Simple

simplephp.org

A PHP Framework for Beginners

Documentation

Introduction

Simple provides a robust base for your application. It is not your average power-packed framework but a set of PHP codes that is used to layout a strong base for your application to keep your code Modular and organized.

Based on the Model-View-Controller principles, it allows you a lot of flexibility inside your application but at the same time restricts you from stepping out of the conventions which help you maintain structure.

The framework also provides a basic organizational structure, from filenames to database table names, keep ing your entire application consistent and logical. This concept is simple but powerful. Follow the conventions and you'll always know exactly where things are and how they're organized.

The best way to experience and learn Simple is to sit down and build something. To start off we'll build a simple blog application.

Blog Tutorial

Requirements:

1. A running Web Server, Apache/Ningx, etc with PHP (>=5.3.0) with mod_rewrite enabled.
2. A MySQL server.
3. Basic PHP knowledge. The more object-oriented programming you've done, the better.
4. Basic Knowledge of MVC pattern.

Once you have everything ready, Lets get started!

Getting Simple

You can also clone the repository using git (http://git-scm.com/).

```
git clone git://github.com/deepanshumehndiratta/simplephp.git
```

Regardless of how you downloaded it, place the code inside of your DocumentRoot. Once finished, your directory setup should look something like the following:

```
/path_to_document_root
      /app
      /core
      /plugins
      .htaccess
      README.md
```

Creating the Blog Database

Next, lets set up the underlying database for our blog. if you haven't already done so, create an empty database for use in this tutorial, with a name of your choice. Right now, we'll just create a single table to store our posts. We'll also throw in a few posts right now to use for testing purposes. Execute the following SQL statements into your database:

```
/* First, create our posts table: */
CREATE TABLE posts (
id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
title VARCHAR(200),
body LONGTEXT,
created INT DEFAULT 0,
modified INT DEFAULT 0
);
/* Then insert some posts for testing: */
INSERT INTO posts (title,body,created)
VALUES ('Title 1', 'Body 1.', UNIX_TIMESTAMP(NOW()));
INSERT INTO posts (title,body,created)
VALUES ('Title 2', 'Body 2.', UNIX_TIMESTAMP(NOW()));
INSERT INTO posts (title,body,created)
VALUES ('Title 3', 'Body 3.', UNIX_TIMESTAMP(NOW()));
```

The table 'posts' is automatically linked to the Model 'PostsModel' which is hooked to 'PostsController' directly.

Configuring the Database for Application

The config file for database can be found in app/config directory.

app/config/database.php

```
$database = array (
                    array (
                       'host' => 'localhost',
                       'database' => 'simpleBlog',
                       'user' => 'root',
                       'persistent' => true,
                       'passwd' => null
                    ),
                    array (
                       'host' => 'localhost',
                       'database' => 'simpleBlog',
                       'user' => 'root',
                       'persistent' => true,
                       'passwd' => null
                    ),
                    array (
                       'host' => 'localhost',
                       'database' => 'simpleBlog',
                       'user' => 'root',
                       'persistent' => true,
                       'passwd' => null
                    ),
               );
```

The 3 configurations are for the 3 modes your application can be set to in basic.php file in the same directory:

0 => Development

1 => Testing

2 => Live

By default your application is set to use the Development (0) mode.

Note: Simple does not use the PDO MySQL extension.

Checklist

1. Apache mod_rewrite is enabled.
2. Database is configured and 'posts' table has been created.
3. APP_PATH, CORE_PATH, APP_NAME and APP_FOLDER are defined in app/webroot/index.php
4. Set the name of your application in app/config/basic.php

With this, your application has been configured for basic usage. Now lets create our app.

Blog Tutorial – Adding a layer

Creating the Posts Model

The Model class is the point of interaction of our database and our application, we'll have the foundation in place needed to do our view, add, edit, and delete operations with it.

The model class files go in /app/models, and the file we'll be creating will be saved to

/app/models/PostsModel.php

The completed file should look like this:

```
<?php
	class PostsModel extends AppModel {
	}
?>
```

Naming convention is very important in Simple. By naming our model PostsModel, Simple can automatically infer that this model will be used in the PostsController, and will be tied to a database table called posts.

Note: The Model Object is automatically created and can be accessed in the PostsController by Model::get(). Inside any other controller or model the Model object for Posts can be accessed by Model::get (posts').

All controller methods should have a Model file associated with them, failing which the application may produce errors at runtime.

Creating the Posts Controller

Next, we'll create a controller for our posts. The controller is where all the business logic for post interaction will happen. In a nutshell, it's the place where you play with the models and get post-related work done.

We'll place this new controller in a file called PostsController.php inside the /app/controllers directory. Here's what the basic controller should look like:

```
<?php
	class PostsController extends AppController {
	}
?>
```

Now, lets add an action to our controller. Actions often represent a single function or interface in an application. For example, when users request www.example.com/posts (which is also the same as www.example.com/posts/), they might expect to see a listing of posts. The code for

that action would look something like this:

```
<?php
	class PostsController extends AppController
	{
		public function index()
		{
			$this->set ('posts', Model::get()->fetchAll());
			$this->render();
		}
}
```

Now when www.example.com/posts is accessed, the public method index() of PostsController will automatically be invoked.

Basically, index is a reserved method which can only be invoked by accessing only the class name in the url and not the method name. Eg: if a public function foobar() is created in the PostsController class, it can be accessed by the URL: www.example.com/posts/foobar, whereas index() method is only accessible at www.example.com/posts

All public method of a class are directly accessible from URLs by default (unless over-ridden by the Router).

Although Simple supports highly flexible routing, for now we'll use the default inbuilt routing to avoid any configuration needed.

Creating Post Views

Now that we have our data flowing to our model, and our application logic and flow defined by our controller, let's create a view for the index action we created above.

For most applications the views are HTML mixed with PHP, but they may end up as XML, CSV, or even binary data.

Layouts are presentation code that is wrapped around a view, and can be defined and switched between, but for now, let's just use the default.

Remember in the last section how we assigned the 'posts' variable to the view using the set() method?

That would hand down data to the view that would look something like this:

```
// print_r($posts) output:
Array
(
    [0] => Array
    (
        [id] => 1
        [title] => Title 1
        [body] => Body 1.
        [created] => 1345469798
        [modified] => 0
    )
    [1] => Array
    (
        [id] => 2
        [title] => Title 2
        [body] => Body 2.
        [created] => 1345469798
        [modified] => 0
    )
    [2] => Array
    (
        [id] => 3
        [title] => Title 3
        [body] => Body 3.
        [created] => 1345469798
        [modified] => 0
    )
)
```

View files are stored in /app/views inside a folder named after the controller they correspond to (we'll have to create a folder named 'posts' in this case). To format this post data in a nice table, our view code might look something like this:

```
<!-- File: /app/views/posts/index.php -->
```

```
<h1>Blog posts</h1>
<table>
	<tr>
		<th>Id</th>
		<th>Title</th>
		<th>Created</th>
	</tr>
<!-- Here is where we loop through our $posts array, printing out post info -->
<?php foreach ($posts as $post): ?>
	<tr>
		<td><?php echo $post['id']; ?></td>
		<td> <a href="<?= BASE_URL .  'posts/' . $post['id'] ?>
		"><?= $post['title'] ?></a>
		</td>
		<td><?= date("F j, Y, g:i a", $post['created']); ?></td>
	</tr>
<?php endforeach;  unset($post); ?>
</table>
```

If you happened to have clicked on one of the links we created in this view (that link a post's title to a URL /posts/view/<some_id>), you were probably informed by Simple that the URL does not exist. If you were not so informed, either something has gone wrong, or you actually did define it already, in which case you are very sneaky. Otherwise, we'll create it in the PostsController now:

```
class PostsController extends AppController
{
	public function index()
	{
		$this->set('posts', Model::get()->fetchAll());
		$this->render();
	}
	public function _view($id = null)
	{
		$this->set('post', Model::get()->fetchAll("`id`=$id");
		$this->render();
	}
}
```

Well, to curb this Simple's Non-Linear Routing comes in handy. In the /app/router.php add the following lines of code:

```
Router::get(array ('/posts/(:num)'),
        array(
                'class' => 'posts',
                'func' => '_view',
                'params' => array($_parms['req_args'][1])
        )
);
```

Notice that our view action takes a parameter: the ID of the post we'd like to see. This parameter is handed to the action through the Router. If a user requests /posts/3, then the value '3' is passed as $id to the _view action of PostsController in this way.

Note: The Functions listenning on HTTP Requests of type 'GET' begin with an underscore ('_') in Simple.

Now let's create the view for our new 'view' action and place it in /app/views/posts/view.php.

```
<!-- File: /app/views/posts/view.php -->
<h1><?= $post[0]['title'] ?></h1>
<p><small>Created: <?= date("F j, Y, g:i a", $post[0]['created']) ?></small></p>
<p><?= $post[0]['body'] ?></p>
```

Verify that this is working by trying the links at /posts or manually requesting a post by accessing /posts/1.

Adding Posts

Reading from the database and showing us the posts is a great start, but let's allow for the adding of new posts.

First, start by creating an add() action in the PostsController:

```
<?php
class PostsController extends AppController
{
        public function index()
        {
```

```
                $this->set('posts', Model::get()->fetchAll());
                $this->render();
        }
        public function _view ($id = null)
        {
                $this->set('post', Model::get()->fetchAll("`id`=$id");
                $this->render();
        }
        public function add ()
        {
                $this->data['time'] = time();
                if (Model::get()->add($this->data))
                {
                        $this->Session->setFlash('Your post has been saved.');
                } else {
                        $this->Session->setFlash('Unable to add your post.');
                }
        }
        $this->redirect('posts');
}
```

Since the method name does not begin with an underscore, it is interpreted as a POST HTTP request.

The redirect() method of the Controller, redirects the user to another URL, inside or outside of the domain. The argument supplied to the method is the URL to be redirected to. And works similar to header ('Location: posts');. Since Simple uses non linear custom routing, the URLs for the redirect method have to be manually supplied.

The setFlash method of the Session object is used to set a session message.

Data Validation

To ease the process of Data Input, sanitization and validation. Simple goes an extra step to automatically Sanitize the user input and validate it according to the rules specified in the Model for the Class.

Here's our add view:

```
<!-- File: /app/views/posts/add.php -->
<h1>Add Post</h1>
```

```
<?= isset($error) ? ('<p style="color:red;">' . $error . '</p>') : null ?>
<form method='post' action='<?= BASE_URL ?>posts/add'>
        <label>Title</label><input type='text' name='title'/>
        <label>Content</label><textarea name='body'></textarea>
        <input type='submit' value='Add Post'>
</form>
```

Now let's go back and update our /app/views/posts/index.php view to include a new "Add Post" link. Before the <table>, add the following line:

```
<a href="<?= BASE_URL ?>posts/add">Add Post</a>
```

Now, we have created the view and the controller logic for add method, but we have still not specified the custom validation logic for our method. For this we open our PostsModel.php file in app/views/models/PostsModel.php and add a few lines of code to it:

```
<?php
class PostsModel extends AppModel {
        public $validators = array(
                'title' => array (
                   'notEmpty' => array (
                      'msg' => 'Please supply a value for Post title.',
                      'code' => 'emptyTitle',
                   ),
                ),
                'body' => array (
                   'notEmpty' => array (
                      'msg' => 'Please supply a value for Post Content.',
                      'code' => 'emptyContent',
                   ),
                ),
             );
}
```

The $validators array tells Simple how to validate your data before the controller method is called (Given all the method arguments are set in the HTTP request). Here, we've specified that both the data and title fields must not be empty. Simple's validation engine is strong, with a number of pre-built rules (email addresses, data-length, etc.) and flexibility for adding your own validation rules.

Editing Posts

Post editing: here we go. You're a pro at Simple by now, so you should have picked up a

pattern. Make the action, then the view. Here's what the edit() action of the PostsController would look like:

```
<?php
	public function _edit ($id = null) {
		if (($data = Model::get->fetchAll("posts.id=$id")))
		{
			$this->set('data', $data);
			$this->render();
		} else {
			$this->Session->setFlash('Please supply a valid Post ID.');
			$this->redirect('posts');
		}
	}

	public function edit ()
	{
		$id = $this->data['id'];
		unset($this->data['id']);
		if (Model::get()->edit($this->data, "posts.id=$id"))
			$this->Session->setFlash('Post successfully edited.');
		else
			$this->Session->setFlash('There was an error in editing your post, Please
try again later.');
		$this->redirect('posts');
	}
}
```

Also, add the following to your /app/router.php file:

```
Router::get(array ('/posts/edit/(:num)'),
	array(
		'class' => 'posts',
		'func' => '_edit',
		'params' => array($_parms['req_args'][2])
	)
);
```

The GET request is handeled by the _edit() method and the POST request is handeled by the

edit() method. In the edit() method the pre-built edit() Method of the Model is called and the data array is supplied to it to be updated corresponding to the Post record with the id = $id.

The _edit() method fetches data related to the Post record with the id = $id.

The edit view might look something like this:

```
<!-- File: /app/views/posts/edit.php -->
<h1>Edit Post</h1>
<?php
        if (isset($error))
        {
                print '<p style="color:red;">' . $error . '</p><br><a
href="javascript:window.history.go(-1)">Go Back</a>';
                return;
        }
<?php endif; ?>
<form action="<?= BASE_URL ?>posts/edit">
        <input type="hidden" name="id" value="<?= $this->args->req_args[2] ?>" />
        <label>Title: </label><input type="text" name="title" value="<?= $data[0]['title'] ?
>"/>
        <label>Content: </label><input type="text" name="body" value="<?= $data[0]
['data'] ?>" />
        <input type="submit" value="Edit Post" />
</form>
```

You can now update the file /app/views/posts/index.php to include a link to edit posts:

```
<h1>Blog posts</h1>
<table>
        <tr>
                <th>Id</th>
                <th>Title</th>
                <th>Created</th>
        </tr>
<!-- Here is where we loop through our $posts array, printing out post info -->
<?php foreach ($posts as $post): ?>
        <tr>
                <td><?php echo $post['id']; ?></td>
                <td> <a href="<?= BASE_URL .  'posts/' . $post['id'] ?>
                "><?= $post['title'] ?></a>
                </td>
                <td><a href="<?= BASE_URL ?>posts/edit/<?= $post['id'] ?>">Edit</a></td>
```

```
            <td><?= date("F j, Y, g:i a", $post['created']); ?></td>
        </tr>
<?php
        endforeach;
        unset($post);
?>
</table>
```

Now, lets add a validation rule for 'id' in PostsModel.

```
<?php
class PostsModel extends AppModel {
        public $validators = array(
                'id' => array (
                   'notEmpty' => array (
                      'msg' => 'Please supply a value for Post ID.',
                      'code' => 'emptyId',
                   ),
                ),
```

Deleting Posts

Next, let's make a way for users to delete posts. Start with a delete() action in the PostsController:

```
<?php
        public function delete()
        {
                if (Model::get()->delete("posts.id=" . $this->data['id']))
                        $this->Session->setFlash('The post with id: ' . $this->data['id'] . ' has
been deleted.');
                else
                        $this->Session->setFlash('There was an error in deleting the post.');
                $this->redirect('posts');
        }
```

This logic deletes the post specified by $id, and uses $this->Session->setFlash() to show the

user a confirmation message after redirecting them on to /posts. The function only works on a POST request.

Because we're just executing some logic and redirecting, this action has no view. You might want to update your index view with links that allow users to delete posts, add the following code snippet to the posts HTML table:

```
<td>
	<form method="post" action=<?= BASE_URL ?>posts/delete">
		<input type="hidden" name="id" value="<?= $post['id'] ?>"/>
		<input type="submit" value="Delete Post" />
	</form>
</td>
```

Routes

Still wondering why while accessing / on your application generates a 404 Error Document?

Well, to curb this Simple's Non-Linear Routing comes in handy. In the /app/router.php add the following lines of code:

```
Router::get(array ('/'), array ('class' => 'posts'));
```

Finally your /app/router.php file should look like this:

```
Router::get(array ('/'), array ('class' => 'posts'));
Router::get(array ('/posts', '/posts/view'),
	array(
		'class' => 'posts',
		'func' => 'render',
		'params' => array('404')
	)
);
Router::get(array ('/posts/(:num)'),
	array(
		'class' => 'posts',
		'func' => '_view',
		'params' => array($_parms['req_args'][1])
```

```
        )
);
Router::get(array ('/posts/edit/(:num)'),
        array(
                'class' => 'posts',
                'func' => '_edit',
                'params' => array($_parms['req_args'][2])
        )
);
```

The first tick maps / to index action of Posts controller, so / now displays the content that earlier used to be addressed using the URL /posts.

The second tick generates the pre-built 404 view for the /posts and /posts/view URLs.

The third tick maps /posts/<numerical value> to the path /posts/view/?id=<numerical value>. i.e the _view method of PostsController Class. $_parms['req_args'] is an array which contains all the arguments of the Requested URL as seperate array keys.

For advanced Routing principles please head to the Routes section.

Installation

Simple is fast and easy to install. The minimum requirements are a webserver and a copy of Simple, that's it! While this manual focuses primarily on setting up with Apache (because it's the most common), you can configure Simple to run on a variety of web servers such as LightHTTPD or Microsoft IIS.

Requirements

- HTTP Server. For example: Apache. mod_rewrite is preferred, but by no means required.
- PHP 5.3.0 or greater.

Technically a database engine isn't required, but we imagine that most applications will utilize one.

Simple supports only MySQL (4 or greater) currently.

License

Simple is licensed under the MIT license. This means that you are free to modify, distribute and republish the source code on the condition that the copyright notices are left intact. You are also free to incorporate Simple into any Commercial or closed source application.

Downloading Simple

There are two main ways to get a fresh copy of Simple. You can either download an archive copy (zip/tar.gz/tar.bz2) from the main website, or check out the code from the git repository.

To download the latest major release of Simple. Visit the main website http://www.simplephp.org and follow the "Download Now" link.

All current releases of Simple are hosted on Github (http://github.com/deepanshumehndiratta/simplephp).

Alternatively you can get fresh off the press code, with all the bug-fixes and up to the minute enhancements.

These can be accessed from github by cloning the Github (http://github.com/deepanshumehndiratta/simplephp) repository:

git clone git://github.com/deepanshumehndiratta/simplephp .git

Permissions

Simple uses the app/tmp directory for a number of different operations. Cached views, and session information are just a few examples. As such, make sure the directory app/tmp and all its subdirectories in your Simple installation are writable by the web server user.

Setup

Setting up Simple can be as simple as slapping it in your web server's document root, or as complex and flexible as you wish. This section will cover the three main installation types for Simple: development, production, and advanced.

• Development: easy to get going, clean URLs, Database and Cache Profiling, reports all errors.

• Testing: Requires the ability to configure the web server's document root, clean URLs, se- cure, ignores warnings .

• Live: No error reporting, no Cache or Database Profiler.

Development

A development installation is the default method to setup Simple. This example will help you install a Simple application and make it available at http://www.example.com/simple/. We assume for the pur- poses of this example that your document root is set to /var/www/html.

Unpack the contents of the Simple archive into /var/www/html. You now have a folder in your document root named after the release you've downloaded (e.g. simple_0_1). Rename this folder to simple.

Your development setup will look like this on the file system:

```
/var/www/html/
        simple/
                app/
                core/
                plugins/
                .htaccess
                index.php
                README .md
```

If your web server is configured correctly, you should now find your Simple application accessible at http://www.example.com/simple/.

Using one Simple checkout for multiple applications

If you are developing a number of applications, it often makes sense to have them share the same Simple core checkout.. To start off, clone Simple into a directory. For this example, we'll use ~/projects:

```
git clone git://github.com/deepanshumehndiratta/simplephp.git ~/projects/simple
```

This will clone Simple into your ~/projects directory. If you don't want to use git, you can download a zipball and the remaining steps will be the same.

To configure your Simple installation, you'll need to make some changes to the following files.

• /app/webroot/index.php

There are three constants that you'll need to edit: CORE_PATH, BASE_PATH, and APP_FOLDER.

• CORE_PATH should be set to the path of the directory that contains your core libraries folder.

• BASE_PATH should be set to the (base)path of your html folder or sub-folder.

• APP_FOLDER should be set to the name of your Simple app folder or relative path of app folder from html directory(if it is not inside the BASE_PATH folder).

Let's run through an example so you can see what an advanced installation might look like in practice.

Imagine that I wanted to set up Simple to work as follows:

• The Simple core libraries will be placed in /usr/lib/simple.

• My application's webroot directory will be /var/www/mysite/.

• My application's app directory will be /home/me/myapp.

Given this type of setup, I would need to edit my webroot/index.php file (which will end up at /var/www/mysite/index.php, in this example) to look like the following:

```
<?php
if (!defined('BASE_PATH')) {
        define('BASE_PATH', DS . 'home' . DS . 'me');
}
if (!defined('APP_FOLDER')) {
        define ('APP_FOLDER', 'myapp');
}
if (!defined('CORE_PATH')) {
        define('CORE_PATH', DS . 'usr' . DS . 'lib');
```

```
}
```

It is recommended to use the DS constant rather than slashes to delimit file paths. This prevents any missing file errors you might get as a result of using the wrong delimiter, and it makes your code more portable.

Apache and mod_rewrite (and .htaccess)

While Simple is built to work with mod_rewrite out of the box–and usually does–we've noticed that a few users struggle with getting everything to play nicely on their systems.

Here are a few things you might try to get it running correctly. First look at your httpd.conf (Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.conf).

1. Make sure that an .htaccess override is allowed and that AllowOverride is set to All for the correct DocumentRoot. You should see something similar to:

```
# Each directory to which Apache has access can be configured with respect
# to which services and features are allowed and/or disabled in that
# directory (and its subdirectories).
#
# First, we configure the "default" to be a very restrictive set of
# features.
#
<Directory />
        Options FollowSymLinks
        AllowOverride All
#       Order deny,allow
#       Deny from all
</Directory>
```

2. Make sure you are loading up mod_rewrite correctly. You should see something like: LoadModule rewrite_module libexec/apache2/mod_rewrite.so

In many systems these will be commented out (by being prepended with a #) by default, so you may just need to remove those leading # symbols.

After you make changes, restart Apache to make sure the settings are active. Verify that you your .htaccess files are actually in the right directories. This can happen during copying because some operating systems treat files that start with '.' as hidden and therefore won't see them to copy.

3. Make sure your copy of Simple is from the downloads section of the site or our GIT repository, and has been unpacked correctly by checking for .htaccess files.

Simple root directory (needs to be copied to your document, this redirects everything to your Simple app):

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

Simple app directory (will be copied to the top directory of your application by bake):

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

Simple webroot directory:

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

If your Simple site still has problems with mod_rewrite you might want to try and modify settings for virtualhosts. If on ubuntu, edit the file /etc/apache2/sites-available/default (location is distribu-tion dependent). In this file, ensure that AllowOverride None is changed to AllowOverride

All, so you have:

```
<Directory />
        Options FollowSymLinks
        AllowOverride All
</Directory>
<Directory /var/www>
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Order Allow,Deny
        Allow from all
</Directory>
```

If on Mac OSX, another solution is to use the tool virtualhostx to make a virtual host to point to your folder.

For many hosting services (GoDaddy, 1and1), your web server is actually being served from a user directory that already uses mod_rewrite. If you are installing Simple into a user directory (http://example.com/~username/simple/), or any other URL structure that already utilizes mod_rewrite, you'll need to add RewriteBase statements to the .htaccess files Simple uses (/.htac-cess, /app/.htaccess, /app/webroot/.htaccess).

This can be added to the same section with the RewriteEngine directive, so for example your webroot .htaccess file would look like:

```
<IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteBase /path/to/simple/app
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

The details of those changes will depend on your setup, and can include additional things that are not Simple related. Please refer to Apache's online documentation for more information.

Pretty URLs on nginx

nginx is a popular server that uses less system resources than Apache. Its drawback is that it does not make use of .htaccess files like Apache, so it is necessary to create those rewritten URLs in the site-available configuration. Depending upon your setup, you will have to modify this, but at the very least, you will need PHP running as a FastCGI instance.

```
server {
        listen 80;
        server_name www.example.com;
        rewrite ^(.*) http://example.com$1 permanent;
}
server {
        listen 80;
        server_name example.com;
        # root directive should be global
        root
        /var/www/example.com/public/app/webroot/;
        access_log /var/www/example.com/log/access.log;
        error_log /var/www/example.com/log/error.log;
        location / {
                index index.php index.html index.htm;
```

```
                try_files $uri $uri/ /index.php?$uri&$args;
        }
        location ~ \.php$ {
                include /etc/nginx/fcgi.conf;
                fastcgi_pass 127.0.0.1:10005;
                fastcgi_index index.php;
                fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        }
}
```

URL Rewrites on IIS7 (Windows hosts)

IIS7 does not natively support .htaccess files. While there are add-ons that can add this support, you can also import htaccess rules into IIS to use Simple's native rewrites. To do this, follow these steps:

1. Use Microsoft’s Web Platform Installer to install the URL Rewrite Module 2.0.

2. Create a new file in your Simple folder, called web.config.

3. Using Notepad or another XML-safe editor, copy the following code into your new web.config file...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <system.webServer>
                <rewrite>
                        <rules>
                                <rule name="Imported Rule 1" stopProcessing="true">
                                        <match url="^(.*)$" ignoreCase="false" />
                                        <conditions logicalGrouping="MatchAll">
                                                <add input="{REQUEST_FILENAME}"
matchType="IsDirectory" negate="tru
                                                <add input="{REQUEST_FILENAME}"
matchType="IsFile" negate="true" />
                                        </conditions>
                                        <action type="Rewrite" url="index.php?url={R:1}"
appendQueryString="true" />
                                </rule>
                                <rule name="Imported Rule 2" stopProcessing="true">
                                        <match url="^$" ignoreCase="false" />
                                        <action type="Rewrite" url="/" />
```

```
                        </rule>
                        <rule name="Imported Rule 3" stopProcessing="true">
                                <match url="(.*)" ignoreCase="false" />
                                <action type="Rewrite" url="/{R:1}" />
                        </rule>
                        <rule name="Imported Rule 4" stopProcessing="true">
                                <match url="^(.*)$" ignoreCase="false" />
                                <conditions logicalGrouping="MatchAll">
                                        <add input="{REQUEST_FILENAME}"
matchType="IsDirectory" negate="true" />
                                        <add input="{REQUEST_FILENAME}"
matchType="IsFile" negate="true" />
                                </conditions>
                                <action type="Rewrite" url="index.php/{R:1}"
appendQueryString="true" />
                        </rule>
                </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

It is also possible to use the Import functionality in IIS's URL Rewrite module to import rules directly from Simple's .htaccess files in root, /app/, and /app/webroot/ - although some editing within IIS may be necessary to get these to work. When Importing the rules this way, IIS will automatically create your web.config file for you.

Once the web.config file is created with the correct IIS-friendly rewrite rules, Simple's links, css, js, and rerouting should work correctly.

Fire It Up

Alright, let's see Simple in action. Depending on which setup you used, you should point your browser to http://example.com/ or http://example.com/simple/. At this point, you'll be presented with Simple's default home, and a message that tells you the status of your current database connection.

Congratulations! You are ready to create your first Simple application.

What is Simple? Why Use it?

Simple (http://www.simplephp.org/) is a free (http://en.wikipedia.org/wiki/MIT_License), open-source (http://en.wikipedia.org/wiki/Open_source), rapid development (http://en.wikipedia.org/wiki/Rapid_application_development) framework (http://en.wikipedia.org/wiki/Application_framework) for PHP (http://www.php.net/). It's a foundational structure for programmers to create web applications. Our primary goal is to enable you to work in a structured and rapid manner–without loss of flexibility. Simple takes the monotony out of web development. We provide you with all the tools you need to get started coding what you really need to get done: the logic specific to your application. Instead of reinventing the wheel every time you sit down to a new project, check out a copy of Simple and get started with the real guts of your application.

Here's a quick list of features you'll enjoy when using Simple:

- Flexible licensing (http://en.wikipedia.org/wiki/MIT_License)
- Compatible with versions PHP 5.3.0 and greater.
- Integrated CRUD (http://en.wikipedia.org/wiki/Create,_read,_update_and_delete) for database interaction.
- Application scaffolding (http://en.wikipedia.org/wiki/Scaffold_(programming)).
- MVC (http://en.wikipedia.org/wiki/Model-view-controller) architecture.
- Request dispatcher with clean, custom URLs and routes.
- Built-in validation (http://en.wikipedia.org/wiki/Data_validation).
- Email, Security, Session, and Request Handling Components.
- Data Sanitization.
- Flexible Caching (http://en.wikipedia.org/wiki/Web_cache).
- Localization.

- Works from any web site directory, with little to no Apache (http://httpd.apache.org/) configuration involved.

Understanding Model-View-Controller

Simple follows the MVC (http://en.wikipedia.org/wiki/Model-view-controller) software design pattern. Programming using MVC separates your application into three main parts:

The Model layer

The Model layer represents the part of your application that implements the business logic. this means that it is responsible for retrieving data, converting it into meaningful concepts to your application, as well as processing, validating, associating and any other task relative to handling this data.

At a first glance, Model objects can be looked at as the first layer of interaction with any database you might be using for your application. But in general they stand for the major concepts around which you implement your application.

In the case of a social network, the Model layer would take care of tasks such as Saving the user data, saving friends associations, storing and retrieving user photos, finding new friends for suggestions, etc. While the model objects can be thought as “Friend”, “User”, “Comment”, “Photo”.

The View layer

The View renders a presentation of modeled data. Being separated from the Model objects, it is responsible for using the information it has available to produce any presentational interface your application might need.

For example, as the Model layer returns a set of data, the view would use it to render a HTML page containing it. Or a XML formatted result for others to consume. The View layer is not only limited to HTML or text representation of the data, it can be used to deliver a wide variety of formats depending on your needs, such as videos, music, documents and any other format you can think of.

The Controller layer

The Controller layer handles requests from users. It’s responsible for rendering back a response with the aid of both the Model and the View Layer. Controllers can be seen as managers taking care that all needed resources for completing a task are delegated to the correct workers. It waits for petitions from clients, checks their validity according to authentication or authorization rules, delegates data fetching or processing to the model, and selects the correct type of presentational data that the client is accepting, to finally delegate this rendering process to the View layer.

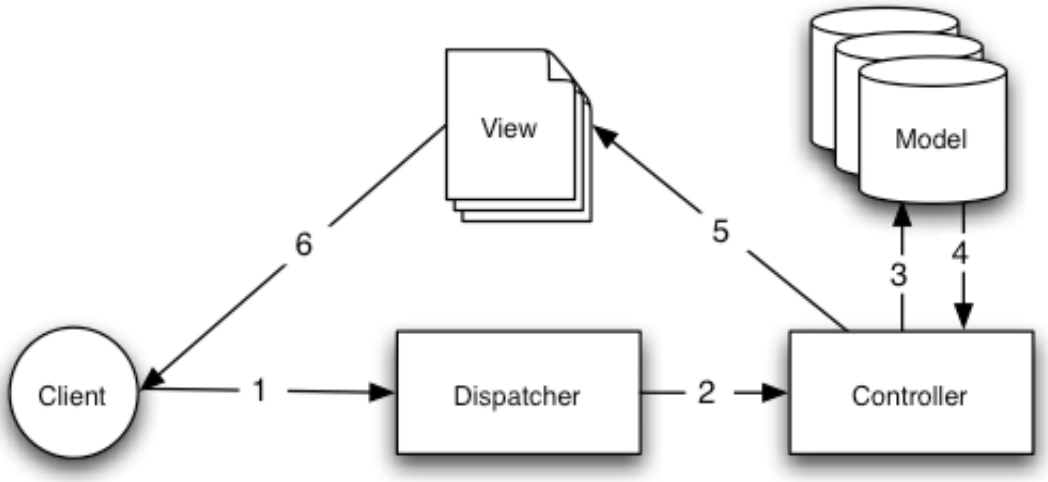

Figure: 1: A Basic MVC Request

The typical Simple request cycle starts with a user requesting a page or resource in your application. This request is first processed by a dispatcher which will select the correct controller object to handle it. Once the request arrives at the controller, it will communicate with the Model layer to process any data fetching or saving operation that might be needed. After this communication is over, the controller will proceed at delegating to the correct view object the task of generating an output resulting from the data provided by the model. Finally, when this output is generated, it is immediately rendered to the user Almost every request to your application will follow this basic pattern. We'll add some details later on which are specific to Simple, so keep this in mind as we proceed.

Benefits

Why use MVC? Because it is a tried and true software design pattern that turns an application into a maintainable, modular, rapidly developed package. Crafting application tasks into separate models, views, and controllers makes your application very light on its feet. New features are easily added, and new faces on old features are a snap. The modular and separate design also allows developers and designers to work simultaneously, including the ability to rapidly prototype (http://en.wikipedia.org/wiki/Software_prototyping). Separation also allows developers to make changes in one part of the application without affecting the others.

If you've never built an application this way, it takes some time getting used to, but we're confident that once you've built your first application using Simple, you won't want to do it any other way.

To get started on your first Simple application, try the blog tutorial now.

Controllers

Controllers are the ‘C’ in MVC. After routing has been applied and the correct controller has been found, your controller’s action is called. Your controller should handle interpreting the request data, making sure the correct models are called, and the right response or view is rendered. Controllers can be thought of as middle man between the Model and View. You want to keep your controllers thin, and your models fat. This will help you more easily reuse your code and makes your code easier to test.

Commonly, controllers are used to manage the logic around a single model. For example, if you were building a site for an on-line blog, you might have a PostsController and an UsersController managing your posts and their authors. In Simple, controllers are named after the primary model they handle. It’s totally possible to have controllers work with more than one model as well.

Your application’s controllers extend AppController class, which in turn extends the core Controller class and which again inherits the Simple Framework base class. The AppController class is defined in /app/controllers/AppController.php and it should contain methods that are shared between all of your application’s controllers.

Controllers provide a number of methods which are called actions. Actions are methods on a controller that handle requests. By default all public methods on a controller are an action, and accessible from a url.

Actions are responsible for interpreting the request and creating the response. Usually responses are in the form of a rendered view, but there are other ways to create responses as well.

The App Controller

As stated in the introduction, the AppController class is the parent class to all of your application’s controllers. AppController itself extends the Controller class included in the Simple core library. As such, AppController is defined in /app/controllers/AppController.php like so:

```
<?php

class AppController extends Controller

{ }
```

Controller attributes and methods created in your AppController will be available to all of your application’s controllers. It is the ideal place to create code that is common to all of your controllers.

Request parameters

When a request is made to a Simple application, Simple's Router and Scribe classes use Routes Configuration to find and create the correct controller. The request data is encapsulated into a request object. Simple puts all of the important request information into

the $this->request, $this->data, $this->args and $this->ajax properties.

Controller actions

Controller actions are responsible for converting the request parameters into a response for the browser/user making the request. Simple uses conventions to automate this process and remove some boiler-plate code you would otherwise need to write.

By convention Simple renders a view with an inflected version of the action name. Returning to our online blog example, our PostsController might contain the view(), share(), and search() actions.

The controller would be found in /app/controllers/PostsController.php and contain:

```
<?php
# /app/controllers/PostsController.php
class PostsController extends AppController
{
        // Listens on a GET type HTTP Request with “id” passed to the Method by Router
        public function _view ($id)
        {
                //action logic goes here..
        }
        // Listens on a POST type HTTP Request
        public function share ()
        {
                //action logic goes here..
        }
        // Listens on a GET type HTTP Request with “query”  passed to the Method by Router
        public function _search ($query)
        {
                //action logic goes here..
        }
}
```

The view files for these actions would be app/views/posts/view.php, app/views/posts/share.php, and app/views/posts/search.php. The conventional view file name is same the action name. Controller actions generally use set() to create a context that View uses to render the view. Because of the conventions that Simple uses, you don’t need to create and render the view manually. Instead once a controller action has completed, Simple will handle rendering and delivering the View. If for some reason you’d like to skip the default behavior. Both of the following techniques will by-pass the default view rendering behavior.

• Exiting your application inside the method.

• Returning true from inside a method.

• Using $this->returnAjaxJson() inside the Controller method to return a JSON output.

Check the request type before returning:

```
<?php
class PostsController extends AppController
{
	public function _popular()
	{
		$popular = Model::get()->popular();
		if ($this->ajax) {
			$this->returnAjaxJson($popular);
		}
		$this->set('popular', $popular);
	}
}
```

The above controller action is an example of how a method can be used with Ajax

and normal requests. Returning a HTML view to a JSON Ajax request will cause errors and should be avoided.

In order for you to use a controller effectively in your own application, we'll cover some of the core attributes and methods provided by Simple's controllers.

Request Life-cycle callbacks

Simple controllers come fitted with callbacks you can use to insert logic around the request life-cycle:

```
Controller::__pre()
```

This function is executed before execution of the requested action in the controller after applying the routing. It's a handy place to check for an active session or inspect user permissions.

Note: The __pre() method will not be called for missing actions, and scaffolded actions.

```
Controller::__beforeRender()
```

Called after controller action logic, but before the view is rendered (This method is not called if Scribe is used to handle the request). This callback is not used often, but may be needed if you are calling render() manually before the end of a given action.

```
Controller::__post()
```

Called after every controller action, and after rendering is complete. This is the last controller method to run. And is run during the shutdown cycle of the Request.

Controller Methods

Interacting with Views

Controllers interact with the view in a number of ways. First they are able to pass data to the views, using set(). You can also decide which view class to use, and which view file should be rendered from the controller.

Controller::set(string $var, mixed $value)

The set() method is the main way to send data from your controller to your view. Once you've used set(), the variable can be accessed in your view:

```
<?php
// First you pass data from the controller:
$this->set('color', 'pink');
// Then, in the view, you can utilize the data:
?>
You have selected <?php echo $color; ?> icing for the cake.
```

The set() method also takes an associative array as its first parameter. This can often be a quick way to assign a set of information to the view.

```
<?php
$data = array(
        'color' => 'pink',
        'type' => 'sugar',
        'base_price' => 23.95
);
// make $color, $type, and $base_price
// available to the view:
$this->set($data);
```

Controller::render(string $class)

The render() method is automatically called at the end of each requested controller action. This method performs all the view logic (using the data you've given in using the set() method), places the view inside its layout and serves it back to the end user.

The default view file used by render is determined by convention. If the search() action of the PostsController is requested, the view file in /app/views/posts/search.php will be rendered:

```
<?php
```

```
class PostsController extends AppController
{
// ...
public function search()
{
        // Render the view in /app/views/recipes/search.php
        $this->render();
}
// ...
}
```

Although Simple will automatically call it after every action's logic, you can use it to specify an alternate view file by specifying the custom view filename in $class.

The view file is called relative to the path /app/views/ directory.

```
<?php
// Render the element in /app/views/_elements/ajaxreturn.php
$this->render('/_elements/ajaxreturn');
```

Rendering a specific Layout

You can also specify an alternate layout for a class by setting the $this->layout parameter. The $layout parameter allows you to specify the layout the view is rendered in.

The $layout should be relative to the /app/views/_layouts/ directory.

The default rendered layout for a class should located in /app/views/_layouts/<class_name>/main.php if this is not present, the default Application view at /app/views/_layouts/main.php is used.

Rendering a specific view

In your controller you may want to render a different view than what would conventionally be done. You can do this by calling render() directly. Once you have called render() Simple will not try to re-render the view:

```
<?php
class PostsController extends AppController
{
        public function my_action() {
                $this->render('custom_file');
        }
}
```

This would render app/views/custom_file.php instead of app/views/posts/my_action.php

Flow Control

Controller::redirect(mixed $target, boolean $type)

The flow control method you'll use most often is redirect(). This method takes its first parameter in the form of a Complete URL or A Path (relative or absolute) to redirect them to a receipt screen.:

```
<?php
public function place_order()
{
	// Logic for finalizing order goes here
	if ($success) {
		$this->redirect("http://www.example.com");
	} else {
		$this->redirect("http://simplephp.org");
	}
}
```

You can also use a relative or absolute Path as the $target argument:

```
<?php
$this->redirect('/orders/thanks'));
$this->redirect('thanks');
```

The second parameter of redirect() allows you to define the type of HTTP status code to accompany the redirect. You may want to use 0 for a 302 (moved temporarily) or 1 for (moved permanently), depending on the nature of the redirect.

The method will issue an exit() after the redirect.

If you need to redirect to the referer page you can use:

```
<?php
$this->redirect($this->args->ref);
```

Controller::flash(string $target, mixed $msg, integer $time)

Like redirect(), the flash() method is used to direct a user to a new page after an operation.

The flash() method is different in that it shows a message before passing the user on to another URL.

The first parameter should hold the Target location (same as the redirect action), the second parameter should hold the message to be displayed, and the third parameter is time to pause between displaying the message and redirecting the user to the target URL.

For in-page flash messages, be sure to check out SessionComponent's setFlash() method.

Sending Mails

Controller::sendMail ($recievers = array(), $subject = null, $variables = array(), $details = array(), $template = 'default')

The **$recievers** array can have 3 keys:

“to”, “cc” and “bcc”

The “to” key is required, the “cc” and “bcc” keys are optional.

All the three keys if set should be arrays containing details of each recipient as a sub array with the keys “email” and “name” set.

The **$subject** variable contains the subject of the mail and is set to null by default.

The **$variables** is an associative array whose keys are set as variables with there values in the array inside the E-mail template.

The **$details** array contains 4 sub-keys: “from”, “sender”, “reply”, “rName”.

“**from**” contains the E-mail address of the sender.

“**sender**” contains the name of the sender.

“**reply**” contains the Reply-to E-mail address and

“**rName**” contains the Name for the Reply-to.

The **$details** array if left empty (only initialized) when passed during function call, the values for the from, sender, etc are taken from the **$config** global configuration array.

The **$template** contains the name of the template to be used, example if $template is set to “my_mail”, the templates at:

/app/views/_layouts/_email/html/my_mail.php and
/app/views/_layouts/_email/text/my_mail.php will be used to render and send the email.

Loading an external file

Controller::requires($files)

This method is used to load external files (Non-Framework) during request execution. The **$files** variable can be both an array with file paths as well as a single file path. In both case the file-path should be relative to the Base Application Path.

Views

Views are the V in MVC. Views are responsible for generating the specific output required for the request. Often this is in the form of HTML, XML, or JSON, but streaming files and creating PDF's that users can download are also responsibilities of the View Layer.

View Templates

The view layer of Simple is how you speak to your users. Most of the time your views will be showing (X)HTML documents to browsers, but you might also need to serve AMF data to a Flash object, reply to a remote application via SOAP, or output a CSV file for a user.

By default Simple view files are written in plain PHP/HTML and have a default extension of .php. These files contain all the presentational logic needed to get the data it received from the controller in a format that is ready for the audience you're serving to. Simple View files are stored in /app/views/, in a folder named after the controller that uses the files, and named after the action it corresponds to. For example, the view file for the Products controller's "view()" action, would normally be found in /app/views/products/view.php.

The view layer in SImple can be made up of a number of different parts. Each part has different uses, and will be covered in this chapter:

• views: Views are the part of the page that is unique to the action being run. They form the meat of your application's response.

• re-usable elements: smaller, reusable bits of view code. Elements are usually rendered inside of views.

• layouts: view files that contain presentational code that is found wrapping many interfaces in your application. Most views are rendered inside of a layout.

Layouts

A layout contains presentation code that wraps around a view. Anything you want to see in all of your views should be placed in a layout.

Layout files should be placed in /app/views/_layouts/<class short name>/. Simple's default layout can be overridden by creating a new default layout at /app/views/_layouts/main.php. Once a new default layout has been created, controller-rendered view code is placed inside of the default layout when the page is rendered.

When you create a layout, you need to tell Simple where to place the code for your views. To do so, make sure your layout includes a place for **$content.**

Here's an example of what a default layout might look like:

```
<!DOCTYPE html>
<html lang="en">
        <head>
                <title><?php echo $title_for_layout?></title>
```

```
		<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
	</head>
	<body>
		<!-- If you'd like some sort of menu to
		show up on all of your views, include it here -->
		<div id="header">
			<div id="menu">...</div>
		</div>
		<!-- Here's where I want my views to be displayed -->
		<?php echo $content; ?>
		<!-- Add a footer to each displayed page -->
		<div id="footer">...</div>
	</body>
</html>
```

The content block contains the contents of the rendered view.

$title_for_layout here contains the page title. This variable is **not** generated automatically and is user set using the Controller::set() method:

```
<?php
class UsersController extends AppController {
	public function view_active()
	{
		$this->set('title_for_layout', 'View Active Users');
	}
}
```

You can also set the title_for_layout variable from inside the view file:

```
<?php
$this->set('title_for_layout', $titleContent);
```

You can create as many layouts as you wish: just place them in the app/views/layouts directory, and switch between them inside of your controller actions using the controller or view's $layout property:

```
<?php
// from a controller
public function admin_view()
{
	// The Layout at /app/views/_layouts/admin/main.php will be rendered
	$this->layout = 'admin';
```

```
}
        // The Layout at /app/views/_layouts/loggedin/main.php will be rendered
        $this->layout = 'loggedin';
```

For example, if a section of my site included a smaller ad banner space, I might create a new layout with the smaller advertising space and specify it as the layout for all controllers' actions using something like:

```
<?php
class UsersController extends AppController {
        public function view_active()
        {
                $this->set('title_for_layout', 'View Active Users');
                $this->layout = 'default_small_ad';
        }

        public function view_image()
        {
                $this->layout = 'image';
                //output user image
        }
}
```

Rendering View without Layout

Simple allows rendering views without a Layout wrapper around them. To do so, simply use the following code:

```
<?php
$this->layout = "";
```

This can be particularly helpful in cases where the output is an Ajax Response or a file.

Partial Rendering

Many applications have small blocks of presentation code that need to be repeated from page to page, sometimes in different places in the layout. Simple can help you repeat parts of your website that need to be reused. These reusable parts are called Partially Renderable Elements. Ads, help boxes, navigational controls, extra menus, login forms, and callouts are often implemented in Simple as elements. An element is basically a miniview that can be included in other views, in layouts, and even within other elements. Elements can be used

to make a view more readable, placing the rendering of repeating elements in its own file. They can also help you re-use content fragments in your application. Elements should be placed in the /app/views/_elements/ folder. They are output using the renderPartial method of the view:

```
<?php echo $this->renderPartial('/app/views/_elements/helpbox'); ?>
```

Note: If file-path begins with / it is mapped relative to BASE_PATH, else it is mapped relative to the current directory.

Passing Variables into an Element

You can pass data to an element through the renderPartial's second argument:

```
<?php
echo $this->renderPartial('/app/views/_elements/helpbox', array(
        "helptext" => "Oh, this text is very helpful."
));
```

Inside the element file, all the passed variables are available as members of the parameter array (in the same way that Controller::set() in the controller works with view files). In the above example, the /app/views/Elements/helpbox.ctp file can use the $helptext variable:

```
<?php
// inside /app/views/_elements/helpbox.php
echo $helptext; //outputs "Oh, this text is very helpful."
```

Default Controller Methods for interacting with Views

Controller::set(string $var, mixed $value)

The set() method is the main way to send data from your controller to your view. Once you've used set(), the variable can be accessed in your view:

```
<?php
// First you pass data from the controller:
$this->set('color', 'pink');
// Then, in the view, you can utilize the data:
?>
You have selected <?php echo $color; ?> icing for the cake.
```

The set() method also takes an associative array as its first parameter. This can often be a quick way to assign a set of information to the view.

```
<?php
```

```
$data = array(
        'color' => 'pink',
        'type' => 'sugar',
        'base_price' => 23.95
);
// make $color, $type, and $base_price
// available to the view:
$this->set($data);
```

Controller::is_set(string $var)

Check if a variable is set for view.

Controller::vget(string $var)

Get the value of a variable set for view.

Controller::render(string $class)

The render() method is automatically called at the end of each requested controller action. This method performs all the view logic (using the data you've given in using the set() method), places the view inside its layout and serves it back to the end user.

The default view file used by render is determined by convention. If the search() action of the PostsController is requested, the view file in /app/views/posts/search.php will be rendered:

```
<?php
class PostsController extends AppController
{
        // ...
        public function search()
        {
                // Render the view in /app/views/recipes/search.php
                $this->render();
        }
        // ...
}
```

Although Simple will automatically call it after every action's logic, you can use it to specify an alternate view file by specifying the custom view filename in $class.

The view file is called relative to the path /app/views/ directory.

```
<?php
```

```
// Render the element in /app/views/_elements/ajaxreturn.php
$this->render('/_elements/ajaxreturn');
```

Rendering a specific Layout

You can also specify an alternate layout for a class by setting the $this->layout parameter. The $layout parameter allows you to specify the layout the view is rendered in.

The $layout should be relative to the /app/views/_layouts/ directory.

The default rendered layout for a class should located in /app/views/_layouts/<class_name>/main.php if this is not present, the default Application view at /app/views/_layouts/main.php is used.

Rendering a specific view

In your controller you may want to render a different view than what would conventionally be done. You can do this by calling render() directly. Once you have called render() Simple will not try to re-render the view:

```
<?php
class PostsController extends AppController
{
	public function my_action() {
		$this->render('custom_file');
	}
}
```

This would render app/views/custom_file.php instead of app/views/posts/my_action.php

Models

Models are the classes that sit as the business layer in your application. This means that they should be responsible for managing almost everything that happens regarding your data, its validity, interactions and evolution of the information workflow in your domain of work.

Usually model classes represent data and are used in Simple applications for data access, more specifically they represent a database table.

This section will explain what features of the model can be automated, how to override those features, and what methods and properties a model can have. It'll explain the different ways to associate your data. It'll describe how to find, save, and delete data. Finally, it'll look at Datasources.

Basics Methods for Fetching Data:

Model::get ($table = null)

Get Model object for a Class. The parameter $table is optional.

Model::fetchAll ($sql = null, $args = array(), $table = null)

Fetch All fields records for a given Model.

$sql variable is optional and should contain conditional MySQL part. Eg: $sql = "posts.id=1", etc.

$args array can have 3 keys, "order", "limit", "group"

Order<String>: MySQL order syntax, eg: "posts.id", orders he fetched dataset by posts.id field.

Limit<Array/Integer>: If an array, eg: array(5, 10), fetches all records between limits 5 and 10. If an integer, eg: 10, fetches all records between 0 and 10.

Group<String>: The fields name(s) to group the dataset by, eg: "posts.category"

$table variable is used to select data from a custom table, other than the default for the current Model.

Returns the Dataset as an array.

Model::fetchCount ($sql = null, $args = array(), $table = null)

This function fetches the count of the dataset according to the given SQL parameters. All parameters are same as Model::fetchAll

Model::fetchMax ($args = array(), $sql = null, $table = null)

This function selects Maximum values for all columns specified in the $args variable. Rest all parameters are same as Model::fetchAll

$args['columns'] <array/string>: Names of all columns whose maximum has to be found.

Returns the Dataset as an array.

Model::fetchMin ($args = array(), $sql = null, $table = null)

This function selects Minimum values for all columns specified in the $args variable. All parameters are same as Model::fetchMax

Returns the Dataset as an array.

Model::fetchAvg ($args = array(), $sql = null, $table = null)

This function selects Average values for all columns specified in the $args variable. All parameters are same as Model::fetchMax

Returns the Dataset as an array.

Model::fetchByQuery ($sql)

Fetch the output of a SQL query as an array.

Model::fetchKeys ($keys = array(), $sql = null, $args = array(), $table = null)

Fetch only specific columns from records for a given Model.

$keys<array/string>: Names of the columns to be fetched.

Rest all parameters are same as Model::fetchAll()
Returns the Dataset as an array.

Model::add ($data = array(), $table = null)

Add a Row to the Table.
$data is an associative array with key-value pairs of the columns and data.

$table is an optional field to add data to a custom table other than that used by the current Model.
Returns True/False

Model::id ()

Returns the ID of the Auto-Increment field from the Last Add operation for the current Model.

Model::edit ($data = array(), $sql = null, $table = null)

Edit a Database record.

Model::delete ($sql = null, $table = null)

Delete a databse record.

Model::fetchByJoin ($args = array())

The most complicated of the lot. Used to fetch values from different Tables by Equi-Join.
$args ahs the following keys:
tables, keys, limit, joins, sql, order

eg:

```
$data = Model::fetchByJoin(
        array(
                'tables' => array('users', 'posts'),
                'keys' => array(
                        'users' => array(
                                'id', 'name', 'email'
                        ),
                        'posts' => array(
                                'id', 'user_id', 'title', 'body'
                        )
                ),
                'joins' => array(
                        array(
                                array(
                                        'table' => 'users',
                                        'key' => 'id'
                                ),
                                array(
                                        'table' => 'posts',
                                        'key' => 'user_id'
                                )
                        )
                ),
                'sql' => null,
                'order' => 'posts.id',
                'group' => 'users.id'
        )
)

// View the Format of the output
print_r($data);
```

Note:
Executing methods like $model_object->rt_obj()->fetchAll() will return the Dataset as ORM.

Data Validations:

Validate Data before ADD and EDIT actions.

Format:

```
<?php
class PostsModel extends AppModel {
        public $validators = array(
                'title' => array (
                   'notEmpty' => array (
                      'msg' => 'Please supply a value for Post title.',
                      'code' => 'emptyTitle',
                   ),
                ),
                'body' => array (
                   'notEmpty' => array (
                      'msg' => 'Please supply a value for Post Content.',
                      'code' => 'emptyContent',
                   ),
                ),
             );
}
```

This will validate 'title' and 'body' fields passed to the Model during ADD and EDIT operatons to be Not Empty.

If they are empty, the appropriate view for the Controller Method called will be rendered with the Error Messages.

In-built Valiators:

'**notEmpty**' – Check if a field is empty or not.

'**checkLength**' – eg:

```
        'title' => array (
                'checkLength' => array (
                   'msg' => 'Post title should not be more than 20 characters.',
                   'code' => 'invalidLength',
                    'params' => array(20)
```

```
                    ),
                ),
```

'**checkPassword**' – Check if a field is 8-20 characters and contains [a-zA-z] and [0-9] or a special character.

'**checkEmail**' – Check if a field contains valid E-mail address.

You can write your own validation function, eg:

```
 $validators = array(
        'title' => array (
                    'myValidationFunction' => array (
                        'msg' => 'Custom Test Failed.',
                        'code' => 'Failed Test',
                         'params' => array('some random value1', 'value2')
                    ),
                ),
        );



public function  myValidationFunction ($data, $value1, $value2) {
        // Perform some Logic here and Return True or False
        return false;
}
```