6COSC022W Advanced Server-Side Web Programming
Lecture 3 Notes – Building a CodeIgniter Application

Introduction

Last week, we had a first look at a CodeIgniter application. We installed the base CodeIgniter code, and added a controller and a view, and showed how to link up the different components to make something display in the browser.

This week, we will continue looking at building CodeIgniter applications with controllers, views and models connected together. To help illustrate this process, we will, as a case study, build a quiz application called WhoAmI? After we have looked at a simple implementation, we will also look at two other CodeIgniter topics – URL segments and routes.

WhoAmI?

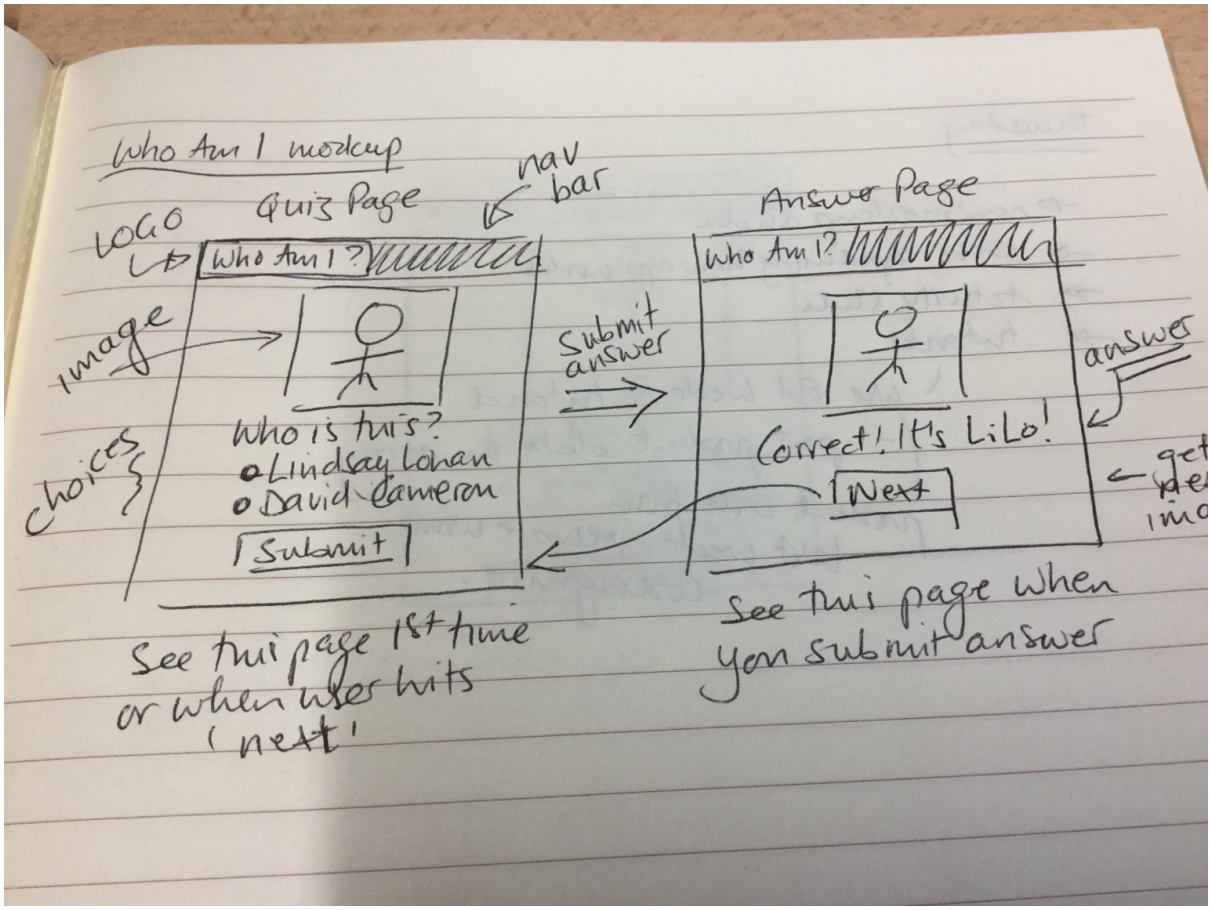
In the WhoAmI application, a user is presented with a picture of a person and two possible names. They just have to pick the right name.

Given this picture, for example,

The user could be asked who is this? Dwayne Johnson or George Clooney?

How do we start designing this application? One way is start with the interface and produce mockups of what we thing the interface might look like.

Here's a mockup I made of what I thought the application might look like to the user.

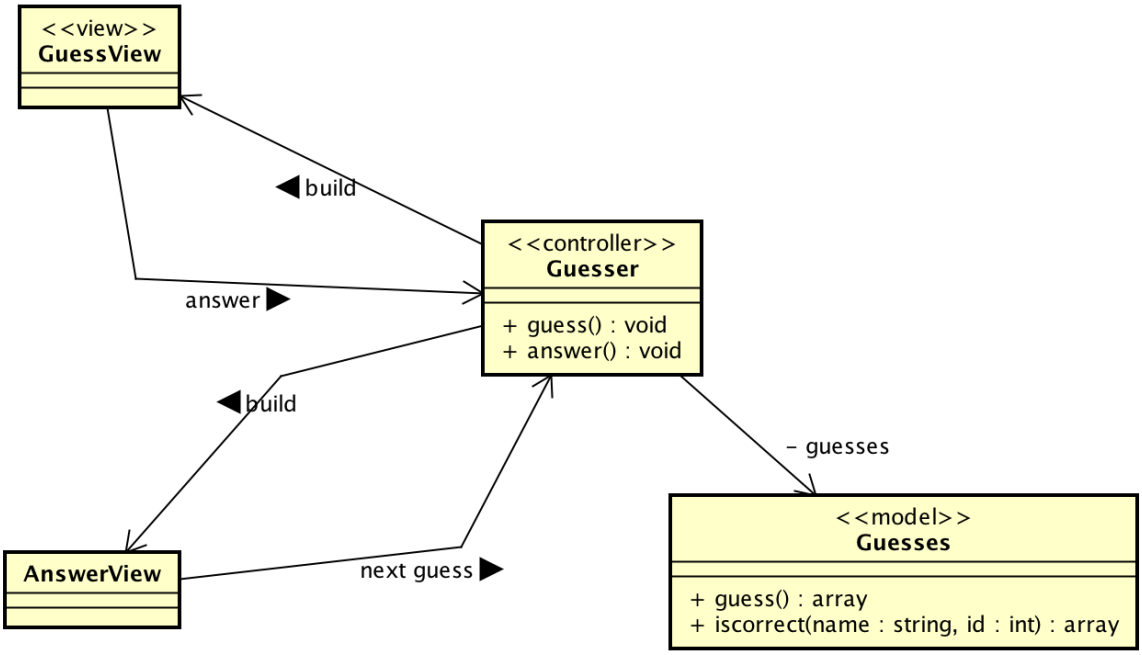


It's a bit rough, but it shows

1. what pages might be needed
2. what kind of data needs to be on each page (sample data is preferable to mock latin such as 'Lorem Ipsem' etc.)
3. the relationship between the pages and what gets passed in requests
4. Some explanation to help the diagram make sense

You might want to produce mockups that aren't as rough as this – in which case, online wireframing applications such as balsamiq, can be useful.

From this diagram, we can see that we expect there to be two views – one to display the picture and the two possible names (the 'guess' view) and one to display the answer (the 'answer' view). The user's guess is submitted from the 'guess' view to be checked and produce the 'answer' view. Since we will need a controller to generate these views and to send requests to, we'll call this the 'Guesser' controller. A class diagram that represents an initial object-oriented design of our application might look like this:



I've added in a model class, because, clearly, we need data for a guess and we want to check an answer – and this kind of data retrieval and data processing is what models are for.

How do we decide which controller and model classes we need? Usually, we would follow conventional object-oriented design guidelines and practice – each class should represent a coherent and cohesive concept. Here, everything is about guesses – so it makes sense to have one controller class for guesses and the same for models. If we were to introduce another concept – for example, Users – then we would think about having separate controllers and models for Users, in addition to those for guesses. Lastly, note that we are not limited to one model per controller. A controller can use as many model classes as it needs.

Creating the application

We will split the development of the application into 3 parts: the controller, the model, and the two views. One of the advantages of MVC is that it helps us structure our work into these distinct components based on their responsibilities

1. *The Controller*

The controller is responsible for making decisions: for example, which model should be loaded to process data, which views should be loaded to generate the next page that the user sees. Each controller is itself structured into a number of actions – each of which is implemented by a method function in the controller class. When we use a URL such as

http://www.somedomain.com/app/index.php/Mycontroller/myaction

we are saying that we want the Mycontroller class to be loaded and the action 'myaction' to be executed. This action or method function within the controller represents some service we want carried out, and the code in that function decides what models are needed and what views should be generated.

In our application, both the mockups and the class diagram show that there are two services the application needs to provide: (i) present a picture and two possible names for the user to select, and (ii) check the user's answer and tell them if they are right or wrong. Since both services are about guessing, we will put them into the same controller (see the Guesser class in the class diagram above).

You can find some sample code for such a controller here:
https://gist.github.com/simoncourtenage/f922d73e90f834e7d3f35a562d9a79ae.

This code illustrates two important things:

- Using models
- Accepting data from browsers via HTTP requests

Using models in the Controller

Last week, we saw how the controller connects to a view. When a controller wants to dynamically generate a view to send back to the browser, it just calls:

```
$this->load->view('welcome_message');
```

A controller connects to a model in the same way. The difference with views, however, is that once the model is loaded, it becomes one of the controller's attributes so that we can do things with it (like call its method functions). For example, we load a model like this:

```
$this->load->model('guesses');
```

This will cause CodeIgniter to find a file called 'Guesses.php' in the models sub-folder, and look for a class called 'Guesses' in that file. That class will then be loaded into the controller – the effect of this is to create an attribute variable in the controller called 'guesses' which will hold the model object. In other words, the controller will now have an attribute called

```
$this->guesses
```

And we will be able to use it to call method functions on the model object it holds. For example,

```
$this->guesses->guess();
```

(This is line 24 of the Controller code at the github URL above.)

Note that the limitation of this technique that CodeIgniter has of loading model objects into attributes with the same name as the model class means that we can only have one object of

a model class at a time (unless we engage in some tricky programming). But typically, this is not a significant limitation.

One point to note – the model class MUST be loaded before you use it. In my controller code, I make sure this is the case by loading the model class in the controller constructor. Since the constructor will always be executed before any method function is executed, I know that the model class will be loaded whenever it is used.

Accepting form data in the Controller

When a person is presented with the view displaying a picture of a person, they have to choose one of the two names displayed and click submit. The browser will then send an identifier representing the picture displayed (so we know who the user saw), and their chosen name. The application will then check the correct name associated with that identifier with the chosen name to see if they are the same.

This means we send back data (the identifier and the chosen name) in the HTTP request to the controller. The code for the view (the 'guessview' view) is here: https://gist.github.com/simoncourtenage/800f284f234eae053572df1461ce9873 .

Lines 31-51 show the form that contains the data to be sent back to the controller. We have a hidden form field containing an identifier (a number) that the controller will use to work who the user saw. There is also a radiobox field from which the user will choose one name. When the user clicks the submit button, the identifier and the name chosen from the radiobox are sent to the controller using the GET request method. See the action field of the form tag on line 31 – we send this data to the answer() method function of the controller. Since we don't add the method attribute to the form tag, the default request method is used, which is GET.

Now look again at the controller code, and lines 33-39. This is the answer() method of the controller. This is the code that is run on the server when the user clicks on the submit button. The first thing this function needs to do is retrieve the data it was sent by the browser, and this is shown on lines 35 and 36. Here we use the CodeIgniter `input` object, which is part of the controller, to access data sent using the GET request method. The input object contains a function called `get()`, which returns data held in $_GET. The string we pass as the argument to `get()` should be the name of a field in the HTML form – here we use 'id' (the name of the hidden field in the form) and 'name' (the name of the radiobox in the form).

Why use $this->input->get() when we can use $_GET directly? The value of using get() is that it not only looks up the data in $_GET but it also does other useful things such as sanitising the data to prevent things like SQL injection attacks - see here for more details: https://www.codeigniter.com/userguide3/libraries/input.html.

A similar function exists for data sent using the POST request method. If we know that POST is being used, then we can write in the controller function, for example:

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

The structure of the controller

As mentioned earlier, the controller needs to provide two services – displaying a picture and choices of names, and checking the user's answer. Each service will be a separate method function in the controller class. For example, checking the user's answer will be the answer() method:

```
28      /*
29       * This method tells the user if their guess was correct.  For this,
30       * we need the name the user selected and the actual 'id' of the
31       * person in the picture.
32       */
33      function answer()
34      {
35          $id = $this->input->get('id');
36          $guess = $this->input->get('name');
37          $answer = $this->guesses->iscorrect($id,$guess);
38          $this->load->view('answerview',$answer);
39      }
```

(taken from https://gist.github.com/simoncourtenage/f922d73e90f834e7d3f35a562d9a79ae)

Note what this action does and what it doesn't do. It gets the data from the request, asks the model to do something, gets the result back from the model and passes it to the generation of the view. It doesn't do any actual processing itself, no checking of the answer, nor any generation of output. It just acts as the glue between the processing in the model and the generation of the view using the result. It fulfils the role of being a controller, and as a result of this division of labour, is very simple.

The Model class

Once we have written the controller and we know what it needs the model to do, we can write the model. Sample code for a model is here:
https://gist.github.com/simoncourtenage/0701a420ed5bdf2d272bf1e6a2c6397b

Like the controller, the model extends a core CodeIgniter class:

```
class Guesses extends CI_Model {…
```

Also like the controller, the name of the class MUST match the name of the file: here, the code is in Guesses.php in the models folder.

The actual code of the method functions is not terribly important, but do read through it to make sure you understand how it works. As a quick shortcut, I have hardcoded the data about the people whose identities users have to guess in an array, and the identifiers are simply array indexes. Normally, we would use a database, but we will cover databases later. So, for now, we will make do with this shortcut.

The Views

The code for the guessview is here:
https://gist.github.com/simoncourtenage/800f284f234eae053572df1461ce9873

The code for the answerview is here:
https://gist.github.com/simoncourtenage/8f06f1c93b58707190e83a615bb43df9

Note that the answerview links back to the Guesser controller and the guess() method, allowing the user to have another guess.

Both views use data passed to them from the controller using an associative array. As discussed in the Week 2 lecture, the keys of the associative array passed by the controller to the view become variables in the view.

Look at lines 37 and 38 of the controller. These are in the answer method function. Line 37 calls the iscorrect() method on the model and gets back a result. Lines 57-59 of the model show that this result that is returned to the controller is an associative array with the keys 'iscorrect', 'image' and 'name'.

On line 38 of the controller, this result is passed as the second argument to the view() function that loads the view. Hence, within the view itself, we now have variables $iscorrect, $image and $name. You can see these being used on line 12, 16, 20 and 28 of the answerview file.

Once all this code is in place, and in the right files in the right sub-folders, we have a complete guessing application. We can start playing it by going to:

https://courtes.users.ecs.westminster.ac.uk/guessinggame/index.php/Guesser/guess

URL Segments

We have seen that we can pass form data to a CodeIgniter application using GET or POST and access it using the controller input object with lines like:

```
$this->input->get('field_name');
```

and

```
$this->input->post('field_name');
```

There is another way to pass data to a CodeIgniter application from a client browser. This is through parts of the URL called, in CodeIgniter, *segments*.

For example, consider the following URL:

http://www.somedomain.com/ciapp/index.php/Welcome/index

This is an ordinary CodeIgniter URL. The actual file being request is the index.php file in the folder ciapp. But there is extra path information after index.php – the strings "Welcome" and "index". The actual term that CodeIgniter uses for these extra strings after 'index.php' is *segments*.

We know that CodeIgniter uses the two segments after index.php to locate which controller class to load and which function in the class to execute. But we can also add further segments and access these within the controller. For example, we could have a URL such as:

http://www.somedomain.com/ciapp/index.php/Shop/browse/laptops/123

In this URL, the classname is Shop, and the method function we want to execute in the controller class Shop is browse(). But we have two more segments, "category" and "123". In the browse() function, we can access these values using code like this:

```
$producttype = $this->uri->segment(3); // returns value "laptops"
$prodcode    = $this->uri->segment(4); // returns value "123"
```

Note the numbers used with $this->uri->segment() – the segments are numbered from 1, where 1 = the controller class, 2 = method function and so on.

So we have two ways to pass data to controllers from the browser:

1. Through form data (in the query string if using GET or as data in the request if POST)
2. Through URL segments

When you would use one or the other? I suggest that if your data is hierarchial – as in the example above (product category "laptops", and laptop product code "123") – then segments might be a good choice. If, however, data is non-hierarchial, then form data is better suited.

Routes

The browser makes requests of the server-side part of a web application by requesting URLs. For example, a URL such as

http://www.exampleshop.com/index.php/store/getlaptops

is used to make a request of the server-side part of the web application for exampleshop.com, where, clearly, we want to see a view that displays laptops.

We could create a 'Store' controller class with an method function called 'getlaptops()', so that when the CodeIgniter application receives the request, it loads the Store controller class and executes the 'getlaptops()' method function within it to generate that view.

However, this approach, although simple, has its issues. Firstly, it makes visible the structure of the code to the outside world. In general, this is not good practice. For example, if we decide to change the structure of the code, we may end up changing the URLs. Secondly, the URLs we end up with may make sense to us as developers, but may not help users navigate their way round the application.

For reasons like this, we prefer to create URLs that abstract away from the physical detail of the code – the names of classes and functions – and which present the user with URLs that make sense to them. We then map these abstract URLs to the actual code that will implement the request the URLs represent. This mapping is done via *routes*.

A route is just a mapping between a URL and controllers and their actions. Routes in CodeIgniter are specified in a configuration file called routes.php in the config folder. The contents of this file when you first install CodeIgniter are as follows:

```
$route['default_controller'] = 'welcome';
$route['404_override'] = '';
$route['translate_uri_dashes'] = FALSE;
```

We can add our own routes to this file. For example, given the guessing game application, suppose we want the URL to be:

http://somedomain.com/ciapp/index.php/whoami/

but the actual URL with the real controller and method function names is

http://somedomain.com/ciapp/index.php/Guesser/guess

Then we write in the config/routes.php file

$routes[‘whoami’] = ‘Guesser/guess’;

Then any URL containing the segment ‘whoami’ will be remapped to the segments ‘Guesser/guess’.

More information on routing in CodeIgniter can be found at:
http://codeigniter.com/userguide3/general/routing.html