have met all the requirements, the steps are the same as installing Phalcon on Linux. First, use Git to pull the source code down to your computer and build the extension by issuing the following commands in the Terminal:

```
git clone git://github.com/phalcon/phalcon.git
cd cphalcon/build
sudo ./install
```

Then, use the steps we went over in the *Installing Phalcon on Windows* section of this chapter to locate your `php.ini` file. Once you have found your `php.ini` file, open it and add the following line to the very bottom of your file:

```
extension=phalcon.so
```

Next, restart Apache and you're set.

Installing Phalcon on FreeBSD

If you happen to be installing Phalcon on FreeBSD, you've got it easy. There is a port of Phalcon available. It can be installed with the following command:

```
pkg_add -r phalcon
```

Or

```
export CFLAGS="-O2 -fno-delete-null-pointer-checks"cd/usr/ports/www/
phalcon&& make install clean
```

Optional Phalcon dependencies

By now, you should have a working installation of Phalcon. But to use some of Phalcon's functionality, you may need to install one, some, or all of the following PHP extensions:

- PDO
- PDO/MySQL
- PDO/PostgreSQL
- PDO/SQLite
- PDO/Oracle
- mbstring
- Mcrypt
- openSSL
- Mongo

You will only need these if you plan to use the functionality they provide. For the purposes of this book, you will need to make sure that you have the PDO, PDO/MySQL, Mcrypt, and mbstring extensions installed.

Finding help

Phalcon is constantly being improved, but every now and then, you may run into issues where the standard installation process may not work or you may run into other bugs. It is beyond the scope of this book to cover everything that could happen while installing or using Phalcon. To find help with issues like this, I suggest three places to look:

- Phalcon's official forum: `http://forum.phalconphp.com/`
- Phalcon's bugtracker: `https://github.com/phalcon/cphalcon/issues`
- Phalcon's official documentation: `http://docs.phalconphp.com/en/latest/`

Summary

In this chapter, we learned how to choose the correct version of the Phalcon PHP extension for our operating system. We also learned how to install the extension on a multitude of systems. We edited our `php.ini` file so that Phalcon will now load with our PHP installation and become an integral part of the language.

In the next chapter, we will begin developing our Phalcon blog by putting together the skeleton structure of our site.

2
Setting Up a Phalcon Project

Before we start writing the code for our blog, we are going to take a look at the skeleton we need to have in place for our project. First, we will walk through creating the folder structure and necessary files manually. After we have finished doing it the hard way, we will learn how to use Phalcon Developer Tools in order to do all of this automatically with a few simple terminal commands. Our code will change and morph as we go. This is because as we delve deeper into the framework, we will be refactoring our old code to handle new features that we want to add to our blog application.

In this chapter, we will learn the following topics:

- The folder structure of Phalcon project
- The Phalcon .htaccess file
- How to use a .ini file
- How to create a Phalcon bootstrap file
- How to use Phalcon Developer Tools

Folder structure

Unlike many PHP frameworks, Phalcon doesn't care about your folder structure. You can create any folder structure you want, but for our blog tutorial, we are going to use a basic MVC-structured site. There are many different variations of the folder structure in various frameworks. We will start out with the following code snippet, which has the same structure that the Phalcon Developer Tools will be generating for us later in the chapter:

```
phalconBlog/
  app/
    config/
    controllers/
```

```
        library/
        logs/
        models/
        plugins/
        views/
          index/
          layouts/
      public/
        css/
        files/
        img/
        js/
        temp/
```

The only executable file we have in our `public` folder is our bootstrap file, `index.php`, which we will learn about later in this chapter. Everything else are static files loaded by the browser. In our `app` folder, we have folders for our `models`, `views`, and `controllers`.

There are MVC file structures smaller than this, and there are structures that contain the `template`, `partial`, and `module` folders. As your project gets bigger and more complicated, you may want to switch to a multimodule folder structure, about which you can read at `http://docs.phalconphp.com/en/latest/reference/applications.html`. And for simpler sites, such as a basic API, you may want to opt for a much smaller structure such as the micro application structure. You can find an example of the micro MVC structure at `http://docs.phalconphp.com/en/latest/reference/tutorial-rest.html`.

Using .htaccess files

We are going to use `.htaccess` files in our application, and for this step, you need to make sure `mod_rewrite` is enabled on your Apache server. On many Apache installations, this mod is installed and enabled by default. If `mod_rewrite` is not enabled on your server, you will need to enable it, which could involve running the `sudo a2enmod rewrite` command or uncommenting the `LoadModule mod_rewrite.so` line in the `httpd.conf` or `apache2.conf` file. It all depends on your Apache installation.

For Nginx configuration, you can read the section at the end of this chapter and skip using `.htaccess` files.

Now, we need to tell Apache how to serve our Phalcon project. First, we will have to hide our `app` folder from public view and redirect all visitors to our `public` folder. Therefore, we need to create a `.htaccess` file for our project's root folder, the `phalconBlog` folder. So, open a text editor, create a new file, and insert the following lines of code in the file:

```
<IfModule mod_rewrite.c>
  RewriteEngine on
  RewriteRule ^$ public/     [L]
  RewriteRule (.*) public/$1 [L]
</IfModule>
```

Save this file as `.htaccess` in the `phalconBlog` folder. Now, all visitors to the `phalconBlog` folder will be directed to the `public` folder.

This is not the only option when it comes to serving our `public` folder in Apache. We could edit our Apache configuration file for the virtual host serving our project and set the document root to our `public/` directory. In this case, we would not need this first `.htaccess` file, only the `.htaccess` file we are about to create.

Phalcon has beautiful URLs built in. In other words, URLs can have a more appealing structure; for example, `http://www.blog.com/post/1` is better than `http://www.blog.com/?post=1`. Since every visit must go through the bootstrap file that we will learn about next, we will also have to create a `.htaccess` file for the `public` folder. So, create the file in the `public` folder and insert the following lines of code in the file:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
</IfModule>
```

This `.htaccess` file tells Apache to load a file or folder in the `public` folder if it exists, and if not, send the visitor to the `index.php` file.

The bootstrap file

The `index.php` file that we are redirecting all of our requests to is the cornerstone of our application. In this file, we load any configuration we set up and put together the various pieces of the Phalcon framework into a complete application. In this step, we will start with a simple bootstrap file and then make it more configurable in the next step with the use of a `.ini` file.

Handling exceptions

First, we want to make sure we catch any exceptions our application may throw at us. So, open up your text editor or IDE and create an `index.php` file and place it in the `public` directory. In the `index.php` file, insert the following code snipppet:

```
<?php
  try {
    //Our code
  } catch(\Phalcon\Exception $e) {
    echo "PhalconException: ", $e->getMessage();
  }
?>
```

The first thing you should notice is that we haven't included a file in our application, yet we can still reference a Phalcon exception. Phalcon is a PHP extension and is now part of your PHP installation. There is no need to include any PHP files. We are using a PHP try-catch construction to catch any exception that Phalcon may throw.

Dependency Injection

Phalcon uses a Dependency Injection container to handle all the services you may need to use in your application. Whenever our application needs one of these services, it asks for it by name from the container. Phalcon is designed to be decoupled, meaning that you can use all or part of its features rather than being required to use a specific base set of services. This Dependency Injection container, or DI, is the glue that holds all of our chosen services together.

Let's create a DI. We will be using Phalcon's `FactoryDefault` DI. So, in the `try` brackets of your `index.php`, insert the following code snippet:

```
try {
  //Create a DI
  $di = new Phalcon\DI\FactoryDefault();

  //Set up our views
  $di->set('view''view', function(){
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir(__DIR__ . '../app/views/');
    return $view;
  });
} catch(\Phalcon\Exception $e) {
  echo "PhalconException: ", $e->getMessage();
}
```

So, we have created our DI. Next, we register our view service with it, telling Phalcon where to find our view files. Now, let's set the model and controller folders for our application.

Autoloaders

Now, we add our autoloaders.

```
//Create a DI
  $di = new Phalcon\DI\FactoryDefault();

  //Set up our views
  $di->set('view', function(){
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir(__DIR__ . '../app/views/');
    return $view;
  });

  //Our autoloaders
  $loader = new \Phalcon\Loader();
  $loader->registerDirs(array(
    __DIR__ . '../app/controllers/',
    __DIR__ . '../app/models/'
  ))->register();
```

This will tell Phalcon where to find our controller and model files. Currently, we are using only two folders in our blog project, but it is possible to load more folders with Phalcon Loader if we need to. We also created `library` and `plugin` folders, which we will be using later.

Initializing our application

Now, we only have to add a couple more lines of code to initialize our application to handle requests. We create an instance of `Phalcon\MVC\Application` and name it `$application`. Then, we invoke `$application->handle->getContent()`.

```
<?php
  try {
    //Create a DI
    $di = new Phalcon\DI\FactoryDefault();

    //Set up our views
    $di->set('view', function(){
      $view = new \Phalcon\Mvc\View();
      $view->setViewsDir(__DIR__ . '../app/views/');
```

```
        return $view;
      });

      //Our autoloaders
      $loader = new \Phalcon\Loader();
      $loader->registerDirs(array(
        __DIR__ . '../app/controllers/',
        __DIR__ . '../app/models/'
      ))->register();

      //Initialize our application
      $application = new \Phalcon\Mvc\Application($di);
      echo $application->handle()->getContent();
    } catch(\Phalcon\Exception $e) {
      echo "PhalconException: ", $e->getMessage();
    }
  ?>
```

Now that we have hardcoded everything into our bootstrap file, we are going to change it up a bit and make our application a bit more flexible. Also, we are going to use the classic `.ini` file as the configuration file for our application.

Using a configuration file

Using a configuration file gives the developer a simple place to edit various settings on a website that might otherwise change in different environments. If you move a project from your localhost to another server, all you have to do is edit this file, and the application should run wherever you put it.

You can use a standard PHP array to store Phalcon configurations, JSON files, or the `.ini` file format. We will be using a `.ini` file for readability, but the PHP array format is native to PHP and loads quicker. Later in this book, we will add more settings to our file, but for now, our project's folder paths look like good additions to our `.ini` file. Most likely, we won't have to change these settings, but it gives us the option to change our mind later. So, you should create a new folder, `config`, in the `app` folder of your blog project, and create a new file, `config.ini`, in your editor and save it in this folder. In this file, add the following lines of code:

```
[phalcon]
controllersDir = "../app/controllers/"
modelsDir = "../app/models/"
viewsDir = "../app/views/"
```

Now, we have a `.ini` file hanging out all alone. It's time to load it in our bootstrap file. So, open up the `index.php` file again. We are going to make some changes.

The `Phalcon\Config` component can currently read the configuration files of either the `.ini` or PHP array types. For our project, it is `.ini` all the way. Right after you create the DI, you are going to load the `.ini` file.

```
//Create a DI
$di = new Phalcon\DI\FactoryDefault();

//Load ini file
$config = new \Phalcon\Config\Adapter\Ini(__DIR__ . '../app/config/
config.ini');
$di->set('config', $config);
```

Now, we have our configuration file loaded, but it's not doing anything; it's just taking up space. It is time to replace our hardcoded file paths with our configuration data. The only path we cannot replace is the location of the `.ini` file itself.

So, we are going to take the following bit of code out of our index file:

```
//Set up our views
$di->set('view', function(){
  $view = new \Phalcon\Mvc\View();
  $view->setViewsDir(__DIR__ . '../app/views/');
  return $view;
});

//Our autoloaders
$loader = new \Phalcon\Loader();
$loader->registerDirs(array(
  __DIR__ .'../app/controllers/',
  __DIR__ .'../app/models/'
))->register();
```

Change it to the following code snippet:

```
//Set up our views
$di->set('view', function() use ($config) {
  $view = new \Phalcon\Mvc\View();
  $view->setViewsDir($config->phalcon->viewsDir);
  return $view;
});

//Our autoloaders
$loader = new \Phalcon\Loader();
```

```
$loader->registerDirs(array(
  $config->phalcon->controllersDir,
  $config->phalcon->modelsDir
))->register();
```

Each heading in our `.ini` file became a child object of our `$config` object, and each setting under the header became a variable in that Phalcon child object. Now that we have a `.ini` file, we can choose a few deployment methods to ensure our application will run on each environment we need to put it on. We could create a `dev.ini` file for our development server and a `prod.ini` file for our production server. Our bootstrap file could then load a `.ini` file based on an Apache environmental variable. Alternatively, we could use a build script that uploads the correct `.ini` file to the correct environment.

Phalcon Developer Tools

Now that we have done everything the hard way to get a feel of how Phalcon works, let's try doing the same in a quick, easy way. Phalcon Developer Tools are a set of tools that will help you get your project up and running quickly by generating a skeleton code for you. We will be using these in the upcoming chapters to generate a lot of things and then review the files that were created to learn how they work.

If you don't want to install the developer tools, no problem. The current skeleton we created manually will work as a base for the rest of the application with a few modifications that we will cover in this section. Phalcon is a very flexible framework. We will also be going through everything we generate with the developer tools in sufficient detail so that you will be able to create all your code manually if you so choose to. But I would advise you to use developer tools wherever you can and then fill in the gaps with custom code. It can really shorten your development time. First, let's install these tools.

Installing Phalcon Developer Tools

Phalcon Developer Tools can be downloaded from the official Git repository available at `https://github.com/phalcon/phalcon-devtools`. Alternatively, you can use Composer to install it as a library in your project. The reason why we have a `library` folder in our project's `app` folder is so that we can store PHP libraries that we may need in our application. Composer is a PHP package manager that will take care of these libraries with only a few commands. You can read more about Composer at `http://getcomposer.org/`.

Firstly, create a new file, `composer.json`, in the `library` folder, add the following lines of code in it, and save the file:

```
{
    "require": {
        "phalcon/devtools": "dev-master"
    }
}
```

This is simply a `.json` file telling Composer that we want to include Phalcon Developer Tools in our project. If you have `curl` installed on your computer, open a terminal or an SSH window and browse to your `library` folder located at `phalconBlog` and run the following command:

```
curl -s http://getcomposer.org/installer | php
```

In Linux, you will most likely have `curl` installed. If you do not, you can simply download `composer.phar` from `http://getcomposer.org/download/` and put the file in your `lib` folder.

Next, we tell Composer to add Phalcon Developer Tools in our `lib` folder.

```
php composer.phar install
```

This will work for you if your system's `PATH` variable has the location of your PHP executable in it. If you are working on Linux and PHP was installed with a package manager, then it most likely is. In Windows, you may have to locate your PHP executable and put its location in your path.

After Phalcon Developer Tools are downloaded via Composer, you must add the path `phalconBlog/app/library/vendor/phalcon/devtools/phalcon.php` to your system's `PATH` variable so that the Phalcon command will work.

Phalcon can also be installed with PEAR, the PHP package manager. For the purposes of this book, we have glossed over the details of installing the developer tools. Covering each operating system would be beyond the scope of this book:

- You can find out more details on all installations at `https://github.com/phalcon/phalcon-devtools`
- For more details on installing Phalcon Developer Tools on Windows, please visit `http://docs.phalconphp.com/en/latest/reference/wintools.html`
- For details on installing these tools on Mac, see `http://docs.phalconphp.com/en/latest/reference/mactools.html`
- And for Linux, visit `http://docs.phalconphp.com/en/latest/reference/linuxtools.html`

Generating a project skeleton

Using one command, we are going to do just about everything we have already done in this chapter. Open up the command-line interface and browse to where you want to generate your project. You can actually delete our previously created project. You can either start over or build your new project in another location. Now, type the following command:

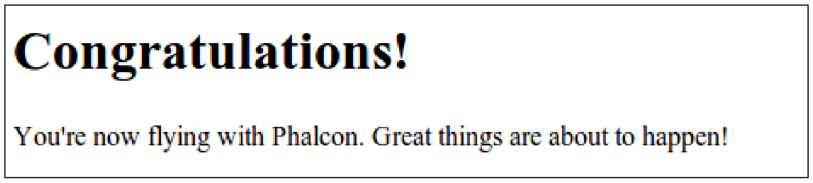
```
phalcon project phalconBlog --use-config-ini --enable-webtools
```

Suddenly, we will have a complete project structure built for us. You will notice a few differences between the generated project and the one we built from scratch. We will go over a few of those differences to see what has changed.

If you didn't use Phalcon Developer Tools, you will want to change your `config.ini` file to this version. This configuration file is going to have a few more settings, as follows:

```
[database]
adapter  = Mysql
host     = localhost
username = root
password = root
dbname     = phalconblog

[application]
controllersDir = ../app/controllers/
modelsDir      = ../app/models/
viewsDir       = ../app/views/
pluginsDir     = ../app/plugins/
libraryDir     = ../app/library/
baseUri        = /phalconBlog/
cacheDir       = ../app/cache/

[models]
metadata.adapter = "Memory"
```

It adds a set of dummy database configurations. It adds a few more folders to the application section of our `.ini` file. The `baseUri` setting may cause you some issues. If the project you are creating is running in a subdirectory of your localhost, the generated setting will work. If you browse directly to the localhost and see the following message, you will have to change this `baseUri` setting:


Congratulations!

You're now flying with Phalcon. Great things are about to happen!

You will not notice any issues with the `baseUri` setting until you start including CSS and JavaScript files in your application and these files don't load. So, if your site is running in the main directory, change that line to the following line of code:

```
baseUri        = /
```

You will also notice that the `views` folder has some new files in it with the `.volt` extension. There is an `index.volt` file in your index folder located at `app/views` and another `index.volt` file in the `views` folder located at `app`. Let's take a look at the file in your `views` folder located at `app`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Phalcon PHP Framework</title>
  </head>
  <body>
    {{ content() }}
  </body>
</html>
```

This file is the main layout for your whole application. Every view that your application serves will be wrapped by this file. The content of the body tags will be provided by the view. That is what the `{{ content() }}` tag is for. This is a Volt template tag. Volt is a template engine that is built into Phalcon. We will learn more about it in *Chapter 3, Using Phalcon Models, Views, and Controllers*.

The view that is served when you browse to your localhost is in the `index.volt` file in your `index` folder located at `app/views`.

```
<h1>Congratulations!</h1>

<p>You're now flying with Phalcon. Great things are about to happen!
  </p>
```

This is just static HTML that lets you know everything has worked.

You will also notice that your bootstrap or your `index.php` file located at `public` is different. It now looks like the following code snippet:

```
<?php

error_reporting(E_ALL);

try {

  /**
   * Read the configuration
   */
  $config = new \Phalcon\Config\Adapter\Ini(__DIR__ . "/../app/config/
    config.ini");

  /**
   * Read auto-loader
   */
  include __DIR__ . "/../app/config/loader.php";

  /**
   * Read services
   */
  include __DIR__ . "/../app/config/services.php";

  /**
   * Handle the request
   */
  $application = new \Phalcon\Mvc\Application($di);

  echo $application->handle()->getContent();

} catch (\Exception $e) {
  echo $e->getMessage();
}
```

It still loads our `.ini` file from the same place. However, it now loads two new files, `loader.php` and `services.php`, from our `config` folder. The rest of the file is about the same. The generated version simply breaks up the autoloader and the Dependency Injection container into separate files and includes them as configuration files, which does make a lot more sense. First, let's take a look at `loader.php` file located at `app/config`:

```
<?php

$loader = new \Phalcon\Loader();

/**
 * We're a registering a set of directories taken from the
   configuration file
 */
$loader->registerDirs(
  array(
    $config->application->controllersDir,
    $config->application->modelsDir
  )
)->register();
```

This is basically the same code that we used to have in our old bootstrap file. Now, let's look at `services.php` file located at `app/config`:

```
<?php
use Phalcon\DI\FactoryDefault,
  Phalcon\Mvc\View,
  Phalcon\Mvc\Url as UrlResolver,
  Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter,
  Phalcon\Mvc\View\Engine\Volt as VoltEngine,
  Phalcon\Mvc\Model\Metadata\Memory as MetaDataAdapter,
  Phalcon\Session\Adapter\Files as SessionAdapter;
/**
 * The FactoryDefault Dependency Injector automatically registers the
   right services providing a full stack framework
 */
$di = new FactoryDefault();
/**
 * The URL component is used to generate all kind of urls in the
   application
 */
$di->set('url', function() use ($config) {
  $url = new UrlResolver();
  $url->setBaseUri($config->application->baseUri);
  return $url;
}, true);
/**
 * Setting up the view component
 */
$di->set('view', function() use ($config) {

  $view = new View();

  $view->setViewsDir($config->application->viewsDir);

  $view->registerEngines(array(
    '.volt' => function($view, $di) use ($config) {

      $volt = new VoltEngine($view, $di);

      $volt->setOptions(array(
        'compiledPath' => $config->application->cacheDir,
        'compiledSeparator' => '_'
      ));

      return $volt;
    },
    '.phtml' => 'Phalcon\Mvc\View\Engine\Php'
  ));

  return $view;
}, true);
/**
 * Database connection is created based in the parameters defined in
   the configuration file
 */
$di->set('db', function() use ($config) {
  return new DbAdapter(array(
    'host' => $config->database->host,
    'username' => $config->database->username,
    'password' => $config->database->password,
    'dbname' => $config->database->dbname
  ));
});
/**
 * If the configuration specifies the use of a metadata adapter, use
   it or use memory otherwise
 */
```

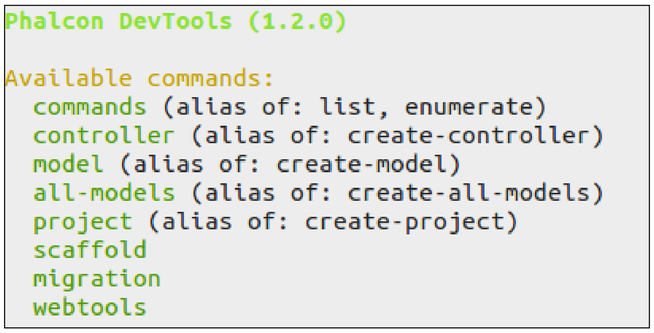
```
$di->set('modelsMetadata', function() {
  return new MetaDataAdapter();
});
/**
 * Start the session the first time a component requests the session
   service
 */
$di->set('session', function() {
  $session = new SessionAdapter();
  $session->start();
  return $session;
});
```

There is definitely a lot more going on in this file than we can cover in this chapter. However, we will touch up on the new services that got added here in later chapters, and will end up adding a few more ourselves. You can see where we set our views directory with this line of code: `$view->setViewsDir($config->application->viewsDir);`. But, we definitely have more services loaded now than in our old bootstrap file. Now, we have a URL resolver, a template engine service, a database adapter, and a session service. These will become important later on in the book.

Available commands

We will be exploring more commands in future chapters to generate the files we will need for our application, but here are some commands to get you started in exploring what these tools can do for you.

Typing `Phalcon` in your command line will give you the output (as shown in the following screenshot):



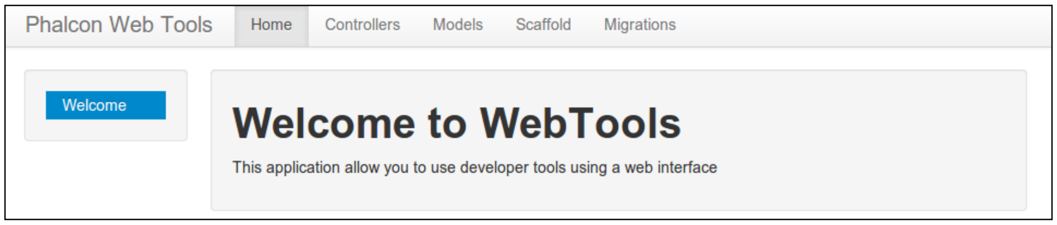
```
Phalcon DevTools (1.2.0)

Available commands:
  commands (alias of: list, enumerate)
  controller (alias of: create-controller)
  model (alias of: create-model)
  all-models (alias of: create-all-models)
  project (alias of: create-project)
  scaffold
  migration
  webtools
```

The `commands` command will actually print out this same list of available commands. We have already taken a look at the project command. We will explore the rest in the upcoming chapters.

Yet, you may have noticed that when we built our project with Phalcon Developer Tools, we used the `-enable-webtools` command. We have an even easier option while writing code for our application; we can generate code right in the browser.

The web interface

Once we create a project with the command-line tool, we can do just about everything else that Phalcon Developer Tools can do in the browser. Navigate to `http://localhost/phalconBlog/webtools.php`, and you should see a page as shown in the following screenshot:



Welcome to WebTools

This application allow you to use developer tools using a web interface

We will learn about the power of this amazing tool in the next chapter.

IDE stubs

Another feature of Phalcon Developer Tools is that they contain IDE stubs for the code hint and autocomplete features of your IDE. Because Phalcon is compiled in C, your IDE won't be able to index the classes and functions of the actual framework, like it could with the Zend framework, for example, which is written in PHP. The IDE stub files mirror the classes and functions in the Phalcon framework so that an IDE has access to their names and documentation.

In the `devtools` folder of our project located at `app/library/vendor/phalcon`, an `ide` folder will be present, and there are stubs for various versions of Phalcon in this `ide` folder. To use the current version of the stubs in our IDE, we would just add the path to the `1.2.4` folder of our project's external libraries in our IDE. The process of doing this will differ depending on which IDE you use, but you can find specific instructions to add the Phalcon IDE stubs to PhpStorm at `http://phalconphp.com/en/download/stubs`.

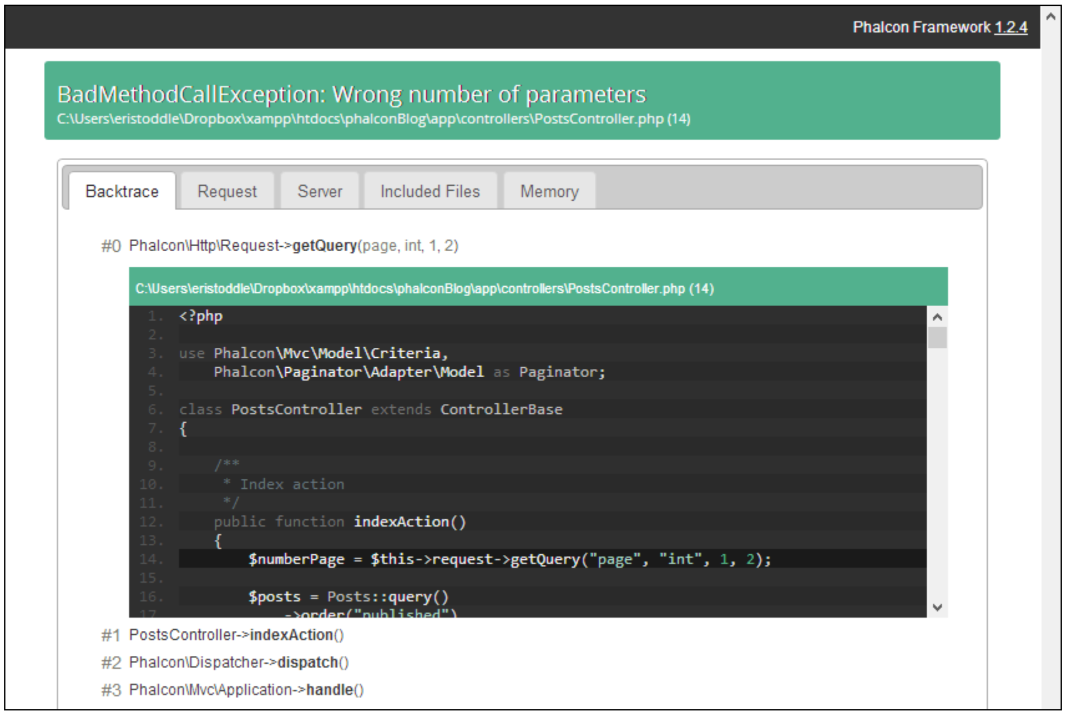
Setting up Phalcon\Debug

An optional step is to add the `Phalcon\Debug` component to the application. It only takes two lines of code to add it, and it will go a long way in helping you track down the causes of errors when they occur.

To add `Phalcon\Debug`, we are going to open up our bootstrap file, `index.php` file located at `public`, and add a couple of lines of code at the beginning of our code, remove the try-catch construction, leaving all the code in the `try` part untouched, and add our debugger.

```
<?php

error_reporting(E_ALL);

$debug = new \Phalcon\Debug();
$debug->listen();

/**
 * Read the configuration
 */
$config = new \Phalcon\Config\Adapter\Ini(__DIR__ . "/../app/config/
  config.ini");
```

Now, when we have an exception, instead of a standard ugly error, we will see a page that looks like the following screenshot:



```
BadMethodCallException: Wrong number of parameters
C:\Users\eristoddle\Dropbox\xampp\htdocs\phalconBlog\app\controllers\PostsController.php (14)

#0 Phalcon\Http\Request->getQuery(page, int, 1, 2)
C:\Users\eristoddle\Dropbox\xampp\htdocs\phalconBlog\app\controllers\PostsController.php (14)
<?php

use Phalcon\Mvc\Model\Criteria,
    Phalcon\Paginator\Adapter\Model as Paginator;

class PostsController extends ControllerBase
{

    /**
     * Index action
     */
    public function indexAction()
    {
        $numberPage = $this->request->getQuery("page", "int", 1, 2);

        $posts = Posts::query()
            ->order("published")
#1 PostsController->indexAction()
#2 Phalcon\Dispatcher->dispatch()
#3 Phalcon\Mvc\Application->handle()
```

Instead of the standard stack trace, we are presented with the section of the code where the exception arose to examine for possible issues. We can also click on the Phalcon functions listed in our trace in order to be taken to the official Phalcon documentation page. `Phalcon\Debug` has a lot of features. You can learn about them more by creating exceptions in the application on purpose and examining the output or by reading more about the debugger at `http://docs.phalconphp.com/en/latest/reference/debug.html`.

Nginx configuration

It is also possible to run Phalcon on Nginx. To run our project in Nginx, we would use the following configuration and skip using the `.htaccess` files, which are of no use in Nginx:

```
server {

    listen   80;
    server_name localhost;

    index index.php index.html index.htm;
    set $root_path '/var/www/phalcon/public';
    root $root_path;

    try_files $uri $uri/ @rewrite;

    location @rewrite {
        rewrite ^/(.*)$ /index.php?_url=/$1;
    }

    location ~ \.php {
        fastcgi_pass unix:/run/php-fpm/php-fpm.sock;
        fastcgi_index /index.php;

        include /etc/nginx/fastcgi_params;

        fastcgi_split_path_info       ^(.+\.php)(/.+)$;
        fastcgi_param PATH_INFO       $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_
          info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_
          name;
    }
```

```
        location ~* ^/(css|img|js|flv|swf|download)/(.+)$ {
            root $root_path;
        }

        location ~ /\.ht {
            deny all;
        }
    }
```

You should only have to edit the `server_name` variable if yours is not `localhost` and the `$root_path` variable to the path to the `public` folder in your project. For more details on configuring Nginx for Phalcon projects, see `http://docs.phalconphp.com/en/latest/reference/nginx.html`, and for configuring Cherokee web server, see `http://docs.phalconphp.com/en/latest/reference/cherokee.html`.

Summary

In this chapter, we learned how to set up our skeleton application. We created folders, `.htaccess` files, and our bootstrap file. Then, we made our application more configurable by using a `.ini` file. Finally, we used Phalcon Developer Tools to make the whole job a lot quicker and easier.

In the next chapter, we will finally be serving pages in the browser when we learn about Phalcon's models, views, and controllers.

3
Using Phalcon Models, Views, and Controllers

Now we will start building a functional blog application. Using Phalcon Developer Tools, we will generate our models, controllers, and views, and then examine the code in sufficient detail so that we can write it ourselves if we have to. We will also learn about forms in Phalcon, which are in the arsenal of view helpers that are components we can use repeatedly in our application. There is no need to reinvent the wheel here. We will also look at the Volt template engine, which is built into Phalcon.

In this chapter, the following topics will be covered:

- The basics of the Model-View-Controller pattern
- Creating a Phalcon model
- Creating a Phalcon controller
- Creating a Phalcon view
- Using the Volt template engine
- Using Phalcon view helpers

Introducing MVC

PHP has changed a lot over the years. In the beginning, it wasn't even object-oriented. But that was added in PHP 3, and has improved in the PHP versions since then. These changes led the way for PHP-based frameworks. And most of these frameworks use the MVC pattern, which was made popular by both Python's Django and Ruby's Ruby on Rails, for use in web frameworks. Let's briefly review each part of the MVC pattern.

View

The view outputs the user interface. This is its only job. Sometimes, a view includes a template engine such as Smarty or Phalcon's own built-in template engine, Volt. We will be using Volt in our blog project. But the truth is that since you can mix `PHP` and `HTML` in the same file, PHP itself acts like a template engine.

Controller

Controllers are the switchboard operators of our application. The actions in a controller pass parameters to the view to display and respond to user input. When we fill out a blog form in our application and click on the **Save** button, the data we posted is sent through our controller. As we are creating a blog post, the controller uses its `createAction` function to create a new post instance in our model. When this is done, it sends a message back to our view stating that our post was successful.

Model

Most of the logic of our application should reside in the model. It should know that each one of our blog posts has a user that posted it, and that these two things are related. It should also know that each blog post can have one or many tags. If we have a database connected to our application, the model should handle the changes to the database. When we create a new post in our application, our controller simply passes the data we posted in our form to the correct action. Our model should handle the creation of the new post's record in the database as well as any records that relate to the user who posted it or the tags we added to it.

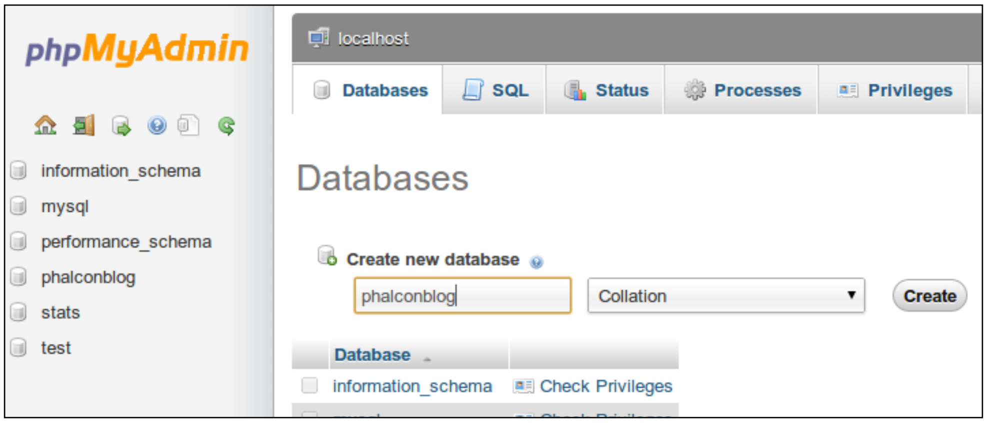
Creating a database

Before we can get Phalcon Developer Tools to generate any more files for us, we need a database. The project skeleton was the easy part. We now need to generate the model, controller, and view files. So, let's create our database. For the purpose of this book, I am going to give examples using raw SQL queries, so you can execute them via the command line or your chosen database tool, and examples using phpMyAdmin, which can be downloaded for free from `http://www.phpmyadmin.net`. This also requires that you have MySQL installed on your machine, which can be downloaded for free from `http://www.mysql.com`.

First we need our database. The following is the command to create the database we need for our project:

```
CREATE DATABASE phalconblog;
```

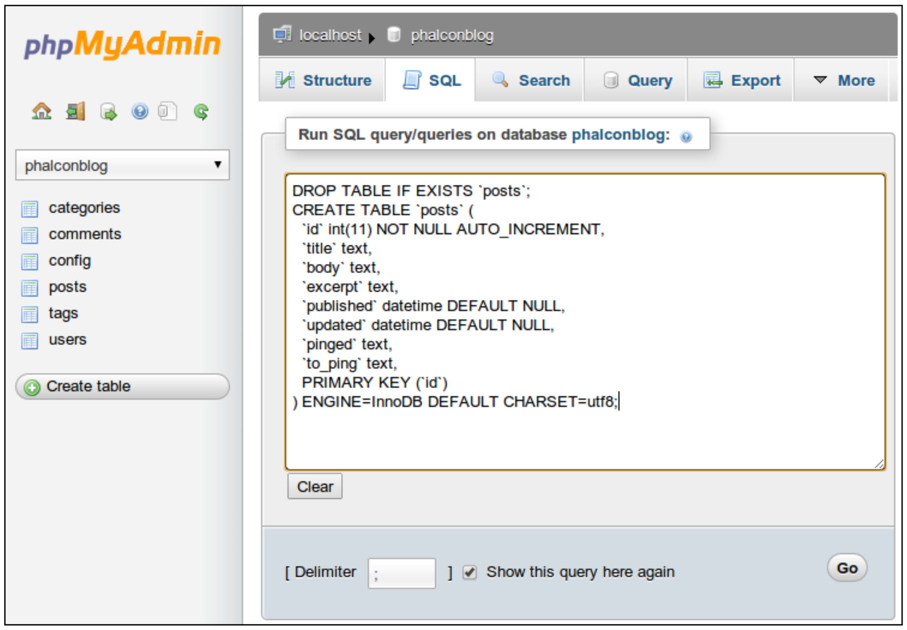
In phpMyAdmin, you could just click on the **Databases** tab, enter `phalconblog` in the **Create new database** section of the page, and click on the **Create** button, as shown in the following screenshot:



Now, we just need to create the table for our blog posts. We will worry about the other tables later. The following is the SQL code for our blog `posts` table:

```
DROP TABLE IF EXISTS `posts`;
CREATE TABLE `posts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` text,
  `body` text,
  `excerpt` text,
  `published` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  `pinged` text,
  `to_ping` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

If you are using phpMyAdmin, browse to your `phalconblog` database, click on the **SQL** tab, and paste the previous SQL code. Then click on the **Go** button and your table will be ready.



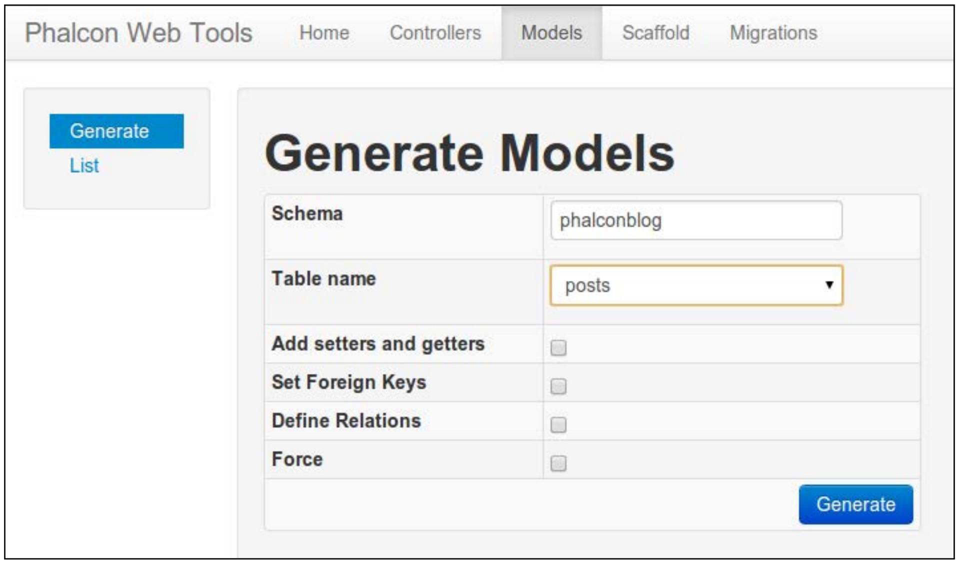
```
DROP TABLE IF EXISTS `posts`;
CREATE TABLE `posts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` text,
  `body` text,
  `excerpt` text,
  `published` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  `pinged` text,
  `to_ping` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We now have a simple table to hold our blog posts. It has a unique ID that autoincrements a title field for the title of our post, a body field to hold the body, an excerpt field to hold the summary of our post for pages such as our home page and category pages, a published date field, an updated date field, and two fields to hold the ping status of our post.

Now that we have a table in place, we can use Phalcon Developer Tools to create a model file for our blog posts.

Creating a model

Open `http://localhost/phalconBlog/webtools.php` in your browser if that is where your project is running. If not, browse to the index page of your project and add `/webtools.php` to the URL. The `webtools.php` file is located in the same folder as the bootstrap file. You should see a page in your browser that looks something like the following screenshot:



Generate Models

Schema	phalconblog
Table name	posts
Add setters and getters	☐
Set Foreign Keys	☐
Define Relations	☐
Force	☐

We are going to generate a model file for our `posts` table. Phalcon Developer Tools connect to our database using the setting we put in our `Ini` file and then load all the table names in the **Table name** select list. So, select the `posts` table from the select list and click on the **Generate** button. You also have the options to add setters and getters to your class, set foreign keys for related tables, and define relations. You can also force web tools to generate the model file even if it already exists.

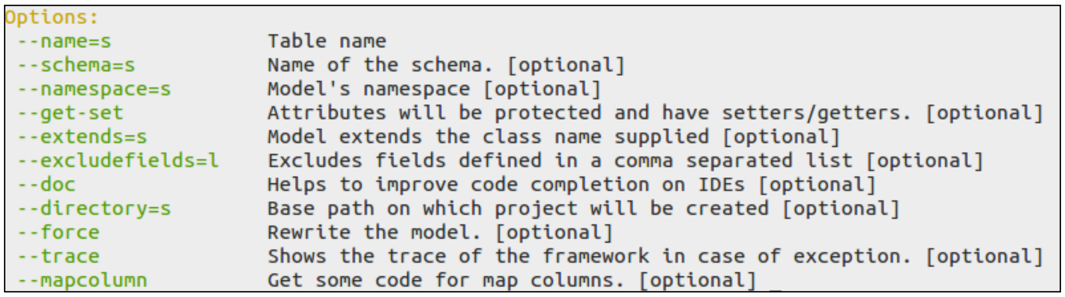
The other option is to use the command-line developer tools. You will need to open a terminal instance, navigate to your `phalconBlog` project folder, and type the following command:

```
phalcon model posts
```

Here, `posts` is the name of the table for which a model file is being generated. The command line in the Phalcon Developer Tools also has options. In fact, there are more options available here than on the web interface. You can list all these options by calling the command without a table name, as follows:

```
phalcon model
```

The previous command should print out options as shown in the following screenshot:



```
Options:
 --name=s             Table name
 --schema=s           Name of the schema. [optional]
 --namespace=s        Model's namespace [optional]
 --get-set            Attributes will be protected and have setters/getters. [optional]
 --extends=s          Model extends the class name supplied [optional]
 --excludefields=l    Excludes fields defined in a comma separated list [optional]
 --doc                Helps to improve code completion on IDEs [optional]
 --directory=s        Base path on which project will be created [optional]
 --force              Rewrite the model. [optional]
 --trace              Shows the trace of the framework in case of exception. [optional]
 --mapcolumn          Get some code for map columns. [optional] _
```

Any one of the commands will print out a similar list of options if called without any options.

Once this is done, we should find the `Posts.php` file in our `models` folder in `app`, and we will find the following code in it:

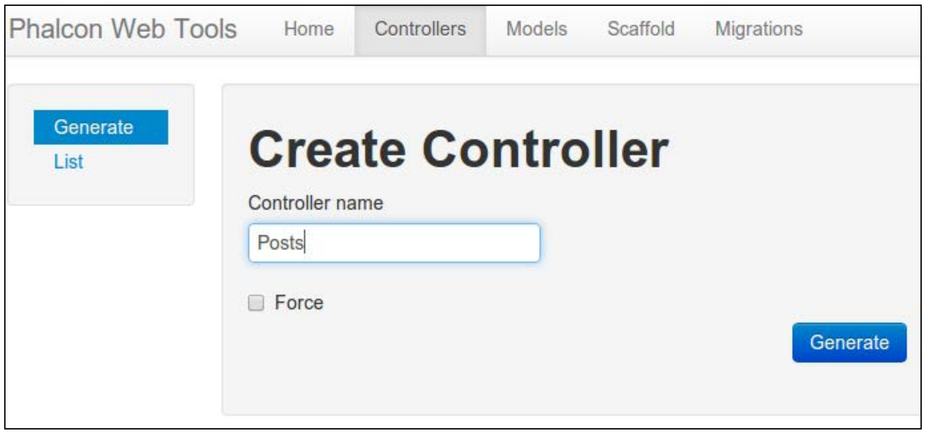
```
<?php
class Posts extends \Phalcon\Mvc\Model
{
    /**
     *
     * @var integer
     */
    public $id;
    /**
     *
     * @var string
     */
    public $title;
    /**
     *
     * @var string
     */
    public $body;
    /**
     *
     * @var string
     */
    public $excerpt;
    /**
     *
     * @var string
     */
    public $published;
    /**
```

```
         *
         * @var string
         */
        public $updated;
        /**
         *
         * @var string
         */
        public $pinged;
        /**
         *
         * @var string
         */
        public $to_ping;
    }
```

We went through all of that and really didn't get much, but it was better than typing out all these variables manually. Phalcon extracted the column names out of our `posts` table, created a `Posts model` class, and turned these names into member variables in our class. It's a start. We can also create a controller using web tools.

Creating a controller

You can also create your controller with web tools. Just click on the **Controllers** tab on the Phalcon web tools menu and then enter your controller name in the **Controller name** field of the form, which in our case is `Posts`, as shown in the following screenshot:



Controller name: Posts

As with the model we created, we can use the following command-line tools to create our controller:

```
phalcon controller posts
```

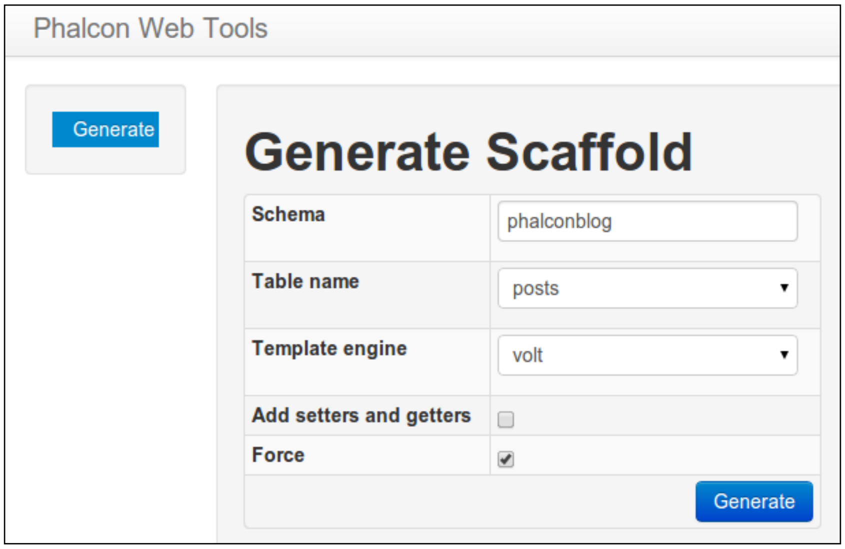
The result of this will be a new `PostsController.php` file located at `app/controllers` as shown in the following code:

```
<?php
class PostsController extends \Phalcon\Mvc\Controller
{
    public function indexAction()
    {

    }
```

There is not much to this file, just our `PostsController` class with one `indexAction` function. But things are about to get a lot more interesting. Let's build the CRUD functions for our Posts model.

Creating CRUD scaffolding

CRUD stands for create, read, update, and delete, which are the four basic functions our application should do with our blog post records. Phalcon web tools will also help us to get these built. Click on the **Scaffold** tab on the web tools page and you will see a page as shown in the following screenshot:

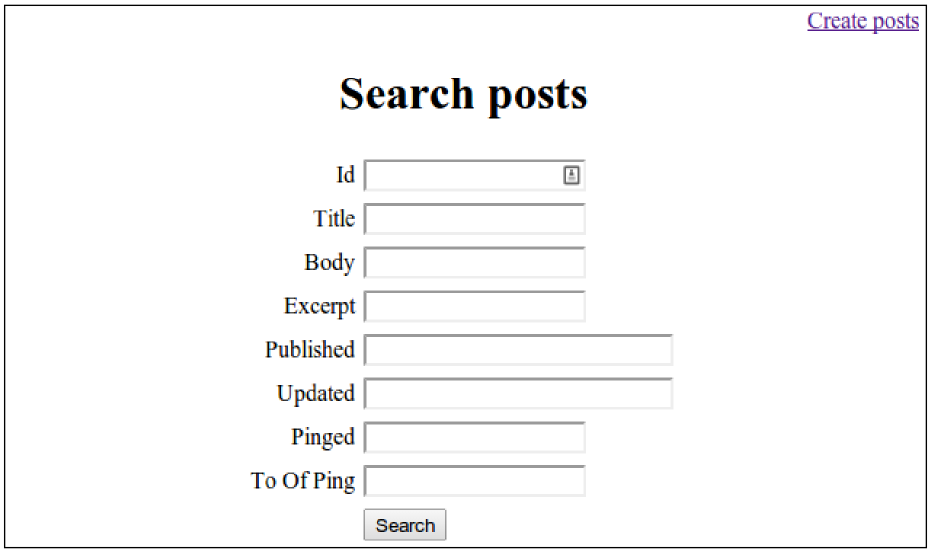


Generate Scaffold

Schema	phalconblog
Table name	posts
Template engine	volt
Add setters and getters	☐
Force	✓

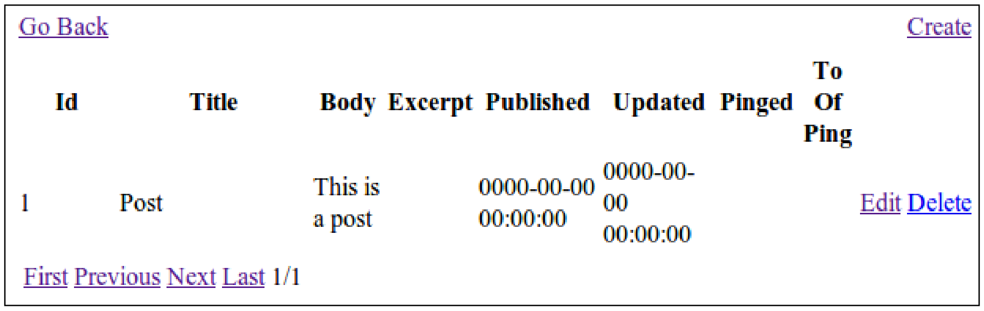
Select `posts` from the **Table name** list and `volt` from the **Template engine** list, and check **Force** this time, because we are going to force our new files to overwrite the old model and controller files that we just generated. Click on the **Generate** button and some magic should happen.

Browse to `http://localhost/phalconBlog/posts` and you will see a page like the following screenshot:



We finally have some functionality we can use. We have no posts, but we can create some. Click on the **Create posts** link and you will see a page similar to the one we were just at. The form will look nearly the same, but it will have a **Create posts** heading. Fill out the **Title**, **Body**, and **Excerpt** fields and click on the **Save** button. The form will post, and you will get a message stating that the post was created successfully.

This will take you back to the post's index page. Now you should be able to search for and find the post you just created. If you forgot what you posted, you can click on **Search** without entering anything in the fields, and you should see a page like the following screenshot:



Go Back | Create

Id	Title	Body	Excerpt	Published	Updated	Pinged	To Of Ping		
1	Post	This is a post		0000-00-00 00:00:00	0000-00-00 00:00:00			Edit	Delete

First Previous Next Last 1/1

This is not a very pretty or user-friendly blog application. But it got us started, and that's all we need. The next time we start a Phalcon project, it should only take a few minutes to go through these steps. Now we will look over our generated code, and as we do, modify it to make it more blog-like.

Examining our Posts model

If you open up `Posts.php` file located at `app/model`, you may notice that it has not changed since we generated it the first time. Even with our new CRUD scaffolding, it still just contains our Posts model class and our database column names as member variables. There is no SQL involved on the PHP side.

The `Phalcon\MVC\Model` path is the base class of all models that your application will use. All the SQL code that is involved happens inside of the Phalcon PHP extension. There will be times when we have to make SQL queries, but for now, we have given our Posts model CRUD capabilities, search capabilities, and the ability to relate to other models without even knowing what database software our application is going to use. We could modify our `services.php` file located at `app/config` to use another database adapter if we decide that we no longer want to use MySQL and want to use Oracle instead. The code for our application would work the same as long as our database tables are structured the same.

Our Posts model currently does everything we need it to do with minimum code, so we won't be modifying it in this chapter. Now let's look at our controller.

Examining our Posts controller

Now let's open `PostsController.php` file located at `app/controllers` in our editor. There is definitely a lot going on in this file. There are seven methods in our `PostsController` class suffixed with actions. Each of these actions handles requests and creates responses in our application. Each corresponds to a URI in our application: `http://localhost/phalconBlog/[model]/[action]`. When we browse to `http://www.localhost/phalconBlog/posts/`, by default, the `indexAction` function is called.

Let's take a look at each of these actions and modify them if we need to.

Create

Let's take a look at the `createAction` function, which saves our new blog posts for us.

First, we should learn a little about `Phalcon\Mvc\Dispatcher`. The dispatcher is another one of the services that was loaded in our Dependency Injection container. The dispatcher takes the request object and instantiates a controller based on the attributes of that object. When a controller is instantiated, it acts as a listener for dispatcher events. Since everything in the Dependency Injection container can be accessed as a property of the controller, we can access the dispatcher in a controller with `$this->dispatcher`.

Another object in our Dependency Injection container that we should know about is the request object, which stores details about the HTTP request and is passed between the dispatcher and the controller. The code is as follows:

```
public function createAction()
{

    if (!$this->request->isPost()) {
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));
    }

    $post = new Posts();

    $post->id = $this->request->getPost("id");
    $post->title = $this->request->getPost("title");
    $post->body = $this->request->getPost("body");
```

```
        $post->excerpt = $this->request->getPost("excerpt");
        $post->published = $this->request->getPost("published");
        $post->updated = $this->request->getPost("updated");
        $post->pinged = $this->request->getPost("pinged");
        $post->to_ping = $this->request->getPost("to_ping");


        if (!$post->save()) {
            foreach ($post->getMessages() as $message) {
                $this->flash->error($message);
            }
            return $this->dispatcher->forward(array(
                "controller" => "posts",
                "action" => "new"
            ));
        }

        $this->flash->success("post was created successfully");
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));

    }
```

First, our method checks if the request is a post by querying the request object from our dispatcher container. If the request is not a post, `$this->dispatcher->forward` is called, and our dispatcher forwards the requests to the `indexAction` function in our posts controller, where the dispatch loop starts up again.

If we are posting a new blog post, then the method continues executing and a post object is created. We can access the data we posted in our form with `$this->request->getPost()`, and our code assigns each parameter from this request object to an attribute of our `$post` instance. In many cases, it would be easier to replace this series of assignments with one line of code as follows:

```
    $post->save($_POST);
```

But we will be setting some of these variables in our controller later in the book, so we will leave them as they are. Since Phalcon is handling all the details about our database behind the scenes, we can just call `$this->post->save()` to save our post to the database; there is no need for SQL.

Next, the code checks if our post has been saved, and if it hasn't, it uses the flashing messages' service that is embedded in Phalcon to store the error message with `$this->flash->error`, while the application forwards us back to the form. If the post was successfully created, we are forwarded to the index with a successful message as `$this->flash->success`. The flashing messages' service provides our application with the ability to give users notification of the status of their actions. There are four message types in the flashing messages' services:

- Error
- Success
- Notice
- Warning

If we want to set custom classes for our messages when they are served by our view, we can add the following code to the `services.php` file in `config`:

```
$di->set('flash', function(){
    return new \Phalcon\Flash\Direct(array(
        'error' => 'alert alert-error',
        'success' => 'alert alert-success',
        'notice' => 'alert alert-info',
    ));
});
```

From then on, any error message served by our application will be given both the alert and alert-error classes. The `Phalcon\Flash\Direct` is a variant of `Phalcon\Flash`, which immediately outputs the messages passed to it. Another variant is `Phalcon\Flash\Session`, which temporarily stores the messages in the user's session to be output in the next request.

Search

Now let's take a look at our `searchAction` function:

```
public function searchAction()
{

    $numberPage = 1;
    if ($this->request->isPost()) {
        $query = Criteria::fromInput($this->di, "Posts", $_POST);
        $this->persistent->parameters = $query->getParams();
    } else {
        $numberPage = $this->request->getQuery("page", "int");
    }

    $parameters = $this->persistent->parameters;
    if (!is_array($parameters)) {
        $parameters = array();
    }
    $parameters["order"] = "id";

    $posts = Posts::find($parameters);
    if (count($posts) == 0) {
        $this->flash->notice("The search did not find any posts");
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));
    }

    $paginator = new Paginator(array(
        "data" => $posts,
        "limit"=> 10,
        "page" => $numberPage
    ));

    $this->view->page = $paginator->getPaginate();
}
```

At the beginning of our function, we set a `$numberPage` variable to `1`. This is because we use the Phalcon paginator that we loaded into our Phalcon services container in our bootstrap file. We will be using that variable to tell the paginator what page of results will be loaded, and if there is no `page` parameter, then we want to load the first page of results.

After that, our code checks if there is a POST request, and if there is, we use the `fromInput` function in `Phalcon\Mvc\Model\Criteria` to change the parameters in our post into an array that we can use to query our Posts model. We persist the parameters in the session by creating a variable parameter in `$this->persist`, which is a Phalcon class that persists variables between sessions. This enables our code to load a second page of results from a search query without adding parameters to the URL. The only parameter needed to load a second page of search results is `page`. If there is no post, we set the `$numberPage` variable to the `page` parameter from the GET request with `getQuery`.

Then, we set a local `$parameters` variable to our persistent parameters, and we will use this variable to query our Posts model using the `find` function, storing the results in `$posts`. Then, the code checks if the post has any results and sets a flash message to tell us when there are no posts in our `posts` table.

Then, we finally get to the paginator we mentioned at the beginning of this section. We create it by setting the `data` parameter to the `$posts` variable, setting the `limit` parameter to the count of records we want for each page, which in this case is `10`, and then set the `page` parameter to our `$numberPage` variable. Then, we set the `page` variable of our `view` object to the `page` object that our paginator returns, which holds the paginated blog post's results.

Index

Our `indexAction` function currently doesn't do much. The index view shows the user a search form. We are going to change that because we want it to show a list of our most recent posts, as shown in the following code:

```
<?php

use Phalcon\Mvc\Model\Criteria,
    Phalcon\Paginator\Adapter\Model as Paginator;

class PostsController extends ControllerBase
{

    /**
     * Index action
     */
    public function indexAction()
    {
        $this->persistent->parameters = null;
    }
```

We are going to change it to the following code:

```
public function indexAction()
    {
        $numberPage = $this->request->getQuery("page", "int", 1);

        $posts = Posts::find();
```

```
            $paginator = new Paginator(array(
                "data" => $posts,
                "limit"=> 10,
                "page" => $numberPage
            ));

            $this->view->page = $paginator->getPaginate();
        }
```

You will notice a difference in the `PostsController` class that we generated compared to the one we wrote from scratch at the beginning of the chapter. Phalcon Developer Tools generated a `ControllerBase` class for us so that we can add universal variables and functionality to it. We can then extend this class to create the controllers for our application.

Here, we are using part of the functionality we saw in the `searchAction` function. We set the page number for the paginator by using the `getQuery` function of `Phalcon\Http\Request`. The first `page` parameter is the GET variable we want, the second parameter is the type we expect this parameter to be, for example, an integer, and the third parameter is the default value, which we set to `1` for the first page. Then, we instantiate the paginator and set the `view page` variable to the results from our paginator. But we use the static `query` function from `Phalcon\MVC\Model` to execute our query in an object-oriented manner, ordering our posts by the published date (which we have not yet set, but we will do so). We will learn more about interacting with Phalcon models and databases in the next chapter.

New

There is nothing in the new action method, literally. The action is still called and the view that holds our new post form will be served:

```
    public function newAction()
    {

    }
```

Edit

The next method we will examine is our `editAction` function:

```
public function editAction($id)
{

    if (!$this->request->isPost()) {

        $post = Posts::findFirstByid($id);
        if (!$post) {
            $this->flash->error("post was not found");
            return $this->dispatcher->forward(array(
                "controller" => "posts",
                "action" => "index"
            ));
        }

        $this->view->id = $post->id;

        $this->tag->setDefault("id", $post->id);
        $this->tag->setDefault("title", $post->title);
        $this->tag->setDefault("body", $post->body);
        $this->tag->setDefault("excerpt", $post->excerpt);
        $this->tag->setDefault("published", $post->published);
        $this->tag->setDefault("updated", $post->updated);
        $this->tag->setDefault("pinged", $post->pinged);
        $this->tag->setDefault("to_ping", $post->to_ping);

    }
}
```

You will notice the `$id` parameter in this function. Any additional URI parameters after the parameter for the action are action parameters. They are assigned in the same order that the route passes them in. First, our action checks if the request was a post. If not, it skips the rest of the functionality and loads the view. If it is a post, we create an instance of our Posts object and use its `findFirstByid` method to load our blog post into the `$post` variable. We check if our `$post` variable contains data and then set our view's ID variable to the ID of the blog post. If we want, we can also replace these multiple assignments that Phalcon Developer Tools generated for us with the following loop.

The `Phalcon\Tag` is a view helper class that will generate HTML for our application. By calling `$this->tag->setDefault` in our action, we can set the default value of a form element with the attribute of the same name. So, by calling `$this->tag->setDefault("id", $post->id)`, we set the default ID form input to the ID of our post instance.

Save

Now, let's check out our `saveAction` function using the following code:

```
public function saveAction()
{

    if (!$this->request->isPost()) {
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));
    }

    $id = $this->request->getPost("id");

    $post = Posts::findFirstByid($id);
    if (!$post) {
        $this->flash->error("post does not exist " . $id);
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));
    }

    $post->id = $this->request->getPost("id");
    $post->title = $this->request->getPost("title");
    $post->body = $this->request->getPost("body");
    $post->excerpt = $this->request->getPost("excerpt");
    $post->published = $this->request->getPost("published");
    $post->updated = $this->request->getPost("updated");
    $post->pinged = $this->request->getPost("pinged");
    $post->to_ping = $this->request->getPost("to_ping");

        if (!$post->save()) {
            foreach ($post->getMessages() as $message) {
                $this->flash->error($message);
            }
            $this->flash->error("post was not saved");
            $this->view->pick('posts/edit');
            $this->view->post = $post;
        } else {
            $this->flash->success("post was updated successfully");
            return $this->dispatcher->forward(
                array(
                    "controller" => "posts",
                    "action" => "index"
                )
            );
        }
}
```

This method is very similar to the create method. This one is called when we are saving a post that we are editing. This is another place where, if we wish, we can replace the multiple lines of assignments with a loop, like we did in the `createAction` function. The only real difference in this method is that when a post is not saved successfully, we pick the view that we want to show using `$this->view->pick()`, and then we set the `$post` variable in the view to the one just created. This way, the content is still saved in the browser if the new post fails to save. If we were to simply forward it to the `editAction` function, it would again look up the post by its `id` and wipe out our changes. We use the same view, we just bypass the action.

Delete

The following function is the `deleteAction` function in our controller:

```
public function deleteAction($id)
{

    $post = Posts::findFirstByid($id);
    if (!$post) {
        $this->flash->error("post was not found");
        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "index"
        ));
    }

    if (!$post->delete()) {

        foreach ($post->getMessages() as $message){
            $this->flash->error($message);
        }

        return $this->dispatcher->forward(array(
            "controller" => "posts",
            "action" => "search"
        ));
    }

    $this->flash->success("post was deleted successfully");
    return $this->dispatcher->forward(array(
        "controller" => "posts",
        "action" => "index"
    ));
}
```

There's nothing fancy happening here. The method tries to delete the post by calling `$post->delete`, and gives us an error message if it wasn't deleted and a success message if it was.

Show

We are going to have to add an action to our controller to view each post by itself. This will be easy because our `editAction` function does some of the same things that we need our `showAction` function to do. We need our controller to pass a post instance to our view based on a post ID. The following is our code:

```
public function showAction($id)
    {

        if (!$this->request->isPost()) {

            $post = Posts::findFirstByid($id);
            if (!$post) {
                $this->flashSession->error("post was not found");
                $response = new \Phalcon\Http\Response();
                $response->setStatusCode(404, "Not Found");
                $response->redirect("posts/index");
            }

        }

        $this->view->post = $post;
    }
```

There is not really much to this method. It takes in our post ID and checks if our post is valid. If it is, we set the `view` variable's post to our `$post` object so we can use it in our view. But if there is no post, we want to at least set a 404 status code. So, we create a new instance of `Phalcon\Http\Response` and set the status code, and with that same response, we are redirected back to the index of our posts. Notice that instead of using `$this->flash`, we are using `$this->flashSession`, because we are using a true HTTP redirect rather than an internal forward, so we need to store the message in the user's session.

Now that our controller is doing everything we want it to do, let's take a look at our views.

Examining our Post views

Well, we didn't do any manual edits on our model, and we only made a few changes to our controller. But we are going to make a lot of changes to our views. They need a lot of work. So, we are not going to compare much of our generated code with our final code.

Fortunately, when we installed Phalcon web tools in our application, it installed Twitter Bootstrap for us. And as it is already there, we are going to use it. As this is a book on the Phalcon framework and not Twitter Bootstrap, I won't be explaining much of what we are doing with it. We will just use it. Twitter Bootstrap is a frontend framework that makes styling easier. You can read more about it at `http://getbootstrap.com/`.

We know from *Chapter 2*, *Setting Up a Phalcon Project*, that every request in our application uses the `index.volt` file located at `app/views` to wrap the content returned from our views. If we want to add JavaScript or CSS globally to our application, this would be where we would want to include it. The code is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Phalcon Blog</title>
    <link rel="stylesheet" href="/phalconBlog/css/bootstrap/
      bootstrap.min.css" type="text/css" />
        <link rel="stylesheet" href="/phalconBlog/css/bootstrap/
          bootstrap-responsive.min.css" type="text/css" />
  </head>
  <body>
      <div class="navbar">
            <div class="navbar-inner">
                <div class="container">
                    <a data-target=".nav-collapse"
                      data-toggle="collapse" class="btn btn-navbar">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </a>
                    <a href="/phalconBlog" class="brand">Phalcon
                      Blog</a>
                    <div class="nav-collapse">
                        <ul class="nav">
                            <li>{{ link_to("posts/", "Posts") }}</li>
                            <li>{{ link_to("posts/search", "Advanced
                              Search") }}</li>
                            <li>{{ link_to("posts/new", "Create
                              posts") }}</li>
                            <li><a href="/phalconBlog/webtools.php?
                              _url=/index">Webtools</a></li>
                        </ul>
                    </div>
                </div>
            </div>
        </div>
        <div id="content" class="container-fluid">
            <div class="row-fluid">
                <div class="span3">
                    <div class="well">
                        {{ form("posts/search", "method":"post",
                          "autocomplete" : "off", "class" :
                          "form-inline") }}
                            <div class="input-append">
                                {{ text_field("body", "class" :
                                  "input-medium") }}
                                {{ submit_button("Search", "class" :
                                  "btn") }}
                            </div>
                       {{ end_form() }}
                    </div>
                </div>
            <div class="span9 well">
            {{ content() }}
            </div>
            </div>
        </div>
    <script src="/phalconBlog/js/jquery/jquery.min.js" type="text/
      javascript"></script>
        <script src="/phalconBlog/js/bootstrap/bootstrap.min.js"
          type="text/javascript"></script>
  </body>
</html>
```

Well, we added quite a bit of markup to support Twitter Bootstrap. We included the `css` files that we need in order to use Bootstrap in the header and the JavaScript in the footer. In the `navbar` div, we have links to the main parts of our site. We have a search form in the sidebar that currently searches only the body field, but we will be changing that later. Then, we output our content. We now have a responsive layout for our blog using Phalcon's own built-in Volt template engine.

Volt uses {{…}} to print variables to the template, and {%...%} to assign variables or execute loops. The variables and function calls in Volt are very similar to the underlying PHP code, only without dollar signs. In the main part of our page, we use `content()`, which is a template tag for `$this->getContent()`. This tag is used here in the main layout. But you will also notice it in the controller layout as well as in the action views. The output of each view is served by the next one in the hierarchy using this variable.

There are three levels in this view hierarchy. The action view only renders when the action of a specific controller is called. The controller layout view will be shown for every action in the controller. And finally, the main layout view will be shown for every controller and action in the application.

For the links, we use the `link_to()` tag, which represents the `linkTo` view helper function of `Phalcon\Tag`. The first parameter is the path in our application, and the second is the anchor text we will use for the link.

For the search form, we use the `form()` template tag, which builds the first HTML form tag for our form thanks to Phalcon's view helpers. To close the form, we use Volt's `end_form()` template tag. The first parameter we pass to this tag is the action attribute. The second parameter is the set of key-value pairs that translate into attributes in the HTML form tag. We add the `input-medium` class to our field to apply Twitter Bootstrap styles.

For the search form field, we again use another Volt tag that represents a Phalcon view helper. The `text_field()` tag accepts the name of our field as its first parameter. And the second parameter is again a key-value pair that translates to HTML tag attributes. We add the `btn` class to our submit button.

Posts layout

The layout that wraps the results of every action in the Posts controller is loacted in the `posts.volt` file at `app/views/layouts`. There is not much in this file. If we wanted to add markup to every view that our Posts model uses, we would do it here. We are going to remove the `align="center"` attribute from the `div`, so it looks like the following code:

```
<div>
    {{ content() }}
</div>
```

Index action view

The view for index action is in the `index.volt` file located at `app/views/posts`. This page originally held a search form. But we changed the `indexAction` function in our controller to list the most recent blog posts. We are going to have to make a lot of changes to this file, which will actually result in less overall markup as follows:

```
{{ content() }}

{% if page.items is defined %}
{% for post in page.items %}
```

```
        <article>
            <h2>{{ post.title }}</h2>
            <div>{{ post.excerpt }}</div>
            <div>{{ link_to("posts/show/"~post.id, "Show") }}</div>
        </article>
    {% endfor %}
    <ul class="pager">
        <li>{{ link_to("posts/index", "First") }}</li>
        <li>{{ link_to("posts/index?page="~page.before, "Previous")
          }}</li>
        <li>{{ link_to("posts/index?page="~page.next, "Next") }}</li>
        <li>{{ link_to("posts/index?page="~page.last, "Last") }}</li>
        <li>{{ page.current~"/"~page.total_pages }}</li>
    </ul>
    {% endif %}
```

In Volt, we use {%...%} to represent loops and control structures. In this view, we wrap everything in an `if` tag, which checks if our `page` object generated by the paginator in our controller contains any items. The results in the `page` object are stored in this `item` variable. If we have items, we loop through them, setting the `post` object to what we find. We can access the variables in this `post` object using the dot syntax. To print out the post excerpt, we use `post.excerpt`.

And finally, we have the paginator part of our paginator. Along with the items, our paginator object contains details about the results. We can use `page.before` to access the page number of the current page, `page.next` for the next page number, and `past.last` for the last page. At the end of the links, we print the current page number and the total number of pages.

Search action view

Now, we will take a look at the view for the search action in the `search.volt` file located at `app/views/posts`. The code is as follows:

```
    {{ form("posts/search", "method":"post") }}
    {{ content() }}

    <div align="center">
        <h1>Search Posts</h1>
    </div>

    <table align="center">
        <tr>
            <td align="right">
                <label for="id">Id</label>
            </td>
            <td align="left">
                {{ text_field("id") }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="title">Title</label>
            </td>
            <td align="left">
                    {{ text_field("title") }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="body">Body</label>
            </td>
            <td align="left">
                    {{ text_field("body") }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="excerpt">Excerpt</label>
            </td>
            <td align="left">
                    {{ text_field("excerpt") }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="published">Published</label>
            </td>
            <td align="left">
                {{ text_field("published", "size" : 30) }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="updated">Updated</label>
            </td>
            <td align="left">
                {{ text_field("updated", "size" : 30) }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="pinged">Pinged</label>
            </td>
            <td align="left">
                    {{ text_field("pinged", "type" : "date") }}
            </td>
        </tr>
        <tr>
            <td align="right">
                <label for="to_ping">To Of Ping</label>
            </td>
            <td align="left">
                    {{ text_field("to_ping", "type" : "date") }}
            </td>
        </tr>

        <tr>
            <td></td>
            <td>{{ submit_button("Search", "class" : "btn") }}</td>
        </tr>
    </table>

    {{ end_form() }}

    {% if page.items is defined %}
    {% for post in page.items %}
        <article>
            <h2>{{ post.title }}</h2>
            <div>{{ post.excerpt }}</div>
            <div>{{ link_to("posts/show/"~post.id, "Show") }}</div>
        </article>
    {% endfor %}
    <ul class="pager">
        <li>{{ link_to("posts/search", "First") }}</li>
        <li>{{ link_to("posts/search?page="~page.before, "Previous") }}</
    li>
        <li>{{ link_to("posts/search?page="~page.next, "Next") }}</li>
        <li>{{ link_to("posts/search?page="~page.last, "Last") }}</li>
        <li>{{ page.current~"/"~page.total_pages }}</li>
    </ul>
    {% else %}
    {{ form("posts/search", "method":"post", "autocomplete" : "off") }}
    {% endif %}
```

Here, we have much of what we have seen in the other views. The form has more fields, so that each field in our posts can be searched. Then, we have a loop to print out the results of our searches, similar to the index action view, and at the end, we have our pager to browse the results.

Edit action view

Next, we have the view for the edit action in the `edit.volt` file located at `app/views/posts`.

```
{{ form("posts/edit", "method":"post") }}

{{ content() }}

<div align="center">
    <h1>Edit Post</h1>
</div>

<table>
    <tr>
        <td align="right">
            <label for="title">Title</label>
        </td>
        <td align="left">
                {{ text_field("title") }}
        </td>
    </tr>
    <tr>
        <td align="right">
            <label for="body">Body</label>
        </td>
        <td align="left">
                {{ text_area("body") }}
        </td>
    </tr>
    <tr>
        <td align="right">
            <label for="excerpt">Excerpt</label>
        </td>
        <td align="left">
                {{ text_area("excerpt") }}
        </td>
    </tr>
    <tr>
        <td>{{ hidden_field("id") }}</td>
        <td>{{ submit_button("Save", "class" : "btn") }}</td>
    </tr>
</table>

</form>
```

Here, we again have the `content` tag and `form` tags for all the fields that we need to edit in our post. And at the end, we have our submit button; everything that we will need to edit a post.

New action view

Last but not least, the view for the new action in the `new.volt` file located at `app/views/posts` which uses the same template tags we covered in the other action views. The code for our view is as follows:

```
{{ form("posts/create", "method":"post") }}

{{ content() }}

<div align="center">
    <h1>Create Post</h1>
</div>

<table>
    <tr>
        <td align="right">
            <label for="title">Title</label>
        </td>
        <td align="left">
                {{ text_field("title", "type" : "date") }}
        </td>
    </tr>
    <tr>
        <td align="right">
            <label for="body">Body</label>
        </td>
        <td align="left">
                {{ text_area("body", "type" : "date") }}
        </td>
    </tr>
    <tr>
        <td align="right">
            <label for="excerpt">Excerpt</label>
        </td>
        <td align="left">
                {{ text_area("excerpt", "type" : "date") }}
        </td>
    </tr>
    <tr>
        <td></td>
        <td>{{ submit_button("Save", "class" : "btn") }}</td>
    </tr>
</table>

</form>
```

Summary

In this chapter, we worked on the model, view, and controller for the posts in our blog. To do this, we used Phalcon web tools to generate our CRUD scaffolding for us. Then, we modified this generated code so it would do what we need it to do. We can now add posts. We also learned about the Volt template engine.

In the next chapter, we will create more models and take a look at the various ways Phalcon handles data, including session data, relationships between models, filtering and sanitizing data, PHQL, and Phalcon's Object-Document Mapper.

4
Handling Data in Phalcon

Unless you are building an application that retrieves all of its data via APIs, you are most likely going to need a database in the application. Phalcon gives you a few ways to access data from PHP. In the previous chapter, we didn't interact directly with our MySQL database. We could create blog posts and store them in our database, but Phalcon models did all of the dirty work for us. All we did was add the database connection details to our configuration file.

There are a few more features our models can handle before we have to start using more of the available Phalcon features. But even the simplest application will grow to the point where unique database queries come into play. However, databases are not the only way to handle data in your application.

In this chapter, you will learn the following topics:

- How to use the more advanced features of Phalcon models
- How to store session data in a Phalcon application
- How to filter and sanitize our data
- How to replace our MySQL database with another database system
- How to access databases with Phalcon Query Language (PHQL)
- How to use Phalcon's Object-Document Mapper (ODM)
- How to handle database migrations

Adding models

Currently, our blog application doesn't do much. We can create and view posts, but that's about it. We need to think about what other tables we need in our blog.

Creating the database tables

We will have users, and each user will be able to write multiple posts. So, we need a `users` table, and we need to add a `users_id` field to our `posts` table so that each post is related to a user. The following is the SQL code for our `users` table:

```
CREATE TABLE IF NOT EXISTS `phalconblog`.`users` (
  `id` INT(11) NOT NULL AUTO_INCREMENT ,
  `username` VARCHAR(16) NOT NULL ,
  `password` VARCHAR(255) NOT NULL ,
  `name` VARCHAR(255) NOT NULL ,
  `email` TEXT NOT NULL ,
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`username`)
);
```

Run this SQL snippet on your MySQL database. If you need help using `phpMyAdmin`, refer to *Chapter 3*, *Using Phalcon Models, Views, and Controllers*, for how to execute this code. Now, you have a table where you can store a minimal amount of data about your users, including their chosen username and password, their name, and their e-mail address. We create a unique key on the `username` column because we don't want two users to have the same username.

Also, a blog is not really a blog without comments. We want visitors to be able to comment on our blog applications. The following is the SQL code that we need to run on our database to create a home for our comments:

```
CREATE TABLE IF NOT EXISTS `comments` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `post_id` int(11) NOT NULL,
  `body` text NOT NULL,
  `name` text NOT NULL,
  `email` text NOT NULL,
  `url` text NOT NULL,
  `submitted` datetime NOT NULL,
  `publish` tinyint(1) NOT NULL,
  `posts_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_comments_posts1` (`posts_id`)
);
```

We need to relate each comment to a specific post, so we have a `post_id` field. To store the commenter's details, we have `name`, `email`, and `url`. We have a `submitted` date so that we can sort our comments. And, we have a `publish` field that we will set to `0` when a comment is submitted, and `1` when we approve the comment.

We might possibly need to organize our posts. We have a few options for doing this. We could use categories, where each post must fit into a specific category. Our categories can have child categories, or we can use tags, where each tag can be freely applied to any post, and each post can have more than one tag. Or we can do both, like most blogging applications. For our application, we are going to use tags only for simplicity. Run the following SQL code to create the `tags` table:

```
CREATE TABLE IF NOT EXISTS `tags` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `tag` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `tag` (`tag`)
);
```

There is not much to this table, just an `id` and `tag` column. That's because we have to use an intermediate table to relate our `posts` table to our `tags` table due to the many-to-many relationship between posts and tags. However, we do want each tag to be unique, so we make the `tag` column a unique key. We are going to call this table `post_tags`, and the following is the SQL code you need to run to create it:

```
CREATE TABLE IF NOT EXISTS `post_tags` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `posts_id` int(11) NOT NULL,
  `tags_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_post_tags_tags1` (`tags_id`),
  KEY `fk_post_tags_posts1` (`posts_id`)
);
```

This is just a table of IDs. A record in this table will have its own unique ID, a post ID, and a tag ID.

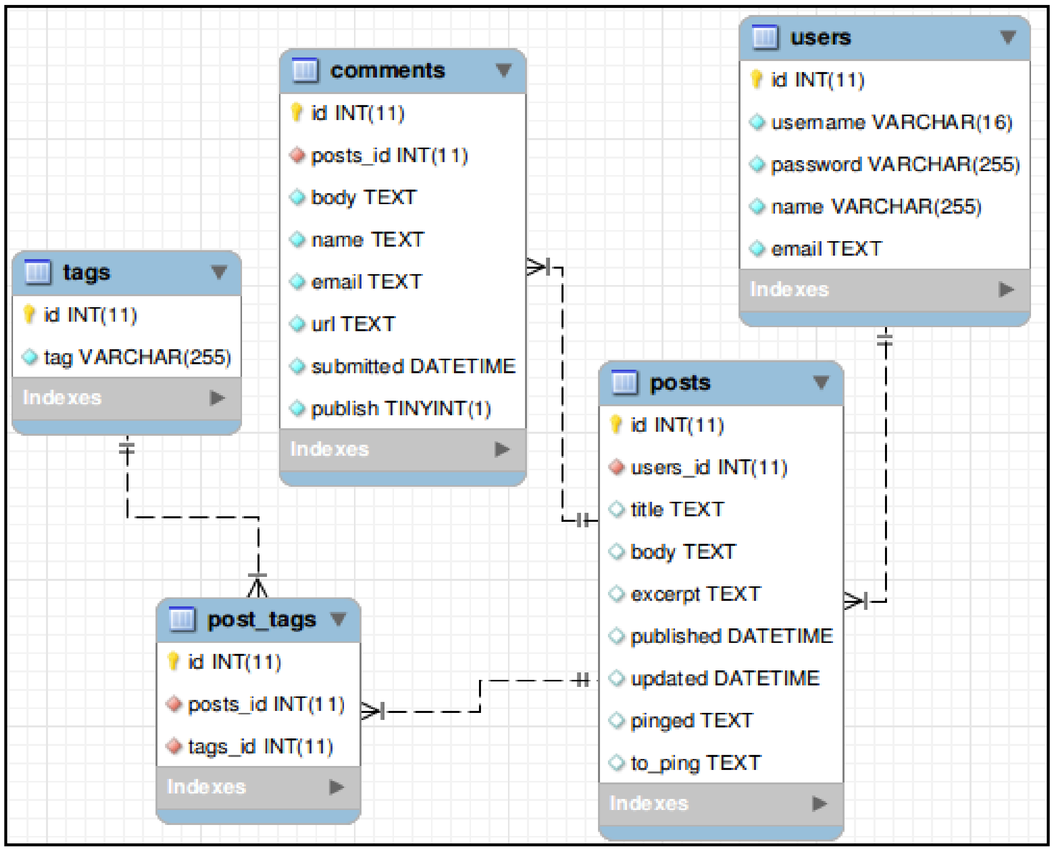
Now, we need to modify our `posts` table. It needs a user ID. The following is the SQL code we need to run to delete and recreate our `posts` table:

```
DROP TABLE `posts`;
CREATE TABLE IF NOT EXISTS `posts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `users_id` int(11) NOT NULL,
  `title` text,
  `body` text,
  `excerpt` text,
  `published` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  `pinged` text,
  `to_ping` text,
  PRIMARY KEY (`id`),
  KEY `fk_posts_users` (`users_id`)
);
```

The last step in our process is to create the foreign key relationships between our tables. Run the following SQL code to add the foreign key constraints to our database:

```
ALTER TABLE `comments`
  ADD CONSTRAINT `fk_comments_posts1` FOREIGN KEY (`posts_id`)
REFERENCES `posts` (`id`);
ALTER TABLE `posts`
  ADD CONSTRAINT `fk_posts_users` FOREIGN KEY (`users_id`) REFERENCES
`users` (`id`);
ALTER TABLE `post_tags`
  ADD CONSTRAINT `fk_post_tags_tags1` FOREIGN KEY (`tags_id`)
REFERENCES `tags` (`id`),
  ADD CONSTRAINT `fk_post_tags_posts1` FOREIGN KEY (`posts_id`)
REFERENCES `posts` (`id`);
```

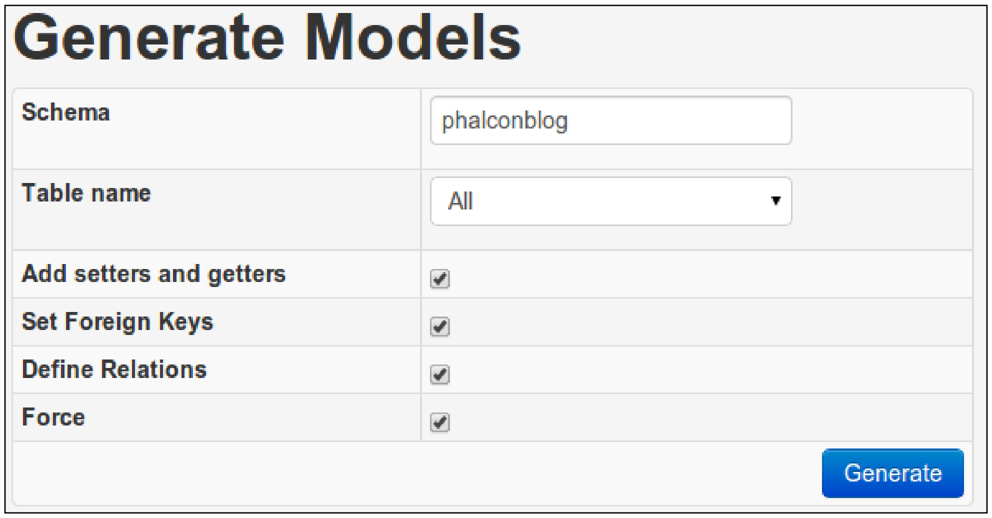
The following screenshot represents a diagram showing the relationships in our database:



The preceding screenshot was made from the EER diagram tool in MySQL Workbench, which you can download for free at `http://www.mysql.com/products/workbench/`. This tool is really handy for designing databases. I created all the tables and relationships for this blog with drag-and-drop and fill in the blanks.

Generating the models

Now, let's use Phalcon web tools to create all of our new models at once. Browse to `http://localhost/phalconBlog/webtools.php?_url=/models/index` and you will see the following screen:



Generate Models

Schema	phalconblog
Table name	All
Add setters and getters	✓
Set Foreign Keys	✓
Define Relations	✓
Force	✓

This time, we are going to select the **Table name** as **All** and make sure we check **Set Foreign Keys**, **Define Relations**, and **Force**. This way, web tools will set up the necessary relationships in our models. This will create the following files in our `models` folder located at `app`:

- `Comments.php`
- `PostTags.php`
- `Tags.php`
- `Users.php`

We will take a look at these files as we continue to refactor our blog project throughout the rest of this chapter. But, to understand a little bit about what we just did, let's take a look at how relationships are handled by Phalcon. The Phalcon model's manager handles the relationships between models, and these relationships must be set up in the `initialize` function of these models. By having Phalcon web tools define relationships, we can simply have it generate the `initialize` function for us.

In the `initialize` function of the `Users.php` file, you will find the following line of code:

```
$this->hasMany("id", "Posts", "users_id");
```

This line of code sets up the relationships between the posts and the users in our application. A user will have many posts. This is one of four methods. The other three methods are as follows:

- `hasOne()`
- `belongsTo()`
- `hasManyToMany()`

The first parameter is the field of the local model that is a key in the relationship. The second parameter is the model being referenced. The third parameter is the remote model's key. And since posts belong to users, we will find the following line of code in the `initialize` function of the Posts model:

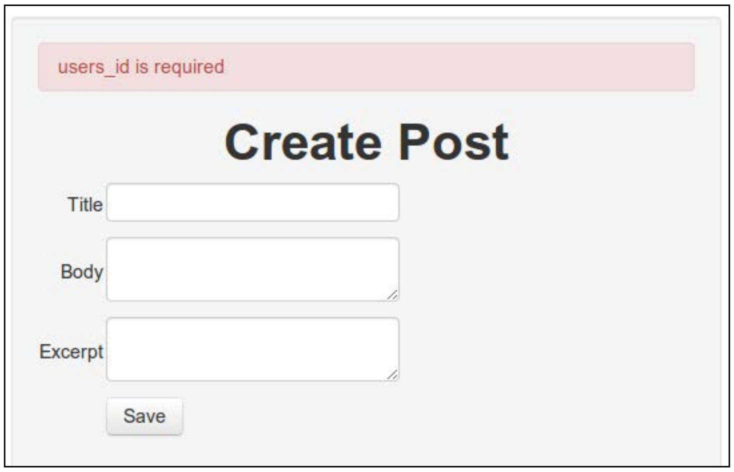
```
$this->belongsTo("users_id", "Users", "id");
```

For the most part, you can forget about the relationships and foreign keys in your database and just treat your records as objects.

Storing session data in Phalcon

First, we are going to give ourselves the ability to log in to our blog and store our user information in our session. We are going to wait to encrypt our passwords and set up permissions for our user until the next chapter. For now, we only want to be able to save a user ID when a user saves a blog post.

We made our changes to the model, but that doesn't mean everything is going to work. Browse to `http://localhost/phalconBlog/posts/create` and try to create a new post. You will not be able to. Instead, you will see the following error message in your browser because we just created a rule that our database requires every post to have a user ID:

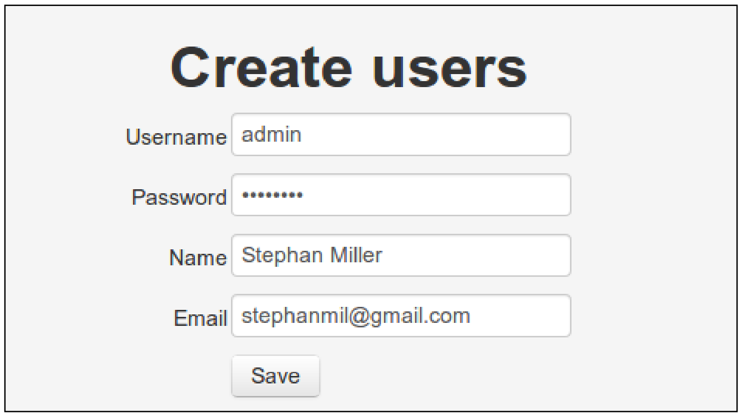

users_id is required

Create Post

Title	
Body	
Excerpt	

Another problem is that in order to have a user ID, we need a user in the `users` table to reference. Therefore, we need to be able to add users to our database with our application. So, go back to `http://localhost/phalconBlog/webtools.php?_url=/scaffold` and generate the scaffolding for your `users` table. We are going to cut a lot out of it and change some things, but again, it is a good place to start.

Once you have generated the scaffolding for your `users` table, browse to `http://localhost/phalconBlog/users/new` and create your user as shown in the following screenshot:


Create users

Username	admin
Password	••••••••
Name	Stephan Miller
Email	stephanmil@gmail.com

Now that we have a user, let's make sure that when we create a new blog post, the user ID gets related to it. We are going to set a `user_id` variable in the user's session. First, we need a form to log in, and we might as well put this in the `indexAction` function's template so we can log in there. Open up the `index.volt` file located `at app/views/users` that you generated with Phalcon web tools and modify it to look like the following code snippet:

```
{{ content() }}

{{ form("users/login", "method":"post", "autocomplete" : "off") }}

<h1>Users</h1>

<div>
    <label for="username">Username</label>
    {{ text_field("username", "size" : 30) }}
</div>

<div>
    <label for="password">Password</label>
    {{ password_field("password", "size" : 30) }}
</div>

{{ submit_button("Login", "class" : "btn") }}

{{ end_form() }}
```

What we have left is a `username` and `password` field and a **submit** button. Also, we have modified the original search form that web tools built for us to post to an action that doesn't exist yet in our controller, that is, the `loginAction` function.

Now, let's create a `loginAction` function in the `UsersController.php` file located at `app/controllers`. The method we will be adding is the following code snippet:

```
public function loginAction() {

    if ($this->request->isPost() && isset($_POST('username') &&
      isset($_POST('password')) {
        $query = Criteria::fromInput($this->di, "Users", $_POST);
        $this->persistent->parameters = $query->getParams();
        $parameters = $this->persistent->parameters;
        if (!is_array($parameters)) {
            $parameters = array();
        }
```

```
            $users = Users::findFirst($parameters);
            if (count($users) == 1) {
                $user = $users->getFirst();
                $this->session->set("user_id", $user->id);
                $this->flash->success("Welcome " . $user->name);
            } else {
                $this->flash->error("Username and Password combination not
                  found");
            }
        }
        return $this->dispatcher->forward(
            array(
                "controller" => "posts",
                "action" => "index"
            )
        );
    }
```

This method is pretty simple. It checks that there is a `POST` variable, and if so, sets `$parameters` to the `POST` variable. Then, we set the `$users` variable to the result of the Phalcon model's `Find` method. Since we made the `username` column a unique key in our database, we should either get one or no records. If we get no records, we set a flash message to tell the user that what they entered was incorrect. If we get one record, we call `$users->getFirst()` to get the first and only result from our query. And then, we set the session variable `user_id` to the `id` instance in our `$user` object. We started the session when our bootstrap file loaded the `services.php` file located at `app/config`, so it is available here in this controller.

While we have this file open, we should also create an action to log out. This method is even easier to write. We just call `$this->session->remove($var)`, where `$var` is the variable we want to remove from the session. Here is what the code looks like:

```
    public function logoutAction(){
            $this->session->remove("user_id");
            $this->flash->success("You have been logged out");
            return $this->dispatcher->forward(
                array(
                    "controller" => "users",
                    "action" => "index"
                )
            );
        }
```

We are going to make one last change to this file. We have already set the `indexAction` function's view to display the login form when the `indexAction` function is called. But, if a user is already logged in, let's show them a list of users. To do this, we will change the `indexAction` function so that it looks like the following code snippet:

```
public function indexAction() {
        $this->persistent->parameters = null;
        if ($this->session->has("user_id")) {
            return $this->dispatcher->forward(
                array(
                    "controller" => "users",
                    "action" => "search"
                )
            );
        }
    }
```

Here, we simply check if there is a user ID in the session, and if so, we forward the user to the search controller, which displays a list of all users when no search parameters are posted.

We still can't create a post yet until we set this user ID in it. So, open up the `PostsController.php` file located at `app/controllers`. We only have to worry about setting this ID when the blog post is created, so we are only going to modify the `createAction` function. Right before we instantiate the new post with `$post = new Posts()`, we are going to add the following code snippet:

```
if (!$this->session->has("user_id")) {
            $this->flash->error("Please log in to create a post");
            return $this->dispatcher->forward(
                array(
                    "controller" => "users",
                    "action" => "index"
                )
            );
        }
```

To check whether the session has a user ID in it or not, forward the user to the login page. If there is an ID, the code creates the new post, and we need to add the following line of code to add our user ID to the `post` object:

```
$post->users_id = $this->session->get("user_id");
```

And finally, we need a way to log out. We have created the `action` method already, but there is no way to get to it. So, open up the `search.volt` file located at `app/views/users` and add the following code snippet right after the `{{ content() }}` tag:

```
{{ link_to("users/logout/", "Logout", "class" : "btn") }}
```

This will add a button to the top of the list of users that will log us out. Now, we can log in and create a post without any issues.

Filtering and sanitizing data

To prevent unauthorized access, SQL injection, and other malicious attacks on our application, we need to filter and sanitize user input. We can use the `Phalcon\Filter` component to do this for us. This component supplies wrappers for the PHP filter extension.

Securing your application is beyond the scope of this book, but we can take a look at a filter that Phalcon development tools already added for us when we named one of our user field's e-mail. In the `UsersControllers.php` file located at `app/controller`, you will find that the `createAction` function has the following line of code:

```
$user->email = $this->request->getPost("email", "email");
```

The filter object is accessed through the Phalcon request object. The first parameter is the name of the variable we are accessing, and the second optional parameter is our filter. Phalcon has the following built-in filters:

Name	**Description**
`string`	Strips tags
`email`	Removes all characters except letters, digits, and special characters such as !, #, $, %, &, *, +, -, /, =, ?, ^, _, ', {, |, }, ~, @, and []
`int`	Removes all characters except digits, dots, plus, and minus
`float`	Removes all characters except digits, dots, plus, and minus
`alphanum`	Removes all characters except a-z, A-Z, and 0-9
`striptags`	Applies the `strip_tags` function
`trim`	Applies the `trim` function
`lower`	Applies the `strtolower` function
`upper`	Applies the `strtoupper` function

You can also write your own filters. You can read more about doing that at `http://docs.phalconphp.com/en/latest/reference/filter.html#creating-your-own-filters`.

Phalcon models revisited

Another thing we need to fix in our application is the fact that our dates don't mean anything. We have published and updated dates in our `posts` table, which we are not using. Let's make sure we're setting those dates. Let's open up the `Posts.php` file located at `app/model`. We are going to add more code to the `initialize` function of our Posts model, and we are going to use a behavior on our date fields. First, add this line of code after the first PHP tag:

```
use Phalcon\Mvc\Model\Behavior\Timestampable;
```

Then, add the following lines of code to the `initialize` function in this model:

```
$this->addBehavior(new Timestampable(
    array(
        'beforeCreate' => array(
            'field' => 'published',
            'format' => 'Y-m-d H:i:s'
        )
    )
));
$this->addBehavior(new Timestampable(
    array(
        'beforeUpdate' => array(
            'field' => 'updated',
            'format' => 'Y-m-d H:i:s'
        )
    )
));
```

We can use the `Timestamable` behavior to handle the setting of our published and updated dates for us. The first parameter is the event we are attaching this behavior to, the next parameter is the field, and the third is the date format. So, the published date will now be set when a new record is created, as will the updated date when it is updated, without having to set the date manually whenever we modify data in our controller.

Now let's add tags to our posts. The first thing we need to do is go back to Phalcon web tools in the browser or Phalcon Developer Tools in the command line and generate the models for both the `tags` table and the `post_tags` table. Then, we regenerate the model for our `posts` table. Make sure to choose **Set Foreign Keys** and **Define Relations**. For the `posts` table, you will have to set the **Forced** option.

Open your newly generated files. You will find them in the `models` folder located at `app`. At the bottom of the `Posts.php` file, you will find the following new code:

```
public function initialize() {
    $this->hasMany("id", "PostTags", "posts_id", NULL);
    $this->belongsTo("users_id", "Users", "id", array("foreignKey" =>
      true));
  }
```

At the bottom of `Tags.php`, you will also find an `initialize` function:

```
public function initialize() {
        $this->hasMany("id", "PostTags", "tags_id", NULL);
    }
```

At the bottom of the `PostTags.php` file, you will find the following lines of code:

```
public function initialize() {
        $this->belongsTo("posts_id", "Posts", "id");
        $this->belongsTo("tags_id", "Tags", "id");
    }
```

The code in the `initialize` functions of these models relate these models based on the foreign keys we created in our database and allow us to create, read, update, and delete our records as objects. However, we still have to write a little bit of code to actually save our tags when we save a post.

Next, let's create a new method in our `Posts.php` file.

```
public function addTags($tags) {
        foreach ($tags as $t) {
            $t = trim($t);
            $tag = Tags::findFirst(array("tag = '$t'"));
            if (!$tag) {
                $tag = new Tags();
                $tag->tag = $t;
                $tag->save();
            }
            $postTag = PostTags::findFirst(
                array(
                    "conditions" => "$this->id = ?1 AND tags_id = ?2",
                    "bind" => array(
                        1 => $this->id,
                        2 => $tag->id
                    )
                )
            );
            if (!$postTag) {
                $postTag = new PostTags();
                $postTag->posts_id = $this->id;
                $postTag->tags_id = $tag->id;
                $postTag->save();
            }
            unset($tag);
            unset($postTag);
        }
```

Our function accepts an array of tags as the first parameter and the ID of a post as the second. Here, we loop through the tags. Using `findFirst`, we check if the tag already exists. In this example, we use an array that simply holds what would be the `WHERE` clause of our SQL. If there is a tag, it is loaded in the `$tag` variable. If not, `$tag` will be false, and then we create a new tag and save it. Next, we check if there is a record in the `post_tag` table that relates the current tag to the post ID. If there is, we load it, and if not, we create it and save it.

We have the model handled, so let's move to the controllers. Open up the `PostsController.php` file located at `app/controllers`. Find the following line of code in both the `createAction` and `saveAction` functions:

```
$success = $post->save;
```

This is the point where the post gets saved to the database after being created or edited. The `$post` object will now have an ID attached to it. After each post, add the following lines of code:

```
  $tags = explode(",", $this->request->getPost("tags", "lower"));
  $post->addTags($tags);
```

Here, we are assuming that when we get to creating our form, we will be accepting comma delimited tags. So, we turn this string of tags into an array with `explode` after setting the string to lowercase letters in order to ensure a bit more uniqueness. Then, we use the `addTags` method that we just added to our `Post` model in order to save the tags to our database.

Now, we need to modify the `editAction` function so that we can send tags to our edit form. Locate the following line of code in that function:

```
$this->view->id = $post->id;
```

Add the following code after it:

```
 $tagArray = array();
 foreach ($post->postTags as $postTag) {
       $tagArray[] = $postTag->tags->tag;
 }
$this->tag->setDefault("tags", implode(",", $tagArray));
```

Now, all we have to do is modify our post views. The `edit.volt` and `new.volt` files located at `app/view/post` just need a field added to our form for tags at the bottom of the table, before the button markup.

```
<tr>
        <td align="right">
            <label for="tags">Tags</label>
        </td>
        <td align="left">
                {{ text_field("tags") }}
        </td>
    </tr>
```

In the `show.volt` file located at `app/view/post`, find the following line of code:

```
{{ post.body }}
```

Insert the following code snippet below the previous code snippet:

```
<div>
   {% for posttag in post.postTags %}
       {{ posttag.tags.tag }},
   {% endfor %}
</div>
```

Now we can browse to `http://localhost/phalconBlog/posts/new` and create a post that has tags.

Using PHQL

Phalcon Query Language (**PHQL**) is a high-level SQL dialect that standardizes SQL queries for the database systems that Phalcon supports. However, this is not the only feature of PHQL. It also has the following features:

- It uses bound parameters to secure your application
- It treats tables as models and fields as class attributes
- It allows data manipulation statements to prevent data loss
- It allows only one query per call to prevent SQL injection

We are going to use PHQL for the search form in our sidebar. Currently, it is actually pretty useless. We will cut it down to search only the body field just to make things easy. Now, we are going to make it work a little bit better. We can make the value from this one search field do a little more work by searching not only the body, but the title and the excerpt as well. So, we need to edit the `searchAction` function in the `PostsController.php` file located at `app/controllers`. The following code snippet tells us how we need to edit the file:

```
if ($this->request->isPost()) {
            //$query = Criteria::fromInput($this->di, "Posts",
              $_POST);
            //$this->persistent->parameters = $query->getParams();
            $this->persistent->parameters = $this->request->getPost();
        } else {
            $numberPage = $this->request->getQuery("page", "int");
        }

        $parameters = $this->persistent->parameters;
        if (!is_array($parameters)) {
            $parameters = array();
        }
        //$parameters["order"] = "id";
        $query = $parameters['body'];

        //$posts = Posts::find($parameters);
        $phql = "SELECT * FROM Posts WHERE body LIKE '%$query%' OR
            excerpt LIKE '%$query%' OR title LIKE '%$query%' ORDER BY
              id";
        $posts = $this->modelsManager->executeQuery($phql);
```

The lines of code that have been commented out in the previous code snippet are old code. The important part of the code is at the end where we create the PHQL query. Notice that instead of selecting from a table, we select from the actual model. Then, we set the `$posts` variable to the results returned by `$this->modelsManager->executeQuery()`. `modelsManager` is a service that was loaded into our Dependency Injection container.

Another thing that PHQL can help us out with is joins. Since we already defined the relationships between our models, we can use a PHQL query like the following line of code to get a list of posts with the users' names:

```
SELECT Posts.title, Users.name FROM Posts JOIN Users ORDER BY Users.
name
```

Note that we don't have to specify the predicates for the join operation. Here, an `INNER JOIN` operation is assumed, but any type of join can be used.

If you don't feel like writing the PHQL statements, there is a query builder available which is similar to the query builder in other PHP frameworks such as Zend and Yii. We could build the previous query with the following lines of code, as well as execute it and return the records:

```
$posts = $this->modelsManager->createBuilder()
    ->from('Posts')
    ->join('Users')
    ->orderBy('Users.name')
    ->getQuery()
    ->execute();
```

PHQL is a powerful language, and the scope of this book and application don't really do it justice. To learn more than what this book can teach you about PHQL, visit `http://docs.phalconphp.com/en/latest/reference/phql.html`.

Switching databases in Phalcon

We have used a MySQL database in our application. If we wanted to change the database software midstream, it would not be too hard, as long as we have the same data structure in our new database. Currently, Phalcon supports the following relational database backends:

- **MySQL**: This is the world's most used relational database management system available at `https://phalcon-php-framework-documentation.readthedocs.org/en/latest/api/Phalcon_Db_Dialect_Mysql.html`
- **PostgreSQL**: This is a powerful, reliable, and open source database management system available at `https://phalcon-php-framework-documentation.readthedocs.org/en/latest/api/Phalcon_Db_Dialect_Postgresql.html`
- **SQLite**: This is a self-contained, server-less database engine available at `https://phalcon-php-framework-documentation.readthedocs.org/en/latest/api/Phalcon_Db_Dialect_Sqlite.html`
- **Oracle**: This is an object-relational database management system produced by the Oracle corporation available at `https://phalcon-php-framework-documentation.readthedocs.org/en/latest/api/Phalcon_Db_Dialect_Oracle.html`

Our application sets the database adapter by importing it in the `services.php` file located at `app/config` with the following line of code:

```
use Phalcon\Db\Adapter\Pdo\Mysql as DbAdapter;
```

If we wanted to use a different database system, we would just include a different adapter here. Each adapter handles most of the proprietary quirks of its respective database.

Phalcon's Object-Document Mapper and MongoDB

Relational databases aren't the only type of database you can use in Phalcon. You can also use Phalcon's **Object-Document Mapper** (**ODM**) to connect to MongoDB. Phalcon's ODM offers CRUD functionality, validation, events, and other useful services.

You start by adding a connection from your MongoDB database to your Dependency Injection container, as in the following code snippet:

```
$di->set('mongo', function() {
    $mongo = new Mongo();
    return $mongo->selectDb("blog");}, true);
```

This will be a connection to the default localhost MongoDB instance running on the default port.

We can now add models just as we add models when our application uses a standard relational database, but instead, we extend `\Phalcon\Mvc\Collection`.

```
class Pages extends \Phalcon\Mvc\Collection{

}
```

And really, that is all you need in your model file. This assumes we will be using a collection called `pages` in our `blog` database to hold the data. This may seem a little confusing, but it is a standard convention to use lowercase names for MongoDB collections. This class has the same functionality as that of the MongoDB PHP extension. So, if we want to find a page by its ID, we would use the same function name.

```
$page = Pages::findById("5087358f2d42b8c3d15ec4e2");
```

Each collection is an object with the flexibility of a MongoDB record. Once you have the `page` object, it can be modified and saved again.

```
$page->title = "New Title";

$page->adhoc = "Adhoc value just in this record";
$page->save();
```

You can read more about the ODM at `http://docs.phalconphp.com/en/latest/reference/odm.html`.

Migrating databases

Another tool that both Phalcon Developer Tools and Phalcon web tools provide is one that manages database changes between your various software environments. To generate a database migration, you would run the following command in your project directory:

```
phalcon migration generate
```

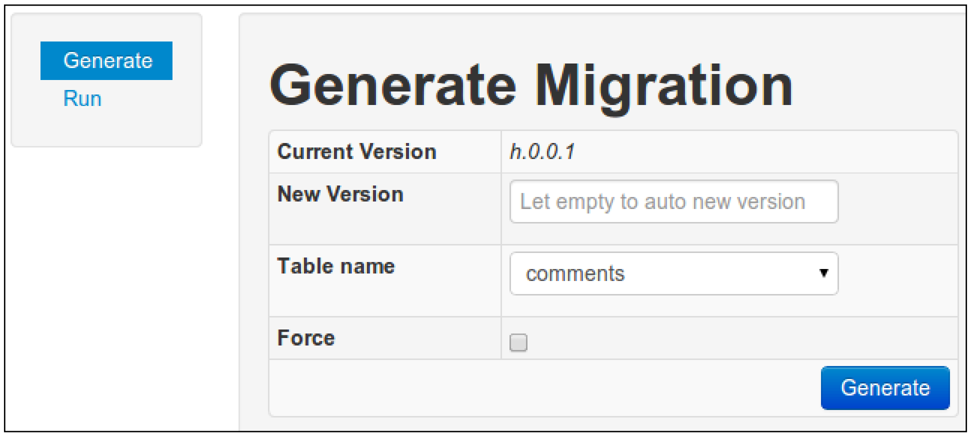
This command will simply dump every object from your database in migration classes. In our application, these classes will be stored inside a folder named after the current migration version in the `migrations` folder located at `app`. Let's say you just changed the data model in the local version of your application and you have created a migration for it. Then, you would just upload the migration to your development server and run the following command to execute the changes to your database on this server:

```
phalcon migration run
```

To list the configuration options available, run the following command:

```
phalcon migration
```

To use web tools, we would just browse to `http://localhost/phalconBlog/webtools.php?_url=/migrations/index` in our current application, and we will see the following screen:



Generate Migration

Current Version	h.0.0.1
New Version	Let empty to auto new version
Table name	comments
Force	☐

You can learn more about Phalcon database migrations at `http://docs.phalconphp.com/en/latest/reference/migrations.html`.

Summary

In this chapter, we fleshed out our blog application a little by adding relationships between our models. We can now log in, create our own posts that have updated and created dates, and add tags to our posts. During the process of refactoring our application to do these things, we learned various ways to handle data in Phalcon. However, our application is far from complete.

In the next chapter, we will fill in some of the missing gaps in our application, including the ability to comment on posts, hiding content and functionality from visitors who aren't logged in, encrypting users' passwords, generating RSS feeds, and pinging sites when we create posts.

5
Using Phalcon's Features

It's time to add some features to our blog and learn more about Phalcon's functionality in the process. There is still a lot to do to get our blog up-to-speed. Users' passwords are still in plain text, and anyone, whether they are a user or not, can browse to a user's page and view the passwords. So, we need to set up at least some sort of security and user-access control. We have a comments table, yet we don't have any way to comment. And a blog is not a blog without an RSS feed and the ability to ping feed aggregators.

In this chapter, we will cover the following topics:

- How to hash passwords with Phalcon
- How to use view helpers and view partials
- How to set cookies
- How to control the user's access
- How to cache data in our application
- How to use Phalcon's event manager
- How to route application requests

Hashing passwords with Phalcon

First, let's at least get the passwords hashed. Hashing is a process by which a password is converted into a fixed length bit string or a hash that cannot be reverse-engineered to retrieve the password, but any change in the entered password will change the resulting hash. So, for a more secure application, we are going to generate a hash from the password that a user enters and store that value in the database. Then, when the user tries to log in again, we will generate a hash from the entered password and compare it to the value of the hash in the database.

Fortunately, this is easy to do with Phalcon. The security component is automatically loaded in the Phalcon services' container. So, open up the `UsersController.php` file and find the `createAction` function; then, find the line where the password is set.

```
$user->password = $this->request->getPost("password");
```

Then, simply replace it with the following two lines of code:

```
$password = $this->request->getPost("password");
$user->password = $this->security->hash($password);
```

Now, when we create a user, the password will be hashed and saved to the database. We also want to save the password again as a hash if we edit. So, we find the same line of code in the `saveAction` function and replace it with the previous two lines of code to hash the password.

Now, all we need to do is edit our `loginAction` function. Our initial function was kind of clunky. So, we are going to replace it with something a little more streamlined. The new `loginAction` function will look like the following code snippet:

```
public function loginAction() {

        if ($this->request->isPost()) {
            $username = $this->request->getPost('username');
            $password = $this->request->getPost('password');

            $user = Users::findFirstByUsername($username);
            if ($user && $this->security->checkHash($password,
              $user->password)) {
                $this->session->set("user_id", $user->id);
                $this->cookies->set('user_id', $user->id);
                $this->session->set('auth', array(
                    'id' => $user->id,
                    'name' => $user->name
                ));
                $this->flash->success("Welcome " . $user->name);
            }else{
                $this->flash->error("Username and Password combination
                  not found");
            }
        }
        return $this->dispatcher->forward(
            array(
                "controller" => "posts",
                "action" => "index"
            )
        );
    }
```

Instead of searching for the user in the database, we try to instantiate the user using the `findFirstByFieldname` function, where `Fieldname` can be replaced with the column in the database we are querying. If we do find a user by searching for the username, we use security services' `checkHash` function to compare the hash in the database with the hash we just generated off of the entered password. If there is a match, we go through the process of setting the `user_id` variable in the session and creating a welcome message.

The Phalcon security component uses the bcrypt hash algorithm, which gives us a high level of security. This component is loaded in Phalcon by default, but it can also be configured manually to tweak the "slowness" of the bcrypt algorithm. You can learn more about this component at `http://docs.phalconphp.com/en/latest/reference/security.html`.

Using Phalcon view helpers

We can also add some security by changing the password fields on the forms to true password fields. So, find the following files in the `users` folder located at `app/views`:

- `new.volt`
- `index.volt`
- `edit.volt`

And find the following line of code:

```
{{ text_field("password", "size" : 30) }}
```

Replace the preceding code with the following line of code:

```
{{ password_field("password", "size" : 30) }}
```

Now, passwords will be masked when they are entered in the field.

As we learned in *Chapter 3*, *Using Phalcon Models, Views, and Controllers*, the Volt template engine is a very thin wrapper that is wrapped around PHP code, which Phalcon compiles to the actual PHP code. To call a Phalcon tag helper in Volt, we just use the uncamelized version of the function.

Using dynamic title tags

Let's replace some of the HTML markup we wrote with some view helpers. A powerful view helper is the one for the document title. Right now, we have a hardcoded `title` tag in our main layout which gives the same title to every page. So, open up the `index.volt` file located at `app/views` and find the following line of code:

```
<title>Phalcon Blog</title>
```

Replace the preceding code with the following line of code:

```
{{ get_title() }}
```

Now, the only problem is that our page has no title because we haven't set it yet. It would probably be a good idea to keep the title of our blog on every page. We could hardcode that or open up the `ControllerBase.php` file located at `app/controllers` and add the following function to the `ControllerBase` class:

```
public function initialize() {
        $this->tag->setTitle("Phalcon Blog");
}
```

Now, we have our global title back again. But we want each blog post to have the title of the post in the title also. So, if we have a blog post called Hello World, we want our title to be Hello World: Phalcon Blog. It's easy to do this. Open up the `PostController.php` file located at `app/controllers` and add the following line of code to the end of the `showAction` function:

```
$this->tag->prependTitle($post->title . " - ");
```

The title of a blog post is displayed before the blog title if you visit a blog post page. Now, if you want, you can add your own title to each action in each controller.

Setting the doctype

By setting the document type with Phalcon, you affect all the tags that it generates with tag helpers, so they conform to your chosen standard. We are going to use HTML5. Add the following line of code to the `initialize` function in the `ControllerBase.php` file located at `app/controllers`:

```
$this->tag->setDoctype(\Phalcon\Tag::HTML5);
```

Now, the document type will be set globally in each controller. Find the following line of code at the top of the `index.html` file located at `app/views`:

```
<!DOCTYPE html>
```

And replace the preceding code with the following line of code:

```
{{ get_doctype() }}
```

Now, you can be sure that Phalcon will output tags that follow whatever standard you choose. To learn about the other document standards, you can visit `http://docs.phalconphp.com/en/latest/reference/tags.html#document-type-of-content`.

Adding JavaScript and CSS with view helpers

We originally just hardcoded the CSS and JavaScript links in the head of our main layout file. This works for now, but if we move our project, we may have to change these links. Phalcon has script and CSS tag helpers that will make our job a little easier. We can replace the CSS links in the head of the `index.phtml` file located at `app/views` with the following code snippet:

```
{{ stylesheet_link("css/bootstrap/bootstrap.min.css") }}
{{ stylesheet_link("css/bootstrap/bootstrap-responsive.min.css") }}
```

We put the JavaScript links at the bottom of the file with the following lines of code:

```
{{ javascript_include("js/jquery.min.js") }}
{{ javascript_include("js/bootstrap.min.js") }}
```

We will explore more view helpers as we read through this chapter. To learn more about Phalcon tag helpers, visit `http://docs.phalconphp.com/en/latest/api/Phalcon_Tag.html`.

Setting cookies

Now we can log in and have secure passwords, but every time the browser is closed, we have to log in again. It would be nice to be able to log in to our blog and stay logged in for a few days at least. It is time to set a cookie.

We are going to change the way we do this as we go through this chapter, but for now, it is very easy to set a cookie that will keep us logged in with very little code. First, we need to set up an encryption key because we will want to decrypt a cookie before setting it and decrypt it while retrieving it. We want to keep it that way, but in order to do this, we need to give it a key. So, go to `http://randomkeygen.com/` to generate a random string of digits. Then, open up the `config.ini` file located at `app/config` and add your key in the application section and call it `encryptKey`, shown as follows:

```
[application]
controllersDir = /home/eristoddle/Dropbox/xampp/htdocs/phalconBlog/
app/controllers/
modelsDir = /home/eristoddle/Dropbox/xampp/htdocs/phalconBlog/app/
models/
viewsDir = /home/eristoddle/Dropbox/xampp/htdocs/phalconBlog/app/
views/
pluginsDir = ../app/plugins/
libraryDir = ../app/library/
baseUri = /phalconBlog/
cacheDir = ../app/cache/
encryptKey = 2tx6]GD}532q4x_
```

Then, open up the `services.php` file located at `app/config` and add the following code snippet to the bottom of the file in order to set the encryption key:

```
$di->set(
    'crypt', function () use ($config) {
        $crypt = new Phalcon\Crypt();
        $crypt->setKey($config->application->encryptKey);
        return $crypt;
    }
);
```

Now, our cookies are ready to go. They were available earlier, but had we used them, they would have thrown an error about requiring an encryption key. Now, find the `loginAction` function in the `UsersController.php` file located at `app/controllers` and add the following line of code:

```
$this->cookies->set('user_id', $user->id);
```

Add it right after you set the `user_id` variable in the session:

```
$this->session->set("user_id", $user->id);
```

Now, open up the `PostsController.php` file located at `app/controllers` and find the part of the `createAction` function where we check the `user_id` variable in the session. Now, we are going to check for cookies first and then set the `user_id` variable in the session if we find a `user_id` cookie:

```
if ($this->cookies->has('user_id')) {
            $this->session->set('user_id', $this->cookies->get
              ('user_id'));
        }
        if ($this->session->has("user_id")) {
            return $this->dispatcher->forward(
                array(
                    "controller" => "users",
                    "action" => "search"
                )
            );
        }
```

And that's about all that we have to do to retrofit our code to add cookies. To learn more about cookie management with Phalcon, see `http://docs.phalconphp.com/en/latest/reference/cookies.html`.

Now, we should be able to log out and log in again, close the browser, and still be logged in. However, this is not really all that we want when it comes to controlling users. We want to be able to seamlessly control everything they access. Currently, our application has holes, and to plug them, we need to copy and paste code to implement the same `user_id` variable check everywhere we need it's functionality. Fortunately, we won't have to do this. Phalcon provides a service that helps us control user access.

Controlling user access

An **access control list** (**ACL**) uses roles to control access to resources. `Phalcon\Acl` provides this functionality for us. With it, we can assign roles to the different types of visitors on our blog, and then use these roles to set up permissions on each action in each of our controllers. For our purposes, we are only going to create two roles, users and guests. A user will simply be a visitor that is logged in, a guest, or anyone else. We are only going to give guest access to perform the following actions:

- View the index page:
    - View the posts' index page
    - Comment on a post
    - View the users' index, which will be a login page, when a visitor is not logged in
- Log in

A logged in user can do everything. To put `Phalcon\Acl` to use in our application, we are going to create a simple Phalcon plugin. Phalcon Developer Tools already added our `plugins` folder and created the `pluginsDir` setting in the `config.ini` file. However, we still need to add the `plugins` directory to our loader. Open the `loader.php` file located at `app/config` and add the `plugins` directory to your loader.

```
$loader->registerDirs(
    array(
        $config->application->controllersDir,
        $config->application->modelsDir,
        $config->application->pluginsDir
    )
)->register();
```

What we want to create is a plugin that will hook into Phalcon's event manager. Using the event manager, we can attach listeners to various Phalcon events. Create a new file called `Security.php` in the `app/plugins` folder and add the following code snippet at the top of the file:

```
<?php

use Phalcon\Events\Event,
    Phalcon\Mvc\User\Plugin,
    Phalcon\Mvc\Dispatcher,
    Phalcon\Acl;

class Security extends Plugin {

    public function __construct($dependencyInjector) {
        $this->_dependencyInjector = $dependencyInjector;
    }
```

We extend `Phalcon\Plugin`. Our constructor must accept a dependency injector instance. Then, we need to create a function that will add our ACL to our persistent variables. We will call it `getAcl`.

```
    public function getAcl() {
        if (!isset($this->persistent->acl)) {

            $acl = new Phalcon\Acl\Adapter\Memory();
            $acl->setDefaultAction(Phalcon\Acl::DENY);

            $roles = array(
                'users' => new Phalcon\Acl\Role('Users'),
                'guests' => new Phalcon\Acl\Role('Guests')
            );
            foreach ($roles as $role) {
                $acl->addRole($role);
            }

            $private = array(
                'comments' => array('index', 'edit', 'delete',
                  'save'),
                'posts' => array('new', 'edit', 'save', 'create',
                  'delete'),
                'users' => array('search', 'new', 'edit', 'save',
                  'create', 'delete', 'logout')
            );
            foreach ($private as $resource => $actions) {
                $acl->addResource(new Phalcon\Acl\Resource($resource),
                  $actions);
            }

            $public = array(
                'index' => array('index'),
                'posts' => array('index', 'search', 'show',
                  'comment'),
                'users' => array('login', 'index')
            );
            foreach ($public as $resource => $actions) {
                $acl->addResource(new Phalcon\Acl\Resource($resource),
                  $actions);
            }

            foreach ($roles as $role) {
                foreach ($public as $resource => $actions) {
                    foreach ($actions as $action) {
                        $acl->allow($role->getName(), $resource,
                          $action);
                    }
                }
            }

            foreach ($private as $resource => $actions) {
                foreach ($actions as $action) {
                    $acl->allow('Users', $resource, $action);
                }
            }

            $this->persistent->acl = $acl;
        }

        return $this->persistent->acl;
    }
```

First, the code checks if we already have an ACL instance. If not, we create one. We then set the `DefaultAction` function to deny access for security reasons. The next bit of code creates the roles, that is, users and guests, in our ACL. We then load all of the resources we want to be private into a variable and then add them all as resources. We will do the same for the resources that we want to be public. Then, we loop through the roles, giving both the roles access to the public resources. Then, we loop through the private resources and give access to users but not to guests. Finally, we set our new ACL to be persistent and return it.

This function just builds the structure of our ACL. We still need to actually control the access, and we will do that with a function that will fire before dispatch. Add the following function at the bottom of the file:

```
    public function beforeDispatch(Event $event, Dispatcher
$dispatcher) {

        $user = $this->session->get('user_id');
        if (!$user) {
            $role = 'Guests';
        } else {
            $role = 'Users';
        }

        $controller = $dispatcher->getControllerName();
        $action = $dispatcher->getActionName();
        $acl = $this->getAcl();

        $allowed = $acl->isAllowed($role, $controller, $action);
        if ($allowed != Acl::ALLOW) {
            $this->flash->error("You don't have access to this
              module");
            $dispatcher->forward(
                array(
                    'controller' => 'index',
                    'action' => 'index'
                )
            );
            return false;
        }

    }

}
```

This function accepts an event instance and a dispatcher instance. We check if the visitor is logged in and set their role. Then, we set a variable to our controller name and one to the action name. We call our `getAcl` function to get our ACL, and we also call `$acl->isAllowed()`, passing the role of the visitor and the names of our controller and action. This will return true if the role is allowed access to the current resource. If the user is not allowed access, we set an error message and forward them to the index of our blog.

Now, we need to hook into the standard dispatcher, attach this new `Security` plugin to our events manager, and set this as the event manager for our dispatcher. Open the `services.php` file located at `app/config` and add the following code snippet to the bottom of the file:

```
$di->set('dispatcher', function() use ($di) {
    $eventsManager = $di->getShared('eventsManager');
    $security = new Security($di);
    $eventsManager->attach('dispatch', $security);
    $dispatcher = new Phalcon\Mvc\Dispatcher();
    $dispatcher->setEventsManager($eventsManager);
    return $dispatcher;
});
```

To learn more about using `Phalcon\Acl`, visit `http://docs.phalconphp.com/en/latest/reference/acl.html`. And to learn more about Phalcon's event manager, visit `http://docs.phalconphp.com/en/latest/reference/events.html`.

Applying the finishing touches to our application

Phalcon Developer Tools can help out a lot, but let's face it, most of the code you are going to write for your application is from scratch. Our blog still doesn't have comments or a feed, so it is really not much of a blog. Adding these is going to be a manual job, but Phalcon makes it easy.

Adding comments

Now, let's give visitors the ability to comment on our posts. Earlier, we needed a user-access control so that we could limit moderating comments to logged-in users. But now, we need a place to do it. We are going to build a very simple comment-moderation system. We could use Phalcon Developer Tools or Phalcon web tools to generate our CRUD scaffolding for us, but in this case, we would almost be taking out more of the generated code than we'd be leaving in, but we can use the other controllers and views we generated as a guide.

For comments, we need to add a form to comment on the posts showing an action view, and we also need to display comments that have been published under each post. We also need a place to list all comments, and a form to edit them and set them to be published.

Comments are already related to posts. We just need to modify a few things to make this relationship functional. Open the `show.volt` file located at `app/views/posts`. At the bottom of the file, add the following code snippet:

```
<div class="comments">
    <h3>Comments</h3>
    <div class="comment">
    {% for comments in post.comments if comments.publish == true %}
        <div>{{ comments.body }}</div>
        <div><a href="{{ comments.url }}">{{ comments.name }}</a>
          </div>
    {% endfor %}
    </div>
    <h3>Leave a Comment</h3>
    <div>
        {{ form("posts/comment", "method":"post") }}
        {{ hidden_field("posts_id", "value" : post.id) }}
        <div>
            <label for="title">Comment</label>
            {{ text_area("body") }}
        </div>
        <div>
            <label for="title">Name</label>
            {{ text_field("name") }}
        </div>
        <div>
            <label for="title">Email</label>
            {{ text_field("email") }}
        </div>
        <div>
```

```
                <label for="title">Website</label>
                {{ text_field("url") }}
            </div>
            {{ submit_button("Comment", "class" : "btn") }}
            {{ end_form() }}
        </div>
    </div>
```

Right below the post, we loop through the comments that are related to that post, and if we set them to publish, we print the comment. Below that, we have the comment form, making sure that we have a hidden field that picks up the ID of the post. We make this form post to posts/comment, which means that we now need a `commentAction` function in our `Posts` controller. Open the `PostsController.php` file located at `app/controllers` and add the following function:

```
    public function commentAction(){
            $comment = new Comments();
            $comment->posts_id = $this->request->getPost("posts_id");
            $comment->body = $this->request->getPost("body");
            $comment->name = $this->request->getPost("name");
            $comment->email = $this->request->getPost("email");
            $comment->url = $this->request->getPost("url");
            $comment->submitted = date("Y-m-d H:i:s");
            $comment->publish = 0;
            $comment->save();

            $this->flash->success("Your comment has been submitted.");

            return $this->dispatcher->forward(
                array(
                    "controller" => "posts",
                    "action" => "show",
                    "params" => array($comment->posts_id)
                )
            );
    }
```

It's pretty simple. We create an instance of the `Comments` object from the `POST` variable, set the submitted date, and set the publish value to 0, which will represent false in our database. Then, we set a success message and forward the user back to the `Posts` show action, setting the ID to the ID of the post that was just commented on.

Now, we need a way to moderate our comments. First, we need a `Comments` controller. So, create a `CommentsController.php` file at `app/controllers` and make sure to include `Paginator`, because we will be using it.

```
use Phalcon\Paginator\Adapter\Model as Paginator;
```

Then, we need our `CommentsController` class.

```
class CommentsController extends ControllerBase {
  //Actions will go here
}
```

And inside the `CommentsController` class, we need actions for indexing, editing, saving, and deleting. Our `indexAction` function returns a paginated result of all comments.

```
    public function indexAction() {
        $numberPage = $this->request->getQuery("page", "int", 1);

        $comments = Comments::query()
            ->order("submitted DESC")
            ->execute();

        $paginator = new Paginator(array(
            "data" => $comments,
            "limit" => 10,
            "page" => $numberPage
        ));

        $this->view->page = $paginator->getPaginate();
    }
```

Our `editAction` function prepares a single comment for use in the edit form, when we are moderating a comment, by setting it to be published.

```
    public function editAction($id) {

        if (!$this->request->isPost()) {

            $comment = Comments::findFirstByid($id);
            if (!$comment) {
                $this->flash->error("comment was not found");
                return $this->dispatcher->forward(
                    array(
                        "controller" => "comments",
                        "action" => "index"
                    )
                );
            }

            $this->view->id = $comment->id;

            $this->tag->setDefault("id", $comment->id);
            $this->tag->setDefault("body", $comment->body);
            $this->tag->setDefault("name", $comment->name);
            $this->tag->setDefault("email", $comment->email);
            $this->tag->setDefault("url", $comment->url);
            $this->tag->setDefault("submitted", $comment->submitted);
            $this->tag->setDefault("publish", $comment->publish);
            $this->tag->setDefault("posts_id", $comment->posts_id);

        }
    }
```

The `saveAction` function saves an edited comment after we moderate it.

```
    public function saveAction() {

        if (!$this->request->isPost()) {
            return $this->dispatcher->forward(
                array(
                    "controller" => "comments",
                    "action" => "index"
                )
            );
        }

        $id = $this->request->getPost("id");

        $comment = Comments::findFirstByid($id);
        if (!$comment) {
            $this->flash->error("comment does not exist " . $id);
            return $this->dispatcher->forward(
                array(
                    "controller" => "comments",
                    "action" => "index"
                )
            );
        }

        $comment->id = $this->request->getPost("id");
        $comment->body = $this->request->getPost("body");
        $comment->name = $this->request->getPost("name");
        $comment->email = $this->request->getPost("email", "email");
        $comment->url = $this->request->getPost("url");
        $comment->publish = $this->request->getPost("publish");
```

```
            if (!$comment->save()) {

                foreach ($comment->getMessages() as $message) {
                    $this->flash->error($message);
                }

                return $this->dispatcher->forward(
                    array(
                        "controller" => "comments",
                        "action" => "edit",
                        "params" => array($comment->id)
                    )
                );
            }

            $this->flash->success("comment was updated successfully");
            return $this->dispatcher->forward(
                array(
                    "controller" => "comments",
                    "action" => "index"
                )
            );

        }
```

The `deleteAction` function gets rid of our comment spam.

```
        public function deleteAction($id) {

            $comment = Comments::findFirstByid($id);
            if (!$comment) {
                $this->flash->error("comment was not found");
                return $this->dispatcher->forward(
                    array(
                        "controller" => "comments",
                        "action" => "index"
                    )
                );
            }

            if (!$comment->delete()) {

                foreach ($comment->getMessages() as $message) {
                    $this->flash->error($message);
                }

                return $this->dispatcher->forward(
                    array(
                        "controller" => "comments",
                        "action" => "search"
                    )
                );
            }

            $this->flash->success("comment was deleted successfully");
            return $this->dispatcher->forward(
                array(
                    "controller" => "comments",
                    "action" => "index"
                )
            );
        }
```

Now, we just need views. First, we need the view for our `indexAction` function. So, create a file called `index.volt` in the `comments` folder located at `app/views` and insert the following code snippet in it:

```
{{ content() }}
<h1>Comments</h1>
<table class="browse" align="center">
    <thead>
        <tr>
            <th>Body</th>
            <th>Name</th>
            <th>Email</th>
            <th>Url</th>
            <th>Submitted</th>
            <th>Publish</th>
         </tr>
    </thead>
    <tbody>
    {% if page.items is defined %}
    {% for comment in page.items %}
        <tr>
            <td>{{ comment.body }}</td>
            <td>{{ comment.name }}</td>
            <td>{{ comment.email }}</td>
            <td>{{ comment.url }}</td>
            <td>{{ comment.submitted }}</td>
            <td>{{ comment.publish }}</td>
            <td>{{ link_to("comments/edit/"~comment.id, "Edit") }}
            </td>
            <td>{{ link_to("comments/delete/"~comment.id, "Delete")
}}</td>
        </tr>
    {% endfor %}
    {% endif %}
    </tbody>
    <tbody><tr><td colspan="2" align="right">
        <table align="center">
```

```
            <tr>
                <td>{{ link_to("comments/search", "First") }}</td>
                <td>{{ link_to("comments/search?page="~page.before,
                  "Previous") }}</td>
                <td>{{ link_to("comments/search?page="~page.next,
                  "Next") }}</td>
                <td>{{ link_to("comments/search?page="~page.last,
                  "Last") }}</td>
                <td>{{ page.current~"/"~page.total_pages }}</td>
            </tr>
        </table>
    </td></tr></tbody>
</table>
```

Then, the view that will be used when we edit our posts. Save the following code snippet as `edit.volt` at `app/views/comments`:

```
{{ content() }}
{{ link_to("comments", "Go Back") }}

<div align="center">
    <h1>Edit comments</h1>
</div>

<div>
    {{ form("comments/save", "method":"post") }}
        <label for="body">Body</label>{{ text_area("body") }}
        <label for="name">Name</label>{{ text_field("name") }}
        <label for="email">Email</label>{{ text_field("email") }}
        <label for="url">Url</label>{{ text_field("url") }}
        <label for="publish">Publish</label>
        {{ radio_field("publish", "value" : 1) }} Yes
        {{ radio_field("publish", "value" : 0) }} No
        {{ hidden_field("id") }}
        {{ submit_button("Save", "class" : "btn") }}
    {{ end_form() }}
</div>
```

Now, we can comment on posts and moderate them if we are logged in as a user.

Adding feeds

A blog is not really a blog without a way to syndicate our posts. Fortunately, it is going to be pretty simple to add a feed to our blog. Let's put our feed at `http://localhost/phalconBlog/posts/feed`. This means that we need a new `feedAction` function in our posts controller. So, open the `PostsController.php` file located at `app/controllers` and add the following function:

```
public function feedAction() {
    $posts = Posts::find(
        array(
            'order' => 'published DESC',
            'limit' => 10
        )
    );

    $rss_posts = array();
    foreach ($posts as $post){
        $post->rss_date = date("D, d M Y H:i:s O", strtotime
          ($post->published));
        $rss_posts[] = $post;
    }
    $this->view->posts = $rss_posts;

    $this->view->setRenderLevel(Phalcon\Mvc\View::LEVEL_ACTION_VIEW);
}
```

Here, we retrieve only 10 posts ordered by the published date in descending order using `Posts::find`. As the date format in an RSS feed is very specific and doesn't match the MySQL datetime format, we loop through our records, converting the published date and adding this date to each `post` object. We then send the array of objects to the view. At the end of the function, we set the render level of the view. In this case, we don't want our main layout or post layout to render. We only want the specialized template that we are about to create to render, so we set our level to `LEVEL_ACTION_VIEW`. The following six levels are available:

- `LEVEL_NO_RENDER`: This does not render any view
- `LEVEL_ACTION_VIEW`: This renders the view associated with the action
- `LEVEL_BEFORE_TEMPLATE`: This generates the views before the controller's layout
- `LEVEL_LAYOUT`: This generates the controller's layout
- `LEVEL_AFTER_TEMPLATE`: This generates the views after the controller is laid out
- `LEVEL_MAIN_LAYOUT`: This generates the main layout

Create a new file in the `posts` folder at `app/views`, call it `feed.volt`, and insert the following code snippet inside the `feed.volt` file:

```
{{'<?xml version="1.0" encoding="UTF-8" ?>'}}

<rss version="2.0">
    <channel>
        <title>{{ config.blog.title }}</title>
        <description>This is a demonstration of the Phalcon
          framework</description>
        <link>{{ config.blog.url }}</link>
        {% for post in posts %}
            <item>
                <title>{{ post.title|e }}</title>
                <description>{{ post.excerpt|e }}</description>
                <link>{{ config.blog.url }}{{ url("posts/show/
                  "~post.id) }}</link>
                <guid>{{ config.blog.url }}{{ url("posts/show/
                   "~post.id) }}</guid>
                <pubDate>{{ post.rss_date }}</pubDate>
            </item>
        {% endfor %}
    </channel>
</rss>
```

We wrap the XML tag at the top of the Volt template tag to hide `<?` from the Volt template engine. We add some necessary RSS elements including a title, description, and a link for our blog. Note that since we loaded our configuration into the DI container, we have access to the settings we created there for our blog's title and URL. The next step is to loop through the posts. This part is similar to the code in the `index.volt` file. But you may notice `|e` behind the title and the excerpt of the post. This is the Phalcon HTML escaping filter being called from Volt so that our feed will remain valid with any characters that may be in our content. We are generating XML and using the `rss_date` attribute that we created in our `feedAction` function.

The last step we need to do is add the link for RSS autodiscovery to the head of our pages. So, open the `index.volt` file located at `app/views` and add the following code snippet between the head tags:

```
{{ tag_html(
            "link",
            [
                "rel": "alternate",
                "type": "application/rss+xml",
                "title": "RSS Feed for Phalcon Blog",
                "href": config.application.baseUri~"posts/feed"
            ],
             true,
             true,
             true)
 }}
```

Here, we use the generic `tagHtml()` tag helper to generate the link for us, which becomes `tag_html()` in Volt. The first parameter is the name of the tag. The second parameter is an array of the tag's attributes. Note that the `href` parameter uses the `baseUri` configuration setting and `~`, which is a character used in Volt to concatenate strings together. The third parameter is a Boolean we set to true because we want this to be a self-closing tag. We set the fourth parameter to true because we only want to generate the start tag. Finally, we set the fifth parameter to true because we want an end-of-line character generated after the tag.

Now, we can start telling the rest of the Internet about our new blog.

Sending update pings

In order to ping sites such as `http://weblogs.com` and notify them that we have a new post, we are going to do a little more refactoring. It seems we used the title of our blog in the HTML title tag. We have to use it again in the function that we'll write to ping. So, instead of hardcoding the title in two places that have to be edited, if we change the title, it would be better to create a configuration setting. So, add the following code snippet to the `config.ini` file located at `app/config`:

```
[blog]
title = Phalcon Blog
url = http://localhost
```

As we are now going to use this setting in a controller, we need to have access to it there. To do this, we can add it to the Dependency Injection container. This can be done by adding these lines of code to the bottom of the `services.php` file located at `app/config`:

```
$di->set('config', $config);
```

Now, we can open the `ControllerBase.php` file located at `app/controllers` and set our title tag with our configuration setting.

```
$this->tag->setTitle($this->config->blog->title);
```

We can create a new function in our `PostsController.php` file to ping the feed aggregator sites for us.

```
private function sendPings(){
        $request = '<?xml version="1.0" encoding="iso-8859-1"?>
                     <methodCall>
                     <methodName>weblogUpdates.ping</methodName>
                     <params>
                      <param>
                       <value>
                        <string>'.$this->config->blog->title.'</string>
                       </value>
                      </param>
                      <param>
                       <value>
                        <string>'.$this->config->blog->url.$this->url-
>get('posts/feed').'</string>
                       </value>
                      </param>
                     </params>
                     </methodCall>';

        $ping_urls = array(
            'http://blogsearch.google.com/ping/RPC2',
            'http://rpc.weblogs.com/RPC2',
            'http://ping.blo.gs/'
        );
        foreach($ping_urls as $ping_url){
            $ch = curl_init();
            curl_setopt($ch, CURLOPT_URL, $ping_url);
            curl_setopt($ch, CURLOPT_RETURNTRANSFER, true );
            curl_setopt($ch, CURLOPT_POST, true );
            curl_setopt($ch, CURLOPT_POSTFIELDS, trim($request));
            $results = curl_exec($ch);
        }
        curl_close($ch);
    }
```

The `$request` variable is the XML we are going to post to the services. Now that we've added the configuration settings to our Dependency Injection container, we can access the blog's title setting with `$this->config->blog->title` and the URL setting with `$this->config->blog->url`. The `$ping_urls` array contains the URLs of services we are going to ping. You can add more services to this array if you like. Then, we use PHP `curl` to ping.

Now, in the `createAction` function, find the following line of code:

```
$this->flash->success("post was created successfully");
```

Add the following line of code before the preceding code:

```
$this->sendPings();
```

Another feature we could add to this function is a way of logging the results of our pings, so we then have a way of debugging what is actually happening as our function doesn't return a status.

Now, it's time to add another variable to the application section of the configuration file.

```
logsDir = ../app/logs/
```

Now, we need to add a service to log the results of our pings in the Dependency Injection container. Open up the `services.php` file located at `app/config` and add the following lines of code:

```
//Logging
$di->set(
    'pingLogger', function () use ($config){
        $logger = new \Phalcon\Logger\Adapter\File
          ($config->application->logsDir.'ping.log');
        return $logger;
    }
);
```

This will enable us to access our `pingLogger` service in the `sendPings` function we just created. So, after we set the `$result` variable, we can then log the URL we pinged along with the response we received. We check if the result is false, indicating that `curl` failed, and then change the `$result` variable to reflect that error.

```
$result = curl_exec($ch);
if($result === false){
       $result = "Curl Error";
}
$this->pingLogger->log($ping_url.PHP_EOL.$result);
```

Using view partials

We have explored layouts and cascading views. We also have the option to use partials in Phalcon. Using partial templates allows us to reuse the same functionality in various parts of our application. A couple of places in which we may want to use partials are the navigation bar and the sidebar. Right now, our main layout does a lot of stuff. There is not much on the navigation bar or the sidebar. But as we add more functionalities, they might become more complex, and keeping them in separate files will help us to organize and simplify our code.

You can create a folder to hold your partial views wherever you choose. You can even put them in your `views` folder located at `app` if you choose. But, to keep our templates more organized, let's create a folder called `partials` inside of the `views` folder located at `app`. Now, open the `index.volt` file located at `app/views`. We are going to cut the sidebar out of this file and paste it in a new file called `sidebar.volt`, which we are going to save in our new `partials` folder. The content of that file should look like the following code snippet:

```
<div class="span3">
    <div class="well">
        {{ form("posts/search", "method":"post", "autocomplete" :
          "off", "class" : "form-inline") }}
            <div class="input-append">
                {{ text_field("body", "class" : "input-medium") }}
                {{ submit_button("Search", "class" : "btn") }}
            </div>
        {{ end_form() }}
    </div>
</div>
```

We will replace this in the `index.volt` file with the following line of code:

```
{{ partial("partials/sidebar") }}
```

This is the code we would use with Volt. If we were using PHP templates, we would use `<?php $this->partial("partials/sidebar"); ?>`. Note that we don't add the file extension to the `path` parameter. Make sure to do this with your `partials` folder. Adding the extension will cause an error.

Now, you can cut the `div` tag with the `navbar` class in the `index.phtml` file located at `app/views`, paste it into a file called `navbar.volt`, save it in your `partials` folder, and replace the `div` tag with the following line of code:

```
{{ partial("partials/navbar") }}
```

Now, the different types of functionalities are separated in your application. You may find other places in your application where you may want to use partials, such as in the header or the footer. To learn more about Phalcon, visit `http://docs.phalconphp.com/en/latest/index.html`.

Caching in Phalcon

Caching can speed up and save resources on a site that gets a lot of traffic or does a lot of intense repeatable database queries, but this should only be implemented where actually needed. In other words, our blog, in its current iteration, probably doesn't need a cache, but we are going to take a look at caching in Phalcon and add caching to a part of our blog just to get a handle on how it works.

Setting up a cache service

In Phalcon, the cache consists of two parts, a frontend cache that handles cache expiration and transformation, and the backend cache that handles the reads and writes when requested to by the frontend. Here is an example of a cache service. We could simply add this to our `service.php` file located at `app/config`.

```
$di->set(
    'viewCache', function () use ($config) {

        //Cache for one day
        $frontCache = new \Phalcon\Cache\Frontend\Data(array(
            "lifetime" => 86400
        ));

        //Set file cache
        $cache = new Phalcon\Cache\Backend\File($frontCache, array(
            "cacheDir" => $config->application->cacheDir
        ));

        return $cache;
    }
);
```

We are pretty used to setting up new Phalcon services by now. In this new service, we use the configuration variable so that it can use our cache directory setting. We give the frontend cache a lifetime of one day and feed this variable to our backend cache along with the location of our cache directory.

We used a file-based cache in this service. It is not the fastest of caching mechanisms and is probably the slowest, but it is easy to set up and requires no other software. However, you are not limited to file-based backends with Phalcon. As long as you have the required software and PHP extensions installed, you can use any one of the following as a backend cache in Phalcon:

- File
    - Memcached
    - APC
    - MongoDB
    - Xcache

Phalcon has a variety of frontend cache adapters too. In the code for our cache service, we specified a data adapter that serializes our data before saving it, but you can also use one of the following frontend adapters:

- Output
- JSON
- IgBinary
- Base64
- None

Using a Phalcon cache

Now that we have our cache service set up, we can use it wherever we need it. First, you need a cache key. All Phalcon caches use a key to store and access cached data. It needs to be unique to that piece of data. A good place for a function to generate keys for the cache in our application would be in the `ControllerBase.php` file located at `app/controllers`. We can add the following function to the `ControllerBase` class so that all of our controllers inherit it:

```
public function createKey($controller, $action, $parameters = array())
{
        return urlencode($controller.$action.serialize($parameters));
}
```

This function will simply create a unique key for us. So, now we test the function using our cache by opening up the `PostController.php` file located at `app/controllers` and modifying the `showAction` function. We can access the cache service we created by adding the following line of code at the beginning of the function:

```
$cache = $this->di->get("viewCache");
```

Then, by executing the following line of code, you can access the cached content or start the cache:

```
$key = $this->createKey('posts', 'show', array($id));
$post = $cache->get($key);
```

Then, we check if we have content, and if we don't, we get it from the database and save it in the cache using our key.

```
if ($post === null) {
            $post = Posts::findFirstByid($id);
            $cache->save($key, $post);
}
```

The rest of the function remains the same.

```
$this->tag->prependTitle($post->title . " - ");
$this->view->post = $post;
```

Open up one of your blog's posts in the browser and refresh the page. You will find the cache file in the `cache` folder located at `app`. Inside the file, serialized data will be present, representing the `$post` object. To learn more about using caching in Phalcon, visit `http://docs.phalconphp.com/en/latest/reference/cache.html`.

You can also cache data at the model level. The mechanism is the same. Try to access the cached data by a key. Check if there is data. If not, execute the database call to retrieve the data and cache it. Then, return the result. To learn about caching in the ORM, visit `http://docs.phalconphp.com/en/latest/reference/models-cache.html`.

Routing in Phalcon

Before we take a look at other application structures in Phalcon, especially the micro application, we should take a look at routing. Up to now, we just have the routes in our application; just magically map to our controllers and the actions in them and let Phalcon do all of the thinking about routing for us. This is because with the single MVC structure that we are using, the routing is in MVC mode. We also have the option of match-only mode. You will notice that our blog application has an index page that does nothing. It still has the default Phalcon message. The bulk of our site currently resides at `http://localhost/phalconBlog/posts`. Well, a hacky way to fix this that we are going to use to learn about routing in Phalcon is to use a router. So, we need to add another service to our Dependency Injection container. Open up the `services.php` file located at `app/config` and add the following lines of code at the bottom:

```
$di->set(
    'router', function () {
        $router = new Router();
        $router->add(
            "/", array(
                'controller' => 'posts',
                'action' => 'index',
            )
        );
        return $router;
    }
);
```

When we add a route to the MVC router, the first parameter is the path and the second is any array that maps this path to a module, controller, and an action.

Match-only mode routing is used in the micro application example coming up. To learn more about routing in Phalcon, visit `http://docs.phalconphp.com/en/latest/reference/routing.html`.

Other project types of Phalcon

We created one type of application in Phalcon, a single MVC application. But, the type of application structure does not fit all types of projects. We will now take a look at a few other types of applications you can build with Phalcon. Of course, Phalcon being a very loosely-coupled framework, you are not limited by just these structures, and you can structure your projects however you choose.

Multimodule applications

As an application grows in size, it may be easy to organize the code into modules. Instead of an `app` folder in a project, you have multiple folders under an `apps` folder, each having their own sets of models, views, and controllers. Routing in MVC mode is already set up to handle modules. There are only a few extra steps involved. To learn more about Phalcon multimodule applications, visit `http://docs.phalconphp.com/en/latest/reference/applications.html#multi-module`.

Micro applications

For even simpler applications than the one we wrote, using a micro framework structure may make more sense. A micro application in Phalcon can be as simple as the following code snippet:

```
<?php

$app = new Phalcon\Mvc\Micro();

$app->get(
    '/user/{name}', function ($name) {
        echo "<h1>Hi $name!</h1>";
    }
);

$app->get(
    '/api/user/{name}', function ($name) {
        echo json_encode(array("message" => "Hi ". $name));
    }
);

$app->handle();
```

There is not much to this application. First, we create an instance of `Phalcon\MVC\Micro` and then we defined our routes. So, when a visitor visits the `/user/Stephan` page of the application, they are greeted with **Hi Stephan**. The second route shows one of the perfect uses of a micro application for an API. Most simple APIs don't need complex controllers. This example is here to show you how small a working Phalcon application can be. It uses routing in the match-only mode. As a micro application grows in size, you can use more of the features it supports, including models, redirects, and services. To learn more about Phalcon micro applications, visit `http://docs.phalconphp.com/en/latest/reference/micro.html`.

Command-line applications

Sometimes, you may not need a complete web application to get the job done. For scripts that you run from a command line or cron, you just need a simple structure to organize the functionality of your code. A Phalcon command-line application's structure can start as small as the following code snippet:

```
app/
  tasks/
  cli.php
```

The bootstrap file is `cli.php`, and it starts out in pretty much the same manner that all command-line applications start in Phalcon.

```
<?php

use Phalcon\DI\FactoryDefault\CLI as CliDI,
    Phalcon\CLI\Console as ConsoleApp;

//Using the CLI factory default services container
$di = new CliDI();

// Define the application path
defined('APPLICATION_PATH')
  || define('APPLICATION_PATH', realpath(dirname(__FILE__)));

//Register the autoloader
$loader = new \Phalcon\Loader();
$loader->registerDirs(
    array(
        APPLICATION_PATH . '/tasks'
    )
);
$loader->register();

// Load the config
$config = include APPLICATION_PATH . '/config/config.ini';
$di->set('config', $config);

//Create a console application
$console = new ConsoleApp();
$console->setDI($di);
```

```
//Process console arguments
$arguments = array();
$params = array();

foreach($argv as $k => $arg) {
    if($k == 1) {
        $arguments['task'] = $arg;
    } elseif($k == 2) {
        $arguments['action'] = $arg;
    } elseif($k >= 3) {
        $params[] = $arg;
    }
}
if(count($params) > 0) {
    $arguments['params'] = $params;
}

// define global constants for the current task and action
define('CURRENT_TASK', (isset($argv[1]) ? $argv[1] : null));
define('CURRENT_ACTION', (isset($argv[2]) ? $argv[2] : null));

try {
    // handle incoming arguments
    $console->handle($arguments);
}
catch (\Phalcon\Exception $e) {
    echo $e->getMessage();
    exit(255);
}
```

All that bootstrap file has to do is process commands and choose the right task and action. We are storing our tasks in the `tasks` folder. A command-line application must have at least a `mainTask` and `mainAction`. So, we can add this file to the `tasks` folder and name it `MainTask.php`.

```
<?php

class mainTask extends \Phalcon\CLI\Task
{

    public function mainAction() {
        echo "I am a task that doesn't do much";
    }

    public function anotherAction(array $params) {
        foreach($params as $p){
            echo "Hi, my name is ".$p.PHP_EOL;
        }
    }
}
```

The `mainAction` function of the main task will be run while executing the following command line:

```
php app/cli.php
```

To call a specific task and action, just assign the task as the first parameter, the action as the second, and any of the parameters (`tom`, `dick`, and `harry`) can be handled by the action function called `Executing`:

```
php app/cli.php main another tom dick harry
```

The output is as follows:

```
Hi, my name is tom
Hi, my name is dick
Hi, my name is harry
```

To learn more about creating command-line applications with Phalcon, visit `http://docs.phalconphp.com/en/latest/reference/cli.html`.

Summary

In this chapter, we learned more about what we can do with the Phalcon framework while adding features to our blog application. It is not the most feature-packed blog, but we showed how to use Phalcon tag helpers to set our document type and generate dynamic titles for our pages. We also explained how to control user access, hash passwords, and set cookies in Phalcon. Then, we broke up our views into reusable bit-sized pieces using view partials. Just in case our blog ever gets any traffic, we dabbled a bit with caching in Phalcon. We also learned how to use routing in Phalcon. Finally, we learned other structures we can use for a Phalcon application. To learn more about Phalcon, visit `http://phalconphp.com`.

Index