Community Experience Distilled

Learn Phalcon interactively and build high-performance web applications

Learning Phalcon PHP

Calin Rada

This book covers the most common and useful parts of Phalcon PHP, which will guide you to make the right decisions while developing a Phalcon-driven application.

You will begin the journey by installing and setting up Phalcon for your environment followed by the development of each module. You will be introduced to Phalcon’s ORM and ODM. Furthermore, you will also be able to create the first models and the database architecture for your project. You will then cover the command-line applications, API module, volt syntax, and hierarchical views. Installing and working with Node and Bower for assets management will also be covered. Finally, you will gain insights into creating the Backoffice and Frontend module along with best practices and resources for development with Phalcon PHP.

By the end of this book, you will be able to confidently develop any kind of application using the Phalcon PHP framework in a short time.

Who this book is written for

If you are a web developer and want to build effective web applications with Phalcon PHP, then this book is ideal for you. The book does not assume detailed knowledge of PHP frameworks.

What you will learn from this book

- Compile and install Phalcon from source code
- Set up an MVC project and learn how to use the Dependency Injection (DI), configuration and routing
- Get to grips with Phalcon’s ORM, ODM, validation and caching, and develop models needed for your application
- Create a command-line application from scratch
- Write a fully functional REST module with Phalcon PHP
- Use Volt as a template engine to create your first views
- Understand how a session works and learn how to implement an authentication system

By the end of this book, you will be able to confidently develop any kind of application using the Phalcon PHP framework in a short time.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Phalcon'
- A synopsis of the book’s content
- More information on Learning Phalcon PHP
About the Author

Calin Rada is a full-stack developer with over 10 years of experience in web development; system architecture; database modeling; the setup, configuration, and administration of servers and hosting systems; and understanding customer and business needs. He is always interested in learning new things and working with new technologies.
Preface

Phalcon is the fastest PHP framework out there, and it is delivered as a C extension. More than that, you will find it very easy to learn. This book will present in detail the most common and useful parts of Phalcon PHP, as well as guide you to lean towards making the right decision when developing a Phalcon-driven application.

*Learning Phalcon PHP* is an interesting journey that starts with guides for installing the required software and preparing the working environment and project structure, and continues with a step-by-step approach development of each module.

By the end of this book, you will have developed a simple but fully functional news website and gained advanced knowledge on how Phalcon works.

What this book covers

*Chapter 1, Getting Started with Phalcon,* introduces the Phalcon Framework. In this chapter, you learn how to install and configure Phalcon.

*Chapter 2, Setting Up the MVC Structure and the Environment for Our Project,* helps you get to grips with the basics of MVC (Model-View-Controller) and setting up the work environment.

*Chapter 3, Learning Phalcon’s ORM and ODM,* is about Phalcon’s ORM (Object-relational Mapping) and ODM (Object-Document Mapper). You learn how to connect to a database and create models and relations between them.

*Chapter 4, Database Architecture, Models, and CLI Applications,* teaches you how to create the database architecture and the models needed for our project. You also learn about Phalcon CLI and develop a simple CLI application.

*Chapter 5, The API Module,* helps you start the development of a RESTful API module.
Preface

Chapter 6, Assets, Authentication, and ACL, explains assets management (JavaScript files, style sheets, and images), and you create a simple authentication system based on an ACL (access control list).

Chapter 7, The Backoffice Module (Part 1), shows you how to develop CRUD operations. This first part is about CRUD for categories and hashtags.

Chapter 8, The Backoffice Module (Part 2), is a continuation of the previous chapter. Here, you develop CRUD operations for users and articles.

Chapter 9, The Frontend Module, helps you develop the frontend template. You learn how to implement Elasticsearch and Mongo to improve the speed of your application.

Chapter 10, Going Further, teaches you common operations, such as file uploads and annotations.
1

Getting Started with Phalcon

What is Phalcon? Let’s start by quoting from the documentation of the official website (http://phalconphp.com):

"Phalcon is an open source, full stack framework for PHP written as a C-extension, optimized for high performance."

Version 2.0 of Phalcon was released in April, and it was developed with a new language called Zephir (http://zephir-lang.com/). Zephir was designed especially for developing PHP extensions, and it is quite user friendly for both (PHP and C) developers.

There are many frameworks out there. The main reasons why we choose Phalcon were for its steep learning curve, speed, and because it is decoupled. (We can use any of its components independently.) If you have some knowledge of the Model-View-Controller (MVC) and some experience with any Object-Relational Mapping (ORM), you will find working with it pretty straightforward.

We will start our journey with this first chapter where we will:

• Configure our web server
• Install Phalcon
• Discuss a bit about how Phalcon works

Before starting, we assume that you are using a *nix environment. Personally, I feel comfortable with Debian distributions, especially Ubuntu, which I am using on a daily basis; so, the installations steps that we will talk about are for Ubuntu. The OS is a matter of personal choice, but I highly recommend any *nix distribution for development. (Even Microsoft decided to open source their ASP.NET for Linux early this year)

For other types of OS, you will have to search their official documentation, in terms of "how to". This book is intended to be about Phalcon and tutorials on installing different software on different kinds of OS are out of the scope of this book.
Getting Started with Phalcon

Here is the list of URLs that contain installation instructions for different operating systems:


Senior developers might not agree with me on certain subjects or certain techniques and/or recommendations. In general, as a developer, I think you should analyze what is suitable for you and develop a platform according to your (or client) requirements. In addition, most importantly, there is no such thing as "The Perfect Solution". There is always room for improvement.

Installing the required software

We need to install the following software that we are going to use in this book:

- PHP
- Nginx and Apache
- MongoDB
- MySQL
- GIT
- Redis
- Phalcon

Installing PHP

You have probably already installed PHP on your system since you are reading this book. However, just in case you haven't, here are the simple steps to quickly install the latest PHP version (Phalcon is running on PHP version >= 5.3). I recommend you to use the Personal Package Archive (PPA) from Ondřej Surý (https://launchpad.net/~ondrej/+archive/ubuntu/php5) because it has the latest PHP version available on it:

$ sudo add-apt-repository ppa:ondrej/php5
$ sudo apt-get update
If you don't want to use this step, you can simply install PHP from the official repositories:

$$\texttt{sudo apt-get install php}$$

Apache will be installed by default with PHP. However, if you want Nginx instead of Apache, you must install PHP in a certain order.

The following command will **automatically install PHP and Apache**. If you don't need/want to use Apache, please skip using this command:

$$\texttt{sudo apt-get install php5 php5-fpm}$$

To avoid Apache installation, execute the following commands in the exact same order:

$$\texttt{sudo apt-get install php5-common}$$
$$\texttt{sudo apt-get install php5-cgi}$$
$$\texttt{sudo apt-get install php5 php5-fpm}$$

The `php5-cgi` package fulfills the dependencies that would otherwise be fulfilled by Apache.

### Installing Nginx

To install the Nginx web server, we need to execute the following commands:

$$\texttt{sudo add-apt-repository ppa:nginx/stable}$$  
$$\texttt{sudo apt-get update}$$  
$$\texttt{sudo apt-get install nginx}$$

### Installing MySQL

MySQL is probably the most widely spread RDBMS system with a market share that is greater than 50 percent. Since we are going to use it to develop our project, we need to install it by executing the following command:

$$\texttt{sudo apt-get install mysql-server}$$
Installing Redis

Redis is an advanced key-value storage/cache system. We are going to use this mostly for our session and to cache objects to improve the speed of our application. Let's install it by executing the following commands:

$ sudo add-apt-repository ppa:chris-lea/redis-server
$ sudo apt-get update
$ sudo apt-get install redis-server
$ sudo apt-get install php5-redis

Installing MongoDB

MongoDB is a document database (NoSQL database) system. We will use this to store data that is accessed frequently. Let's install it:

$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
$ echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
$ sudo service mongodb start
$ sudo apt-get install php5-mongo

Installing Git

Git is a distributed version control system that we will use to track changes to our application and much more. We will install Git by executing the following command:

$ sudo apt-get install git

[!] I strongly recommend that you use the latest versions of all software as much as possible.

Installing Phalcon

Now that we have installed all the required software, we will proceed with the installation of Phalcon. Before we continue, we must install some dependencies:

$ sudo apt-get install php5-dev libpcre3-dev gcc make php5-mysql
For Windows systems and more details about how to compile the extension on different systems, please check the latest documentation at http://phalconphp.com/en/download.

Now, we can clone the repository and compile our extension:

```
$ git clone --depth=1 git://github.com/phalcon/cphalcon.git
$ cd cphalcon/build
$ sudo ./install
$ echo 'extension=phalcon.so' | sudo tee /etc/php5/mods-available/phalcon.ini

$ sudo php5enmod phalcon
$ sudo service php5-fpm restart
```

If everything goes well, you should be able to see Phalcon in the list of PHP installed modules:

```
$ php -m | grep phalcon
```

---

**The Apache and Nginx configuration files**

We will use /var/www/learning-phalcon.localhost as the default directory for our project, and we will refer to it as the **root folder**. Please create this folder:

```
$ sudo mkdir -p /var/www/learning-phalcon.localhost/public
```

Of course, if you want, you can use another folder. Let's create a test file in our public folder under the root directory with some PHP content:

```
$ cd /var/www/learning-phalcon.localhost/public
$ echo "<?php date();" > index.php
```

**Apache**

Let's switch to the default directory where Apache holds the configuration files for the available websites, using the command line:

```
$ cd /etc/apache2/sites-available/
```

After that, perform the following set of steps:

1. Using your favorite editor, create a file named learning-phalcon.localhost for apache version < 2.4 or learning-phalcon.localhost.conf for apache version >= 2.4:

   ```
   $ vim learning-phalcon.localhost.conf
   ```
2. Now, paste the following content to this file:

```xml
<VirtualHost *:80>
    DocumentRoot "/var/www/learning-phalcon.localhost"
    DirectoryIndex index.php
    ServerName learning-phalcon.localhost
    ServerAlias www.learning-phalcon.localhost

    <Directory "/var/www/learning-phalcon.localhost/public">
        Options All
        AllowOverride All
        Allow from all
    </Directory>
</VirtualHost>
```

3. Then, switch to the public folder and add a file named `.htaccess` to it:

```bash
$ cd /var/www/learning-phalcon.localhost/public
$ vim .htaccess
```

4. Then, add the following content to the `.htaccess` file:

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

5. This will not work unless you have enabled `mod_rewrite`. To do so, execute this command:

```bash
$ sudo a2enmod rewrite
```

6. Now that we have configured our virtual host, let's enable it:

```bash
$ sudo a2ensite learning-phalcon.localhost
$ sudo service apache2 reload
```

### The host file

If you open a browser and type `http://www.learning-phalcon.localhost/`, you'll receive a host not found or connection error. This is because there is no name resolver for this TLD (short for **Top Level Domain**). To fix this, we edit our host file and add this name:

```bash
$ echo "127.0.0.1 learning-phalcon.localhost www.learning-phalcon.localhost" | sudo tee /etc/hosts
```
Restart your browser and type the address http://www.learning-phalcon.localhost/ again. If everything goes well, you should see the current date/time.

**Nginx**

If you choose to use Nginx (which I recommend, especially because it can serve more concurrent clients with higher throughput, and it serves static content more efficiently) instead of Apache, here is what you need to do:

Locate the config folder of Nginx (in Ubuntu, it is installed under /etc/nginx/).

Create a file named learning-phalcon.localhost in your sites-available folder (by navigating to /etc/nginx/sites-available):

```
$ cd /etc/nginx/sites-available
$ vim learning-phalcon.localhost
```

Now, add the following content to it:

```server {
    listen 80;
    server_name learning-phalcon.localhost;

    index index.php;
    set $root_path "/var/www/learning-phalcon.localhost/public";
    root $root_path;

    client_max_body_size 10M;

    try_files $uri $uri/ @rewrite;

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

    location .\.* {
        fastcgi_index /index.php;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_intercept_errors on;
        include fastcgi_params;

        fastcgi_split_path_info ^(.+\.php)(/.*);$;

        fastcgi_param PATH_INFO $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED
            $document_root$fastcgi_path_info;
```
Getting Started with Phalcon

In some environments, you might need to edit your `php.ini` file and set `cgi.fix_pathinfo = 0`.

Then, save the file and restart Nginx:

```
$ sudo service nginx restart
```

Please edit and save your host file (check `The host file section`), then open your browser and type `http://www.learning-phalcon.localhost/`. At this point, you should see a page that shows the current date/time.

There are many possible methods to install and configure PHP and Apache/Nginx. Feel free to do a simple Google search and choose one that fits you better, if my method is not the optimal one for your needs.

Assuming that everything went well until now, we will go further by learning a little bit about Phalcon's internals.

**Understanding the framework's internals**

In this section, I will try to make a short introduction to the common parts of the framework. Most of the text presented here is part of the official documentation that you should always read. The idea of this section is to make you familiar with the most common methods and components that will help you to understand quickly how the framework works.
The dependency injection

Probably one of the most powerful characteristics of Phalcon is the dependency injection (DI). If you have no idea about dependency injection, you should read at least the wiki page for this design pattern at http://en.wikipedia.org/wiki/Dependency_injection:

"Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is the passing of a dependency (a service or software module) to a dependent object (a client). The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Dependency injection allows a program design to follow the dependency inversion principle.

The term "Dependency injection" was coined by Martin Fowler.

A real-life example of dependency injection might be the following situation: Suppose you go shopping. At the mall, you will need a bag to put your groceries, but you forgot to take one when you left your home. In this case, you will need to buy a bag. In development, buying this bag can be quite expensive. So, what if your door has a scanner that scans your body for a bag, and will not open unless you have one? This can be called dependency injection.

Phalcon uses the `Phalcon\DI` component, which is a component that implements the Inversion of Control pattern. This reduces the overall code complexity.

The framework itself or the developer can register services. Phalcon has many built-in components that are available in the DI container, such as the following ones:

- Request and response
- Logger
- Crypt
- Flash
- Router and configuration
Setting up a new component in the DI is as easy as the following code:

```php
<?php
$di = new Phalcon\DI();
// Lazy load
$di['mail'] = function() {
    return new \MyApp\Mail();
};
```

When you need to access the "mail" component, in a controller for example, you can simply call it:

```php
<?php
$mail = $this->getID()->get('mail');
// or
$mail = $this->getDI()->getMail();
```

If you need to create your own DI, Phalcon or the DiInterface interface must be implemented to replace the one provided by Phalcon, or you must extend the current one.

These are just a few dummy examples so that you can have an idea about Phalcon's DI by the time we start our project. In the meanwhile, please take your time and read the official documentation that can be found at [http://docs.phalconphp.com/en/latest/reference/di.html](http://docs.phalconphp.com/en/latest/reference/di.html).

**The request component**

The request component is probably one of the most used components in any framework. It handles any HTTP request (such as GET, POST, or DELETE, among others) and also provides a few shortcuts for the `$SERVER` variable. Most of the time, we will use the request component in the controllers. The Phalcon documentation ([http://docs.phalconphp.com/en/latest/reference/mvc.html](http://docs.phalconphp.com/en/latest/reference/mvc.html)) states the following:

"The controllers provide the "flow" between models and views. Controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation."

---

[10]
In Phalcon, all controllers should extend the `Phalcon\Mvc\Controller` component, and the name of the public methods that we want to access via HTTP GET should have the suffix `Action`. For example:

```php
<?php

class ArticleController extends \Phalcon\Mvc\Controller
{
    // Method for rendering the form to create an article
    public function createAction()
    {
    }

    // Method for searching articles
    public function searchAction()
    {
    }

    // This method will not be accessible via http GET
    public function search()
    {
    }
}
```

Okay. So, how do we use the request component? Easy! Do you remember that we talked about built-in components in the DI section? The request component is one of them. All we need to do is get the DI. Here is an example of how to get and use the request component:

```php
<?php

class ArticleController extends \Phalcon\Mvc\Controller
{
    public function searchAction()
    {
        $request = $this->getDI()->get('request');
        // You can also use $request = $this->request; but I don't
        // recommend it because $this->request can be easily overwritten
        // by mistake and you will spend time to debug ... nothing.

        $request->getMethod(); // Check the request method
        $request->isAjax(); // Checks if the request is an ajax request
        $request->get(); // Gets everything, from the request (GET, POST, DELETE, PUT)
```
$request->getPost(); // Gets all the data submitted via POST method
$request->getClientAddress(); // Return the client IP

These are just a few common methods that are built into the request component. Let's continue with the next important component—Response.

**The response component**

So, what can this component do? Well, pretty much everything that is response or output related. Using it, we can set headers, do redirects, send cookies, set content, and much more. Here is a list of common methods from this component:

```php
public function testRedirectAction()
{
    $response = $this->getDI()->get('response');
    // or you can use $this->response directly
    // Redirect the user to another url
    $this->view->disable();
    return $response->redirect('http://www.google.com/', true);
}
```

The `redirect` method accepts three parameters: a location (string), if it is an external redirect (this is a Boolean type which is by default `false`), and a status code (http status code range). The following lines of code is the redirect method:

```php
/**
 * Redirect by HTTP to another action or URL
 * @param string $location
 * @param boolean $externalRedirect
 * @param int $statusCode
 * @return \Phalcon\Http\ResponseInterface
 */
public function redirect($location, $externalRedirect, $statusCode);
```
Another useful method is the `setHeader` method:

```php
public function testSetHeaderAction()
{
    $this->response->setHeader('APIKEY', 'AWQ23XX258561');
}
```

The preceding example sets a header named `APIKEY` with the value as `AWQ23XX258561`. Sending headers is a common approach when you develop APIs. You can send any type of headers and overwrite current headers using this method.

Content related methods: `setContent()` and `setJsonContent()`. Let's take for example the following code:

```php
public function testContentAction()
{
    // First, we disable the view if there is any
    $this->view->disable();

    // Set a plain/text or html content
    $this->response->setContent('I love PhalconPHP');

    // OR

    // Set a json content (this will return a json object)
    $this->response->setJsonContent(array(
        'framework' => 'PhalconPHP',
        'versions' => array(
            '1.3.2',
            '1.3.3',
            '2.0.0'
        )
    ));

    // We send the output to the client
    return $this->response->send();
}
```
Getting Started with Phalcon

When you need to send any JSON content, you should set the header as `application/json` using the built-in method in the response object:

```php
$this->response->setContentType('application/json', 'UTF-8');
```

Now that we know the basics about response/request components, we might find ourselves in a situation where we may need to log different things, such as errors. For this, we need to check the logger component.

### The logger component

In a production environment, we cannot afford to throw errors or blank pages at the client. We will avoid this and log the errors in a log file. You will read more about this in the next chapters. To sum it up, we will implement a custom logger to our DI, catch exceptions, and then log them. For example, perform the following set of steps:

1. Set the custom logger in DI using the following code:
   ```php
   $di['logger'] = function() {
       $error_file = __DIR__.'/../logs/'.date("Ymd_error").'log';
       return new \Phalcon\Logger\Adapter\File($error_file,
       array('mode' => 'a+'));
   };
   ```

2. Create a method that will throw an exception, catch it, and log it, as follows:
   ```php
   public function testLoggerAction()
   {
       try {
           $nonExistingComponent = $this->getDI()->get('nonExistingComponent');
           $nonExistingComponent->executeNonExistingMethod();
       } catch (\Exception $e) {
           $this->logger->error($e->getMessage());
           return $this->response->redirect('error/500.html');
       }
   }
   ```
In the preceding example, we try to execute a nonexistent method, and our code will throw an exception that we catch. It will log it and then redirect the user to a friendly error page, error/500.html. You will notice that our logger component calls a method named error. There are other methods that are implemented, such as, debug, info, notice, warning, and so on.

The logger component can be transactional. (Phalcon stores the logs temporarily in memory, and later on, it writes the data to the relevant adapter.) For example, consider the following code snippet:

```php
<?php
    $this->logger->begin();
    $this->logger->error('Ooops ! Error !');
    $this->logger->warning('A warning message');
    $this->logger->commit();
```

The crypt component

Crypt is a very useful component if someone needs to encrypt data and decrypt it on your side. One situation where you might want to use the crypt component is to send data over the HTTP get method or save sensitive information in your database.

This component has many built-in methods such as encrypt, decrypt, getAvailableChipers, setKey, getKey, and so on. Here is an example of using the crypt component in the HTTP get method.

First, we overwrite the DI, and then we pass a key to it in order to avoid setting it every time:

```php
<?php
    $di['crypt'] = function () {
        $crypt = new \Phalcon\Crypt();
        $crypt->setKey('0urSup3rS3cr3tK3y!?');

        return $crypt;
    };

    public function sendActivationAction()
    {
        $activation_code = $this->crypt->encryptBase64('1234');
    }
```
Of course, you are probably never going to use it this way. The preceding example just demonstrates the power of this component. You might have noticed that there is a new DI method called flash. We are going to talk about it next.

The flash component

This component is used to send notifications to the client and inform him or her about the status of the component's actions. For example, we can send a successful message after a user has completed the registration on our website or submitted a contact form.

There are two kinds of flash messages—direct and session—and both are available in DI. The direct method outputs the message directly and cannot be loaded on a future request. On the contrary, the session method, stores the messages in a session, and they are automatically cleared after they are printed.

Here is a common usage of flash direct and flash session, assuming that you have a page called register, and you post the data on the same page:

```php
public function registerAction()
{
    // ... code
    if ($errors) {
        $this->flash->warning('Please fix the following errors: ');
        foreach($errors as $error) {
            $this->flash->error($error);
        }
    } else {
        $this->flash->success('You have successfully registered on our website');
    }
}
```
In our view, we will render the messages using the `getText()` method or `content()` in the template engine Volt (we’ll cover this later in the chapter).

If we need to redirect our user to another page (let’s call it `registerSuccess`), then we need to use the flash session method; otherwise, the message will not appear.

```php
public function registerAction()
{
    // render our template
}
```

The `register` template will contain a form with method `post` and `action` pointing to the `create` method. The `create` method will look something like this:

```php
public function createAction()
{
    if ($errors) {
        $this->flashSession->warning('Please fix the following errors: ');
        foreach($errors as $error) {
            $this->flashSession->error($error);
        }
    } else {
        $this->flashSession->success('You have successfully registered on our website');
    }

    return $this->response->redirect('/register');
}
```

In the preceding example, we set the messages in the session using the `flashSession` method, and we redirect the user back to the register page. In order to render the messages in our view, we need to call the method `flashSession()`-

>output();.

The recommended way is to forward the request with the help of dispatcher, not using redirects. If you use redirects, the user will lose all the data that he or she filled in the form.
The router component

The router component helps us to map friendly URLs to our controllers and actions.

By default, if the rewrite module is enabled in your web server, you will be able to access a controller named Post and the read action like this: http://www.learning-phalcon.localhost/post/read. Our code can look like this:

```php
class PostController extends \Phalcon\Mvc\Controller
{
    public function readAction()
    {
        // get the post
    }
}
```

However, sometimes, this code is not apt if you need to translate the URLs into multiple languages, or if you need to name the URLs in a different way to how they are defined in the code. Here is a usage example for the router component:

```php
$router = new \Phalcon\Mvc\Router();
// Clear the default routes
$router->clear();

$st_categories = array('entertainment', 'travel', 'video');

$s_categories = implode('|', $st_categories);

$router->add('#^/([^/])[/\]{0,1}$#', array(
    'module' => 'frontend',
    'controller' => 'post',
    'action' => 'findByCategorySlug',
    'slug' => 0
));
```
In the preceding example, we map all the categories to the controller post and action findByCategorySlug. The router component allows us to use regular expressions for our URLs. With preg_match, this can be represented as follows:

```php
$url = 'http://www.learning-phalcon.localhost/video';
preg_match('#^/(entertainment|travel|video)/[/]{0,1}$#', $url);
```

By accessing http://www.learning-phalcon.localhost/video, the request will be forwarded to the findByCategorySlug action from the post controller:

```php
<?php
class PostController extends \Phalcon\Mvc\Controller
{
    public function findByCategorySlug()
    {
        $slug = $this->dispatcher->getParam('slug', array('string', 'striptags'), null);
        // We access our model (entity) to get all the posts from this category
        $posts = Posts::findByCategorySlug($slug);
        if ($posts->count() > 0) {
            $this->view->setVar('posts', $posts);
        } else {
            throw new \Exception('There are no posts', 404);
        }
    }
}
```

The `getParam()` method has three parameters. The first one is the name that we are searching for, the second parameter is an array of filters that can be applied automatically, and the third parameter is the default value in case the requested name does not exist or is not set.

We will discuss models in the next chapter. This was just a simple example of how you can use the router.

The router also supports a precheck of the request method. You may be used to check whether the method is POST, DELETE, PUT, or GET, like this:

```php
<?php
if ($_SERVER['REQUEST_METHOD'] == 'post') {
    // process the information
}
```
While this is perfectly correct, it is not very friendly for our code. Phalcon's router has this capability by which you can add the right type of request that you are expecting, without the need to check this in your code:

```php
// Add a get route for register method within the user controller
$router->addGet('register', 'User::register');

// Add a post route for create method, from the user controller
$router->addPost('create', 'User::create');
```

This is the basic usage of the router. As always, please read the documentation in order to learn everything about this component.


**The config component**

This component can handle configuration files of various formats by using adapters. Phalcon has two built-in adapters for it, which are INI and Array. Using INI files is probably never a good idea. Therefore, I recommend you to make use of native arrays.

What kind of data can or needs to be stored in these files? Well, pretty much everything that will be needed globally in our application, such as database connection parameters. In the old days, we used `$_GLOBALS` (a big security issue), or we used the `define()` method, and then gradually we started using it globally.

Here is an example of a config file, and how we can use it:

```php
$st_settings = array(
    'database' => array(
        'adapter'  => 'Mysql',
        'host'     => 'localhost',
        'username' => 'john',
        'password' => 'johndoe',
        'dbname'   => 'test_database',
    ),
    'app' => array(  
```
The config object can be converted back to an array by using `toArray()` method:

```php
<?php
$st_config = $config->toArray();
echo $config['app']['name']; // Will output Learning Phalcon
```

Another useful method for this object is the `merge` method. If we have multiple configuration files, we can easily merge them into one object:

```php
<?php
$config = array(
    'database' => array(
        'adapter' => 'Mysql',
        'host' => 'localhost',
        'dbname' => 'test_database',
    ),
    'app' => array(
        'name' => 'Learning Phalcon'
    )
);

$config2 = array(
    'database' => array(
        'username' => 'john',
        'password' => 'johndoe',
    )
);

$config = array_merge($config, $config2);
```

Now, the $config object will have the same content as it did before.

There are two other adapters that are not implemented yet (YAML and JSON), but you can use them if you clone Phalcon’s incubator repository (https://github.com/phalcon/incubator). This repository contains a collection of adapters/helpers that might be integrated in Phalcon in the near future.
The view component

This component is used to render our templates. By default, the templates have the `.phtml` extension, and they contain HTML and PHP code. Here are some examples on how to use the view:

1. First, we set up the view in the DI using the following code snippet:

   ```php
   <?php
   $di['view'] = function () use ($config) {
       $view = Phalcon\Mvc\View();
       // Assuming that we hold our views directory in the configuration file
       $view->setViewsDir($config->view->dir);
       return $view;
   };
   ```

2. Now, we can use this service as follows:

   ```php
   class PostController extends Phalcon\Mvc\Controller
   {
       public function listAction()
       {
           // Retrieve posts from DB
           $posts = Post:find();
           $this->view->setVar('pageTitle', 'Posts');
           $this->view->setVar('posts', $posts);
       }
   }
   ```

3. Next, we need to create a view template that must look like this:

   ```html
   <!DOCTYPE html>
   <html>
   <head>
   <meta charset="UTF-8">
   <title><?php echo $pageTitle; ?></title>
   </head>
   <body>
   <?php foreach($posts as $post) { ?>
   <p><?php echo $post->getPostTitle(); ?></p>
   <p><?php echo $post->getPostContent(); ?></p>
   <?php } ?>
   </body>
   ```
Simple, isn't it? This component also supports hierarchical rendering. You can have a base layout, a general template for posts, and a template for a single post. Let's take, for example, the following directory structure:

```
app/views/
- index.phtml
- post/detail.phtml
```

Phalcon will first render `app/views/index.phtml`. Then, when we request for `detailAction()` from the post controller, it will render `app/views/post/details.phtml`. The main layout can contain something similar to this code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Learning Phalcon</title>
</head>
<body>
<?php echo $this->getContent(); ?>
</body>
</html>
```

And, the `details.phtml` template will have the following content:

```
<?php foreach($posts as $post) { ?>
<p><?php echo $post->getPostTitle(); ?></p>
<p><?php echo $post->getPostContent(); ?></p>
<?php } ?>
```

This component also allows you to pick different templates to set a render level, disable or enable the view, and much more.

Phalcon has a built-in template engine named Volt. If you are familiar with PHP template engines such as Smarty or Twig, you will want to use them for sure. Volt is almost identical to Twig, and you will find it very useful—it is inspired by Jinja (http://jinja.pocoo.org/). You can even use your own template engine, or any other template engine that you can find there.
In order to enable the Volt template engine, we need to make a small modification to our view service, and we need to create a Volt service; here is how to do this:

```php
<?php
$di['voltService'] = function($view, $di) use ($config) {
    $volt = new \Phalcon\Mvc\View\Engine\Volt($view, $di);
    if (!is_dir($config->view->cache->dir)) {
        mkdir($config->view->cache->dir);
    }
    $volt->setOptions(array(
        "compiledPath" => $config->view->cache->dir,
        "compiledExtension" => ".compiled",
        "compileAlways" => false
    ));
    $compiler = $volt->getCompiler();
    return $volt;
};

// First, we setup the view in the DI
$di['view'] = function () use ($config) {
    $view = \Phalcon\Mvc\View();
    $view->setViewsDir($config->view->dir);
    $view->registerEngines(array(
        ".volt" => 'voltService'
    ));
    return $view;
};
```

By adding this modification and `voltService`, we can now use this template engine. From the inheritance point of view, Volt acts a little bit differently. We first need to define a main layout with named blocks. Then, the rest of the templates should extend the main layout, and we need to put our content in the same blocks as the main layout. Before we look at some examples, I will tell you a little bit about Volt's syntax, the details are as follows.

- The syntax for outputting data or for echoing content:
  ```
  {{ my_content }}
  ```
- The syntax for defining blocks:
  ```
  {% block body %} Content here {% endblock %}
  ```
• The syntax to extend a template (this should be the first line in your template):
  {% extends 'layouts/main.volt' %}

• The syntax to include a file:
  {% include 'common/sidebar.volt' %}

• The syntax to include a file and pass variables:
  {% include 'common/sidebar' with('section': 'homepage') %}

[!] Please note the missing extension. If you pass variables, you MUST omit the extension.

• The syntax for control structures (for, if, else):
  {% for post in posts %}
    {% if post.getCategorySlug() == 'entertainment' %}
      <h3 class="pink">{{ post.getPostTitle() }}</h3>
    {% else %}
      <h3 class="normal">{{ post.getPostTitle() }}</h3>
    {% endif %}
  {% endfor %}

• The syntax for the loop context:
  {% for post in posts %}
    {% if loop.first %}
      <h1>{{ post.getPostTitle() }}</h1>
    {% endif %}
  {% endif %}

• The syntax for assignments:
  {% set title = 'Learning Phalcon' %}
  {% set cars = ['BMW', 'Mercedes', 'Audi'] %}

The list is long. Additionally, you can use expressions, comparison operators, logic operators, filters, and so on. Let's write a simple template to see how it works:

```html
<!-- app/views/index.volt -->
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{% block pageTitle %}Learning Phalcon{% endblock %}</title>
</head>
```
The session component

This component provides object-oriented wrappers to access session data. To start the session, we need to add the service into the DI container:

```php
<?php

$di['session'] = function () {
    $session = new Phalcon\Session\Adapter\Files();
    $session->start();
    return $session;
};
```
The following is a code example for working with session:

```php
public function testSessionAction()
{
    // Set a session variable
    $this->session->set('username', 'john');

    // Check if a session variable is defined
    if ($this->session->has('username')) {
        $this->view->setVar('username', $this->session->get('username'));
    }

    // Remove a session variable
    $this->session->remove('username');

    // Destroy the session
    $this->session->destroy();
}
```

If you check Phalcon’s incubator, there are many available adapters, such as Redis, Database, Memcache, and Mongo. You can also implement your own adapter.


**The cache component**

To improve the performance of some applications, you will need to cache data. For example, we can cache the query results for a post. Why? Imagine 1 million views or posts. Normally, you will query the database for it, but this will mean 1 million queries (you can multiply this by at least 3, if you are using it, and for ORM—this means 3 million queries at least). Why? When you query, the ORM will act like this:

1. It'll check if the table exists, in the information schema:

   ```sql
   SELECT IF(COUNT(*)>0, 1 , 0)
   FROM `INFORMATION_SCHEMA`.`TABLES`
   WHERE `TABLE_NAME`='user'
   ```
2. Then, it'll check whether it's executing a "Describe" of the table:
   
   ```sql
   DESCRIBE `user`
   ```

3. Then, whether it's executing the actual query:
   
   ```sql
   SELECT * FROM user.
   ```

4. If the `user` table has relations, the ORM will repeat each of the preceding steps for each relation.

To solve this problem, we will save the post object into our caching system.

Personally, I use Redis and Igbinary. Redis is probably the most powerful tool, since it stores the data in memory and, saves the data on disk for redundancy. This means that every time you request the data from cache, you will get it from memory. Igbinary (https://pecl.php.net/package/igbinary) is a replacement for the standard php serializer. Here is an example cache service:

```php
<?php

$di['redis'] = function () {
    $redis = new \Redis();
    $redis->connect(
        '127.0.0.1',
        6379
    );

    return $redis;
};

-di['cache'] = function () use ($di, $config) {
    $frontend = new \Phalcon\Cache\Frontend\Igbinary(array(
        'lifetime' => 86400
    ));
    $cache = new \Phalcon\Cache\Backend\Redis($frontend, array(
        'redis' => $di['redis'],
        'prefix' => 'learning_phalcon'
    ));

    return $cache;
};
```
The cache component has the following methods that are commonly used:

```php
// Save data in cache
$this->cache->save('post', array(
    'title' => 'Learning Phalcon',
    'slug' => 'learning-phalcon',
    'content' => 'Article content'
));

// Get data from cache
$post = $this->cache->get('post');

// Delete data from cache
$this->cache->delete('post');
```

**Summary**

In this chapter, we installed the required software, created the configuration files for the web servers, and you learned a little bit about Phalcon's internals. In the next chapters, we will learn by example, and everything will be much clearer.

Take your time, and before going further, read a little bit more about anything in which you don't have experience.

In the following chapter, we will look at how to set up the MVC structure and the environment for our project.
Where to buy this book
You can buy Learning Phalcon PHP from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.