CODEIGNITER – PART ONE

INTRODUCTION

CodeIgniter is a MVC framework built using PHP. In this lab we'll look at how to get up and running with CodeIgniter on the University's *homepages.shu.ac.uk* server. The lab assumes you are happy with Object Oriented programming in PHP.

INSTALLATION

Download the latest version 3.x of CodeIgniter from:

https://codeigniter.com/

Upzip the CodeIgniter files to your *f:/public_html/* folder. Create a folder in *f:/public_html* called *CI* (for CodeIgniter) and add your files to that folder.

Your file structure should be:

```
├── public_html/CI/
      ├── application
      ├── system
      ├── user_guide
```

The files include an *application*, *system* and *user_guide* folders. The *system* folder should be left alone.

The *application* folder is where we'll build our application. Inside the *application* folder the main sub-folders to be aware of are *config*, *controllers*, *models* and *views*. These hold the respective files when working with the MVC architecture.

```
├── public_html/CI/
      ├── application
            ├── config
            ├── controllers
            ├── models
            ├── views
```

CONFIGURATION

Most frameworks required some configuration.

In *application/config/config.php* find `$config['base_url']` and change it to match your URL ie:

```
$config['base_url'] =
'https://homepages.shu.ac.uk/~<YOURSTUDENTNUMBER>/CI';
```

We'll config this to use `https` from the offset though it could be set to `http` if the application had no requirement for SSL.

Once you have added the files to *f:/public_html/CI/* you can view your site at:

```
https://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php
```

Here you should find the default welcome page.

NAMING CONVENTIONS IN CODEIGNITER

CodeIgniter has a style guide for naming conventions.

https://www.codeigniter.com/user_guide/general/styleguide.html#file-naming

These rules include Capitalizing Classes names and using underscores `_` instead of camelCasing for file and class names.

CONTROLLERS

Open the file *application/controllers/Welcome.php*. We can use this as the base for creating other controllers.

Create a new file in *application/controllers* called *About.php* and add the following:

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

class About extends CI_Controller {

	public function index()
	{
		$this->load->view('about');
	}
}
?>
```

Notice that this class `extends` the `CI_Controller` class. This is CodeIgniter's Controller class, so by inheritance our new class has all the methods of the Controller class that include `load()` and `view()`.

This controller will have the URI:

```
https://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/About
```

When called the controller will call the `index()` method. This will load a view. We need to create the view next.

VIEWS

In *application/views* create a view called *about.php* that contains some basic HTML. Note the file name is not capitalized.

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>About Us</title>
</head>
<body>
<h1>About Us</h1>
</body>
</html>
```

To test this page now visit:

```
https://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/About
```

SENDING DATA FROM THE CONTROLLER TO THE VIEW

To pass a variable to the view we use the second parameter of the `view()` method. Amend the `index()` method of the *About.php* controller as follows:

```
public function index()
	{
		$data['author'] = "Martin";
		$this->load->view('about', $data);
	}
```

The names given to values in the associate array are now available in the view. Add the following to the `<body>` of the *about.php* view file.

```
<p>
	<?php echo $author;?>
</p>
```

Data can be more complex than in the above example as we can send arrays.

Amend the *controller/About.php* controller to include an array ie:

```
public function index()
	{
		$data['author'] = "Martin";
		$data['team'] = ['Bob', 'Jane', 'Fred'];
		$this->load->view('about', $data);
	}
```

And then in the view we can loop the *team* array with a `foreach` loop ie:

```
<ul>
	<?php foreach($team as $team_member){
		echo "<li>{$team_member}</li>";
	}?>
</ul>
```

MODELS

In the above example we have created data in the controller. A purer MVC approach would be for that data to have been created in a model.

Create a new file at *models/My_data.php*.

```
<?php
class My_data extends CI_Model
{

	public function __construct(){
		parent::__construct();
	}

	public function getTeam(){
		$data = ['Bob', 'Jane', 'Fred'];
		return $data;
	}

}
?>
```

Here we have a method `getTeam()` that creates and return the array.

Amend the *controller/About.php* controller to replace the `array()` with the loaded model. The model and the call to the method getTeam() will need to be placed before the controller calls the view.

```
$this->load->model('My_data');
$data['team'] = $this->My_data->getTeam();
```

No changes are needed to the view.

Note on cases:

- Make the model class name Uppercase ie *My_data*
- Make the model php file name Uppercase ie *My_data.php*
- Load the model using the Uppercase file name `$this->load->model('My_data')`;

Later we'll use models with a database.

URLS IN CODEIGNITER

URL can be used to pass data and can do so in a more user / SEO friendly way that standard query strings.

A url in CodeIgniter not only calls a controller but a method and optional arguments/values.

```
http://example.com/[controller-class]/[controller-method]/[arguments]
```

Here is an example. Create a new Controller at *controllers/Team.php* as follows:

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

class Team extends CI_Controller {

	public function show($id){
		$this->load->model('My_data');
		$team = $this->My_data->getTeam();
		$data['team_member'] = $team[$id];
		$this->load->view('team', $data);

	}
}
?>
```

Notice the lack of an `index()` method but the use of method called `show()`. Notice that `show()` also accepts a parameter.

Create a view called *view/team.php* to include:

```
<?php echo $team_member;?>
```

Then call the URL:

```
http://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/Team/show/2
```

The `Team` controller is called, then the `show()` method and a value of 2 is sent to it.

Note: Nothing special about `show()`. It is just the method name chosen for this example.

If no value is given ie:

```
http://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/Team/show/
```

then CodeIgnitor will generate an error.

We can create a default value by amending the `show()` method with:

```
public function show($id = 0){
…
}
```

ADDING MULTIPLE VIEWS

A controller can load multiple views. In the same way a PHP developer might use includes to split elements of a HTML pages into reusable chunks we can do the same with CodeIgnitor.

Create a view in *view/header.php* with the following HTML.

```
<header>
      <ul>
            <li><a href="">Home</a></li>
            <li><a href="About">About</a></li>
            <li><a href="Team/1">Team Member</a></li>
      </ul>
</header>
```

Open the controller *controller/About.php* and load this view before the *about* view.

```
$this->load->view('header');
$this->load->view('about', $data);
```

View the page:

```
https://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/About
```

You should now see the header included. However, the page isn't properly formed HTML so we should build a *view/header.php* to include all the standard HTML expected at the top of a page. We can also use dynamic data to set the title of the page uniquely. We'll also take the opportunity to create a *view/footer.php* file. This will give the design flexibility for example allowing for the addition of Javascript files.

Review the files provided for you in the *starterViews* folder. See how these can work together to create a properly formed HTML file. Move the views in the ZIP *starterViews* folders into your application *application/views*.

Amend the controller *controller/About.php* as follows:

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class About extends CI_Controller {

	public function index()
	{
		$data['author'] = "Martin";
		$titleData['title'] = "About";
		$this->load->model('My_data');
		$data['team'] = $this->My_data->getTeam();
		$this->load->view('header', $titleData);
		$this->load->view('about', $data);
		$this->load->view('footer');
	}
}
```

CONNECTING TO A DATABASE

To connect a database needs some more configuration. There is a dedicated configuration file for working with databases at *application/config/database.php*. Locate the `$db` array and amend the values to match your MySQL database.

```
$db['default'] = array(
      'dsn' => '',
      'hostname' => 'localhost',
      'username' => '<YOURUSERNAME>',
      'password' => '<YOURPASSWORD>',
      'database' => '<YOURDATABASE>',
      'dbdriver' => 'mysqli',
…
```

Create a new model called *models/Movies.php* with a constructor as follows:

```
<?php
class Movies extends CI_Model
{
      public function __construct(){
            parent::__construct();
            $this->db = $this->load->database('default', TRUE);
      }
}
?>
```

Here we load in the default database settings.

Note: We will be using the mysqli driver that uses prepare statements.

Parent constructors are not called implicitly if the child class defines a constructor. In order to run a parent constructor, a call to `parent::__construct()` within the child constructor is required.

Add a method to query the database for all the films.

```
public function getAllMovies(){
            $result_set = $this->db->query('SELECT * FROM movies');
            $data = $result_set->result_array();
            return $data;
}
```

The method `getAllMovies()` uses the `query()` method to build and run the SQL query. This produces a result set that can be formatted as an array via the `result_array()` method.

CREATE A CONTROLLER AND VIEW

We'll need a controller and view to display the database returned in the model.

Build a controller called *controller/All_movies* as follows:

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class All_movies extends CI_Controller {

	public function index()
	{
		$this->load->model('Movies');
		$filmList = $this->Movies->getAllMovies();
		$data['films'] = $filmList;
		$this->load->view('all_movies', $data);
	}
}
```

Then a view called *view/all_movies.php* that loops the data:

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
?><!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="utf-8">
	<title>Film List</title>

</head>
<body>

<h1>Film List</h1>

<ul>
	<?php
	foreach($films as $film){
		echo "<li>{$film['filmName']}</li>";
	}
	?>
</ul>

</body>
</html>
```

In the view we use a `foreach` loop to extract the individual film names.

Things to try: Extend the above example to use the header and footer views as in the previous example.

WORKING WITH FORMS AND THE FORM HELPER

The HTML for forms can be built in views in the standard way. Forms can also be generated automatically by CodeIgniter using its Form Helper

https://codeigniter.com/user_guide/helpers/form_helper.html?highlight=form

The helper can be loaded into the Controllers individually with:

```
$this->load->helper('form');
```

Alternatively, helpers can be loaded throughout the application by editing the *config/autoload.php* file. In *config/autoload.php* locate `$autoload['helper']` and change to:

```
$autoload['helper'] = array('url', 'form');
```

We'll take the opportunity to load the url helper as well as form.

- Open *controller/Form_sample_from_helper.php* and its related view *view/form_sample_from_helper.php*. These are located in the ZIP *starterControllers* and *starterViews* folders so you will need to move them to your *controllers* and *views* folders.

Notice how in the view the form is built through methods such as `form_open()`, `form_input()` and `form_submit()`.

WORKING WITH INPUTS AND INPUT HELPERS

Forms give us a way to send data to our application using the PHP superGlobals `$_POST` and `$_GET`. Working with `$_GET` and `$_POST` is slightly different in the CodeIgniter world.

CodeIgniter has helper functions to making working with these superGlobals easier.

- Open *controller/Form_sample.php* and its related view *view/form_sample.php*.

Notice how the action of the form is:

```
<form action="Form_sample/someMethod" method="post">
```

and that this related to the CodeIgniter URI of:

```
http://example.com/[controller-class]/[controller-method]/[arguments]
```

When the form is submitted the method `someMethod()` inside of the `Form_sample` controller is called.

View this example at:

```
https://homepages.shu.ac.uk/~<YOURSTUDENTID>/CI/index.php/Form_sample.php
```

The method `someMethod()` uses the CodeIgniter `input->post()` method to retrieve the value from the form.

```
public function someMethod(){
      $firstname = $this->input->post('firstname');
      echo $firstname;
}
```

The CodeIgniter `input()` method is a helper that can be used to retrieve values from `$_POST`, `$_GET`, `$_COOKIE` and `$_SERVER`.

Where in standard PHP you might use:

```
$something = isset($_POST['something']) ? $_POST['something'] : NULL;
```

with CodeIgniter we can use:

```
$something = $this->input->post('something');
```

WORKING WITH $_GET

The use of querystrings in CodeIgniter is discouraged in preference to segmented URLs. However, if you still want your application to send data in the URL you can do so.

- Open the example *controller/Form_sample_get.php* and its related view *view/form_sample_get.php*.

Notice that the controller has a methods of `someMethod()` and `show()`. The `showMethod()` is called by the form and this redirects to the `show()` method using the values from `input->get()` to build a segmented URI that contains a call to `show()` and passes the parameter.

BUILDING A SEARCH

CodeIgniter has Query Builder class that is designed to allow selecting, inserting and updating of data with the minimum of scripting.

The full list of the features of codeIgniter's Query Builder class is available at:

https://codeigniter.com/user_guide/database/query_builder.html

To see an example of this we'll build a search page. Open the model *model/Movies.php* and add a `searchMovies()` method as follows:

```
public function searchMovies($filmName){
            $this->db->like('filmName',$filmName);
            $query = $this->db->get('movies');
            return $query->result_array();
}
```

Build a controller for the search in *controllers/Search.php*.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Search extends CI_Controller {
      public function index()
      {
            $this->load->view('search');
      }

      public function send()
      {
            $this->load->model('Movies');
            $searchTerm = $this->input->post('filmName');
            $data['searchTerm'] = $searchTerm;
            $filmList = $this->MyData-> searchMovies($searchTerm);
            $data['filmList'] = $filmList;
            $this->load->view('search', $data);
      }
}
```

Then create a view at *view/search.php*.

```
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
?><!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="utf-8">
	<title>Search</title>
</head>
<body>
<h1>Search</h1>
<?php
	echo form_open('Search/send');
	echo form_input('filmName', '');
	echo form_submit('mysubmit', 'Search');
	echo form_close();
	foreach($filmList as $film){
		echo "<p>{$film['filmID']} {$film['filmName']}</p>";
	}
?>

</body>
</html>
```

The view uses a `foreach` to loop the values pass to the view from the controller which itself gets them from the model.

IMPROVING ERROR HANDLING

The view will work but needs better error handling for when variables aren't defined.

In the view this could be done by placing the `foreach` loop inside a conditional statement as follows:

```
	<?php
	if(isset($searchTerm)){
		foreach($filmList as $film){
			echo "<p>{$film['filmName']}</p>";
		}
	}
	?>
```

This will only display the `foreach` loop is the `$searchTerm` is returned from the controller.

If no value is put into the search box then all records are returned whereas we shouldn't receive any. We can resolve this by declaring a variable in the controller to see if the search is valid.

Amend the Search controller `send()` method with:

```
	public function send()
	{
		$this->load->model('Movies');
		$searchTerm = $this->input->post('filmName');
		$data['searchTerm'] = $searchTerm;
		$data['validQ'] = $searchTerm !== "" ? true : false;
		$filmList = $this->Movies->searchMovies($searchTerm);
		$data['filmList'] = $filmList;
		$this->load->view('search', $data);
	}
```

We can now change the conditional logic in the view *view/search.php* to use the new `validQ` variable.

```
if(isset($searchTerm) && $validQ){
…
```

WORKING WITH THE QUERY BUILDER CLASS

In the above search example the model used:

```
public function searchMovies($filmName){
		$this->db->like('filmName',$filmName);
		$query = $this->db->get('movies');
		return $query->result_array();
}
```

This makes use of CodeIgniter's Query Builder class. This can be used to more easily build common SQL queries. For example a SQL query of:

```
SELECT title, content, date FROM mytable
```

Can be written as.

```
$this->db->select('title, content, date');
$query = $this->db->get('mytable');
```

SQL Queries can be run in the classic fashion ie:

```
$sql = "SELECT * FROM movies WHERE filmID = ?";
$result_set = $this->db->query($sql, array($id));
```

This uses `?` placeholders as used in mysqli and PDO prepare statements.

With CodeIgnitier's Query Builder class the above can be also done with:

```
$where['filmID'] = $id;
$result_set = $this->db->get_where('movies', $where);
```

To order the values in our search we can add the `order_by()` method as follows:

```
      public function searchMovies($filmName ) {
            $this->db->like('filmName', $filmName );
            // order them
            $this->db->order_by('filmName');
            $query = $this->db->get('movies');
            return $query->result_array();
      }
```

This would now produce SQL as follows:

```
SELECT * FROM movies WHERE filmName LIKE '%someval%' ORDER BY filmName;
```

We can change this to `DESC` with:

```
$this->db->order_by('filmName', 'DESC');
```

A full list is available here:

https://codeigniter.com/user_guide/database/query_builder.html

There are also helpers to be found at:

https://codeigniter.com/user_guide/database/helpers.html

Many of these are particular helpful when putting together CRUDing (creating, updating and deleting) queries. We'll explore this in the next lab.

DETAILS PAGE

Now we have a list of films the client will probably want to dig deeper into the data. With web applications this is traditionally done with a query string ie:

www.mysite.com/productDetails.php?id=234

where the `id` is retrieve via PHP's `$_GET`.

With CodeIgniter we can address this with the URL. As we've seen a url in CodeIgniter not only calls a controller but a method and optional parameters.

```
http://example.com/[controller-class]/[controller-method]/[arguments]
```

So for our needs we use:

```
http://homepages.shu.ac.uk/~cmsmjc/CI/index.php/Details/showFilm/41
```

We'll be linking to a view/controller combination of *view/detail.php* and *controller/Detail.php* and calling the `showFilm()` method of the controller passing a value (41 above).

Amend the *view/search.php* view as follows:

```
foreach($films as $film){
      echo "<p><a href=\"";
      echo 'YOURURL/index.php/Details/showFilm/'.$film['filmID'];
      echo "\">{$film['filmName']}</a></p>";
}
```

To get the view to display **YOURURL** correctly we can use the `base_url()` helper.

```
echo base_url('index.php/details/showFilm/'.$film['filmID']);
```

Fortunately, we used *config/autoload.php* to load this helper earlier. It will make sure that the paths we use are correct.

Now create a controller at *controller/Details.php* with a `showFilm()` method as follows.

```
public function showFilm($id)
      {
            $this->load->model('Movies');
            $film = $this->My_data->getDetails($id);
            $data['film'] = $film;
            $this->load->view('details', $data);
      }
```

The core of the controller is the `showFilm()` method above with expects a `$id` parameter. This is then send to the model's `getDetails()` method which we'll build next.

Open the model `Movies.php` and add the `getDetails()` method as follows:

```
	public function getDetails($id){
		$where['filmID'] = $id;
		$result_set = $this->db->get_where('movies', $where);
		//return $result_set->result_array();
		return $result_set->row();
	}
```

This uses the `get_where()` from the query builder class to query the database using the `$id` value passed as a parameter in the URI. The `row()` method is used as we only expect one record to be returned if we query by the primary key. The `row()` method returns the results as an object, therefore in the view we'll need to use the object operator ie `$film->filmName`. The different options for handling query results can be found at:

https://www.codeigniter.com/userguide3/database/results.html

Next create the view *view/details.php*. A simple view would use:

```
echo "<h1>{$film->filmName}</h1>";
echo "<p>{$film->filmDescription}</p>";
```

THINGS TO DO

Can you amend the *view/details.php* to display the images associated with the film?

Can you improve the security of the application by filtering the input accepted by the search?