**CODEIGNITER I**

**Why PHP MVC**

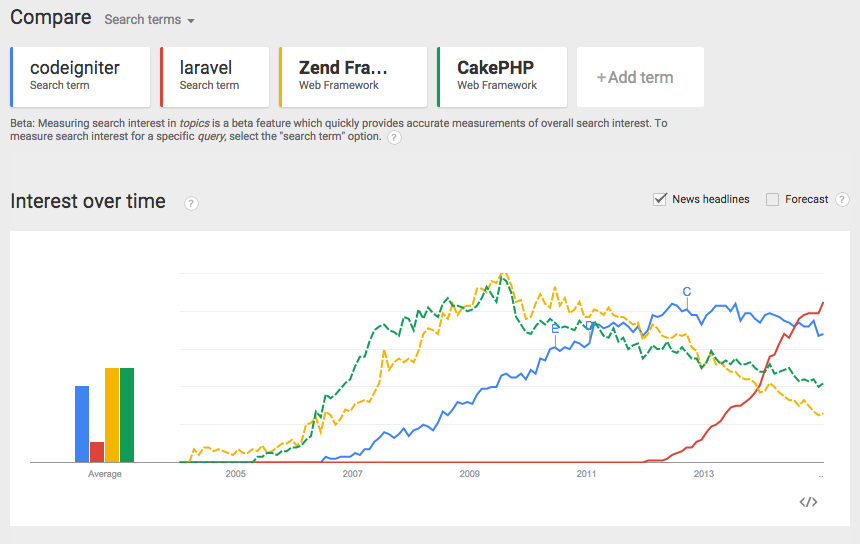
MVC (Model-View-Controller) architecture is a way of organizing server-side code to help create robust web applications. You can imagine that creating all of the PHP code for a website like Facebook using the procedural techniques that we have learned in the past could get pretty messy. By combining OOP with our knowledge of the different web components, we can organize our code to be more modular and we can use an MVC framework to help us.

Most ALL enterprise level web applications are built using an MVC framework and it's CRITICAL that you understand how to build your web application using an MVC framework.

**Why are we teaching CodeIgniter?**

CodeIgniter is one of the most popular MVC frameworks written in PHP. Zend and CakePHP, which are two other popular frameworks in PHP, are not as robust and could take you days/hours to install them properly. For Zend, you also need a dedicated server to install its frameworks whereas, for CodeIgniter, you can simply download it and install it anywhere (even in any shared hosting plan). Because of its popularity as well as the high use of CodeIgniter in enterprise level PHP applications, we are teaching you CodeIgniter as your first MVC framework.

See Google Trend below to see how CodeIgniter is one of the most popular frameworks of choice.



Although Laravel seems to be gaining quite a bit of momentum, most enterprise level web applications that are written prior to 2014 are most likely written in CodeIgniter, hence our reason for teaching you this. In other words, most companies that are looking for PHP developers are most likely looking for someone that knows CodeIgniter, Zend, or CakePHP, unless they are working on an app that's built after 2013. Plus, once you learn CodeIgniter and master Rails in the future, learning Laravel will be very easy and probably only take you a day or two.

**Objectives**

Our objectives are for you to:

* Understand what the different components in an MVC framework are and how they interact
* Understand the path that an HTTP request goes through in the framework and how the response is generated and returned.
* Understand the difference between Procedural code and MVC code and the advantages of both.
* Understand how OOP works in the MVC framework
* Build a website with MVC framework!

**Benefits of Using an MVC framework**

**Image analogy**

Imagine that you were given 100,000 image files and you and 2 other friends were to organize these image files in whatever way you thought made the most sense. How many ways would there be to organize these 100,000 image files? A lot of ways! You could choose to organize these files by the theme of the image, and start categorizing each image, adding each image in respective categories/folders (e.g. Christmas, New Year, Summer Vacations, etc).

Then, someone new joins your group and you are given an additional 50,000 image files to organize, except this time, the new guy who joins your group wants to organize these files in completely different ways. The new person argues that you should organize these image files by the date the picture was taken, etc. You decide to re-organize all the images using this new directory structure.

Then you join company B, who has millions of photos and they have a completely different way of organizing these photos! For whatever reason, the first person who joined company B started organizing their photos by the color of the background of each image (e.g. black, white, yellow, blue, etc).

In this analogy, you can see how there are many different ways to organize things and different people would have different opinions about how things should be done. How you organize things also depends on your purpose/objective.

**Software**

Let's now think about software. When building software, there are many features we need to build (e.g. loading database, doing a query to the database, rendering HTML, displaying errors, writing files, doing string manipulations, doing math calculations, rendering JSON, etc). Let's say you had about 100,000 features that needed to be built but instead of putting all these 100,000 features in a single page, you decide to organize them into different classes (just like in the earlier analogy how you decided to create folders to store the image). How many ways are there to organize these 100,000 features into different classes? Probably unlimited ways.

In software, this poses a lot of problems if developers don't have rigid rules on how to group features. For example, imagine the following scenarios.

1. You join a company and the way they organized their codes are very hard to follow. To debug a simple feature, you actually have to go through at least 10-15 files to even find out what's happening. You also don't know why certain methods are inside a specific class because the name of the class doesn't make much sense (e.g. the previous developer named it class ACD, BCD, EEE, which probably made sense to that developer, but for you, it doesn't make any sense why certain classes are named that way)
2. You join a company who has 100+ classes and 100+ methods inside each class. The way these classes are named is confusing again and you were just given 20 new features to implement, but you can't even understand what most of these classes do, let alone where in the class you need to add the feature you want. It takes you a long time to understand why the previous developers wrote and why he/she did it, although you feel like there is a more intuitive way of organizing the codes.

There are countless ways where with just OOP, developers can organize their codes in so many different ways that when multiple developers are working together, it would benefit if some hard rules were implemented. That's the benefit of using a MVC framework. Using a framework, It enforces some rules, mainly that

1. all things that deal with business logic and database need to be inside a folder/class called **Models**.
2. all things that deal with HTML rendering need to be inside a folder/class called **Views**.
3. all things that deal with URL request need to be inside a folder/class called **Controllers**.
4. we'll have a single file that is in a way the master of all the URL requests and we'll have this file load all the necessary files before any web application is called.
5. we'll have a config folder where things related to configuring the app (database session, security encryption, etc) can be stored. This eliminates some of the configuration settings appearing randomly in the app.

Not only does enforcing these rules make it easier for multiple developers to work together but a framework also will have built-in functions/methods that a lot of developers probably need. So instead of rebuilding each of these functions/libraries on your own, you could just use pre-existing functions/methods.

**Disadvantages**

Are there some downsides to using a framework like CodeIgniter? There would be some downsides. Some we can think of are:

1. There is overhead to running a framework (as it loads a lot of files for each URL request). This could slow things down a little bit.
2. There may be a security bug with the framework itself (this has happened with even some popular frameworks like Rails where all apps built with that framework were vulnerable for hacker attacks) which you probably don't have control of.
3. Whoever is supporting the development of that framework may go out of business and your framework may no longer be updated.
4. Some of the methods/functions you are using were written by other developers so you will have less of a clue about what's actually happening behind the scenes. In a way, you're dependent on other people's code, which may not be so bad, but you have less control over what's happening in the code also.

In many ways, though, these disadvantages are not as significant as the benefits of using an MVC framework, and really to join any enterprise software company, they expect that you know how to use an MVC framework.

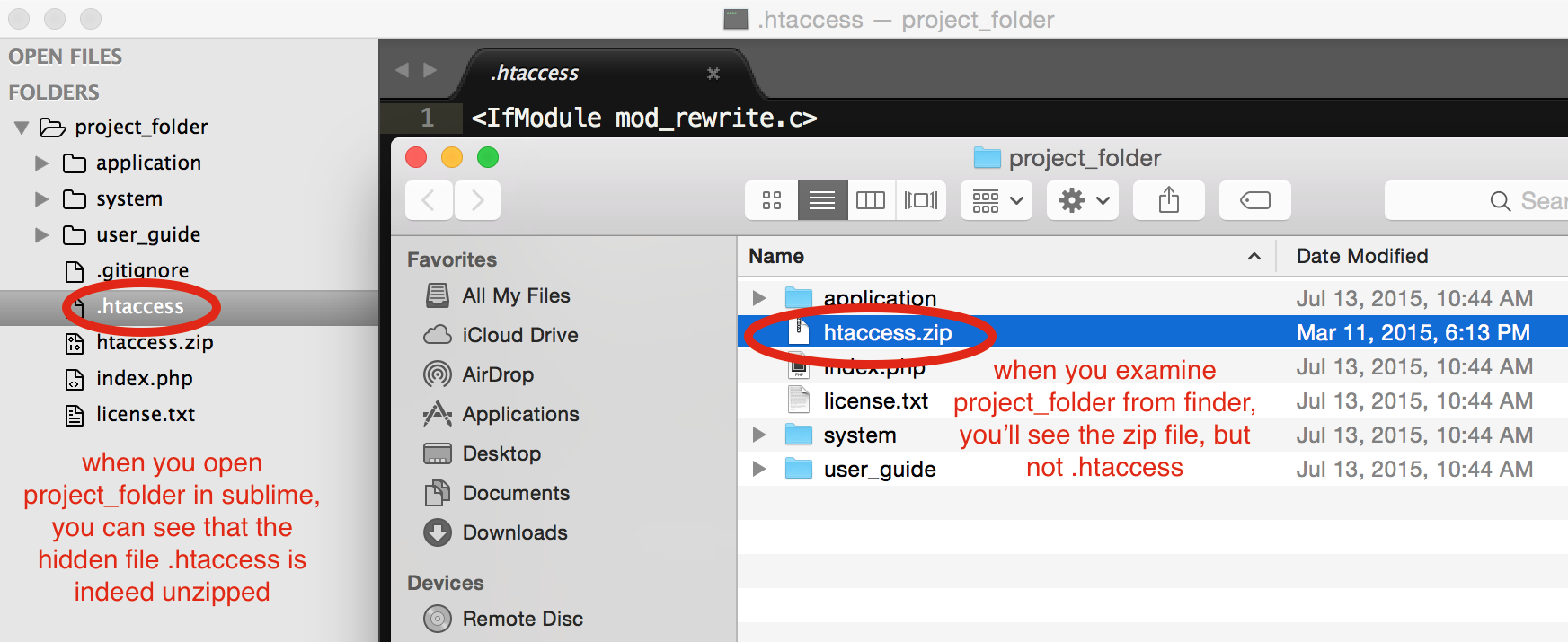
Another benefit by the way of using an MVC framework is that once you know how to use one MVC framework, it becomes a lot easier to pick up another MVC framework, even if it's done in another language. So for example, after you learned CodeIgniter, you can easily pick up other MVC frameworks like Django, Rails, Sinatra, .NET, Spring Framework, YII, etc.

**Installation**

Before you move on to your first MVC framework, [**you must download the CodeIgniter files from the platform here**](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3063/handouts/chapter3063_7612_CI3WithHTAccess.zip). Click on the link below to download a specific version of CodeIgniter we want you to use:

* [CIWithHTAccess.zip](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3063/handouts/chapter3063_7612_CI3WithHTAccess.zip) (click on the file name on the left to download this zip)

PC users, extract the files into your **wamp/www** folder. Mac users, extract the files into a **new folder**(for instance '*my\_first\_CI\_project*') and then point your MAMP's localhost at the folder you created. The version of CodeIgniter you downloaded already has a **.htaccess** file included, which allows us to hide the file types in our URL. This is a hidden file, Mac users may need to load up their project folder into Sublime Text to actually view it.



**For each project you create using CodeIgniter, you will copy these files into a new project folder.** For instance, if you're building a project called *The Wall*, and the project folder is called *the\_wall\_2.0*, ALL the CodeIgniter files will go inside. This will include the*application* and *system*folders, .*htaccess*, and any extra files or documentation.

**Directory Structure**

1. **application folder** - Contains the application you're building. This folder contains models, views, and controllers, which are in essence your main code of your web app.
2. **system/ folder -**The system folder has the default files/classes that are invoked every time CodeIgniter runs. *Never edit files in this directory.*
3. **index.php -**Initialize/loads resources needed to run CodeIgniter (Application and System files).
4. **assets folder -**The assets folder is where all static files (CSS, Images, and JavaScript) will be located.
5. **.htaccess file -**A configuration file to hide*"index.php"* in the URL, making the URL more user friendly.

**Installing Config Files**

**application/config/config.php**

config.php is where we manipulate the basic configuration settings of our application. Find the following $config lines and make the changes below:

// disable query strings by setting it to FALSE

$config['enable\_query\_strings'] = FALSE;

// give your CodeIgniter an encryption salt

$config['encryption\_key'] = '(00|\_3n(rYp+!0n\_k3Y';

// turn off session expiration during development

$config['sess\_expiration'] = 0;

**application/config/autoload.php**

By manipulating *autoload.php*, we can bring in libraries and helpers automatically, without having to require them multiple times throughout our application. Make the following changes:

// make sure you load the database and session libraries automatically

$autoload['libraries'] = array('database', 'session');

// autoload the URL helper as well

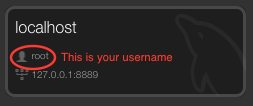
// URL will be helpful later when we deal with redirects

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

**application/config/database.php**

Now let's make sure CodeIgniter can properly connect to our database. **Important: This will change for every project you build, as you'll be connecting to different databases.**

First thing's first, check the username of your SQL server. This defaults as root, and you can check it in your MySQL Workbench.



If your username is anything other than root, put that as your username. *The hostname will ALWAYS be localhost, no matter what you named your connection in the MySQL workbench.*Password will default be root for Mac users and PC users will have the default of an empty string, or blank. Finally, the database will be the name of the schema you are using for your project.

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

// default password is root for Mac and an empty string ('') for PCs

$db['default']['password'] = 'root';

// this will be your schema name

$db['default']['database'] = 'sakila\_full';

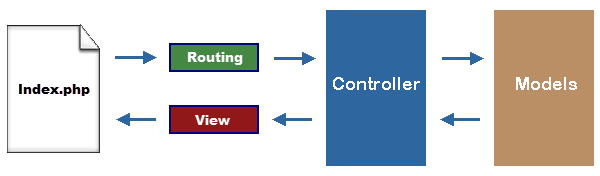
$db['default']['dbdriver'] = 'mysqli';

**Reminders**

1. **Files.**There are a lot of other CI files such as libraries and helpers but don't overwhelm yourself. Keep your main focus on learning how routing, controllers, views, and models work.. As you feel more comfortable with these fundamental concepts, you can start using other libraries available within CodeIgniter but initially these libraries/helpers can be more overwhelming than helpful. Also be sure to **never** **delete files!***Do not delete anything unless you are 100% sure that it's extraneous.*Deleting something that's important can lead to extremely difficult-to-debug errors!
2. MVC is OOP. Keep in mind that Controllers and Models are just Classes just like in OOP. Controllers are directly related to the URL request while models are directly related to the database.
3. Focus. CodeIgniter comes with a great user guide <http://www.codeigniter.com/user_guide/>. This resource is helpful but again you only need to know about 10-15% of the materials there. The other 85-90% of stuff in that documentation are not really necessary and often times are not needed at all. We tried to compile ALL the information you need to know to build practically any web app you can think of. Master these concepts first and for other things that you don't know how to do, you can look up in the user guide. For all of the assignments in our course, you should be able to build them with just what we teach you in this chapter.

**Having a hard time installing CodeIgniter to your PC?**

**Big Picture**

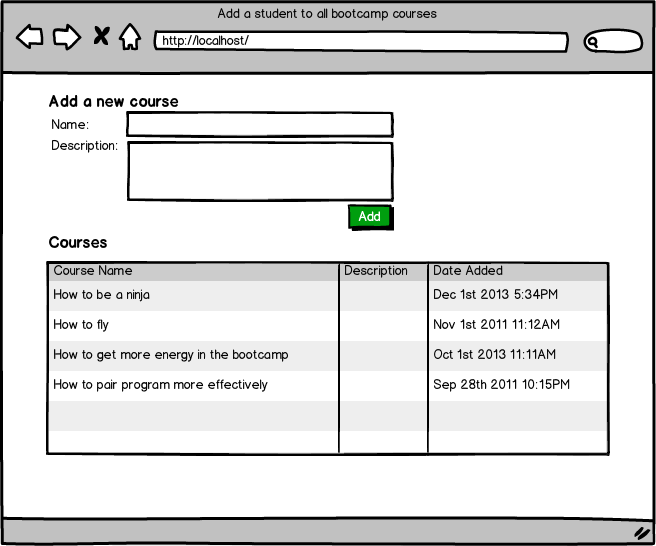


1. The client **requests** a page from the server (e.g. "/", "/dojo/ninja", "/awesome").
2. The server sends that request to **index.php** and passes the URL request's details. *index.php* loads the files necessary to use CodeIgniter.
3. CodeIgniter takes the URL and lets the **routing file** (e.g. routing.php) determine which **controller** (and method within the controller) should handle the URL request
   1. The first parameter in the URL (the "/dojo/" in "/dojo/ninja") signifies the **controller** that should handle the request. The second parameter ("ninja" in "/dojo/ninja") is the name of the **method** in that controller that should handle the request. For example "/users/new" would be handled by "Users" controller and the "new" method inside "Users."
   2. We can also set custom routing rules - shortcuts to controller and method combinations without using the normal "/controller/method" pattern. (E.g., "/add" instead of "/users/add".)
4. Once the **controller** takes over the request, it can talk to **models** (classes that process information and communicate with the database), and then hand information to a **view** (client-facing, HTML) file.
5. The **view**file displays information to the client using HTML, CSS, and JavaScript. It may also display information passed to it by the controller (such as a list of users or blog posts).

**Summary:** All URL requests are sent directly to *index.php*, which hands it over to the routing file, which determines the controller/method that should handle the request, and the controller finally renders the view file.

**Example**

Let's say that we wanted to create a simple web application below.  By looking at the wireframe below, can you determine how many URL requests you need for this web app?



If you answered two, you're right!

One request will be sent to render this HTML/CSS page and another request will be sent to process the POST data submitted via the form.

1. When the user goes to 'http://localhost/', you'll set up your routing file so that the request gets sent to the **Course Controller** and the **index method**. The index method would then render a view file that contains HTML/CSS information.
2. When the user submits the form, this will send the second request to "http://localhost/course\_add". You can set up your routing file so that this custom URL request is handled by the Course**controller** andadd**method**, which would process the POST data.  This method can redirect to "http://localhost/", starting back at step one and displaying the main page.

**Summary**

You don't have to understand yet how to do each of the steps above, but remember that instead of the form being submitted to a specific file, it now goes to a route, whether that is a controller/method URL or a custom route. The method in the controller will determine what to do with the information next.

# Controllers Overview

As discussed in the big picture tab, once an HTTP request is made, all requests are passed to index.php first. This file loads all the libraries, system files, and also loads the routing file (application/config/routes.php). This routing file then determines based on the URL which controller and which method in that controller class should handle the URL request.

The segments in the URL, following the Model-View-Controller approach, usually represent:

example.com/class/function/ID

1. The first segment represents the controller **class** that should be invoked.
2. The second segment represents the class **function**, or method, that should be called.
3. The third and any additional segment, represent the ID and any variables that will be passed to the controller (you can pass unlimited additional segments/arguments as long as they are separated by /).

Remember that now that we're using an MVC framework, the URL does NOT mean anything about which file should be executed/loaded by the interpreter. In fact, since all URL requests go to index.php now, the URL is now just one piece of information that is passed to the routing file, for it to determine which controller/method should handle that URL request. This is one of the key concepts of using an MVC framework.

Controllers

Controllers are stored in application/controllers/ folder (file name should be the same as the class name). To help you start with CI, it has a default controller class Welcome which is located in application/controllers/welcome.php

<?php

*class* Welcome *extends CI\_Controller* {

public *function* index()

{

$this->load->view('welcome\_message');

}

}

?>

When someone accesses localhost/index.php/welcome/index, the routing file will look for /application/controllers/welcome.php and if found, load this controller file and execute the index method in that controller. Similarly, if you were to go to localhost/index.php/dojo/ninjas, the routing file will look for /application/controllers/dojo.php and if found, load this controller and execute the ninjas method in that controller. Make sure you understand this key concept before moving on any further.

Each controller you create is a new class that extends from the CI\_Controller class included in CodeIgniter. Your controller class will inherit properties and methods from this default class, such as the \_\_construct() method.

#### Removing index.php from our URL request

Since we modified .htaccess file to remove "index.php" in the URL, we can directly visit the same page with:

localhost/welcome/index or localhost/dojo/ninjas

**Reminder:**If you're a PC User, make sure that Apache's rewrite\_module is enabled for .htaccess to take effect. For MAMP, this should already be enabled by default.

What would happen if you just visited localhost/welcome and you did NOT specify the name of the method for the routing file to call? Well. By default, CodeIgniter's routing file will open that controller and see if you have a method called 'index'. If 'index' method is found, then it runs that method. So you could run the index method by just calling localhost/name\_of\_the\_controller.

#### Methods and loading a view file

Inside the controller, we can have multiple methods to do different tasks within that controller. Common tasks in the controller's method include

* handling post data
* handling session data
* adding validations
* generating json data (for Ajax)
* loading appropriate view files (and passing data to the view files)
* passing or retrieving information from the database by calling 'models'

To have the method render a view file in /application/views/hello.php (view file in CodeIgniter is basically an HTML file with embedded PHP code), you do something like below:

*class* Dojos *extends CI\_Controller* {

public *function* ninjas()

{

$this->load->view('hello');

}

}

##### When someone visits localhost/dojos/ninjas, this should now display the contents in /application/views/hello.php. Note that the URL alone does not determine which view file gets called. In other words, although our URL is /dojos/ninjas, we had it so that was loading up hello.php!

Sometimes developers like to create a folder each for the view files used by each controller. For example, say that your web app had 5 controllers: dojos, codings, blogs, posts, and comments. Let's say that each of these controllers had 5 methods and each method in the controller had a unique view file it called. Having 25 view files all in /application/views could get a little confusing and it's often done where developers create a folder named after the controller in /application/views and store the view files for that controller there. For example, you could create a folder /application/views/dojos and put your hello.php there. You would then adjust your code so that it says...

$this->load->view('dojos/hello');

##### Although, for simple web apps, this may not be necessary, it's still good to know how developers create view files in general. In fact, some frameworks like Rails, this is the only way to do things. :)

In the demo videos (Counter and Number Game), we'll show you how we would start a new CodeIgniter web application from scratch, starting from installation all the way to developing the appropriate controller and methods to handle the different URL requests.

Video Outline:

1. installation (.htaccess file, folder structures, key point is that you should download and install CI for each new project you'll be creating, in this case, install CI for each assignment you will be working on)
2. brief demo on routing file, autoload.php, database.php and other config files
3. brief demo on the controller, method, and a view file
   1. show a final wireframe of what we would like to create
   2. outline the steps to create this.

#### Passing arguments to methods

You can pass arguments to the method through the URL, For example, say that our URL is now

localhost/dojos/ninjas/red/35

##### We can modify our program so that the ninjas method is like below:

public *function* ninjas($color, $id)

{

echo "Color: ". $color ." - ID: ". $id;

$this->load->view('hello');

}

**Reminder:**Only public functions can be accessed in the URL and the number of parameters allowed is unlimited.

#### Redirect

Sometimes you'll need to have the server redirect to another URL. For example, localhost/posts/create was the URL to handle some data submission. Once this is done, you wanted the client to be redirected to /posts/show. To do this, you would have your controller look like this:

public *function* create()

{

//some logic that may do some form validations, setting sessions, etc

redirect('/posts/show');

}

**Redirect**is one of the methods available in the URL helper, which we automatically had CodeIgniter load by default.  **Always put a slash ('/') before the route, as this starts the request from the base path (where we want it to).**

If you ever experience issues with your redirect, check out your config.php file in the config folder.  Make sure in your config.php file you have the following:

$config['base\_url'] = 'http://localhost:8888/';

$config['index\_page'] = '';

### Routes

#### Defining a Default Controller

The default controller for a CI application can be configured in application/config/routes.php. Default Controller is automatically loaded when a user visits localhost or your web domain name. To set your default controller, modify your application/config/routes.php to state

$route['default\_controller'] = "welcome";

When the default controller gets called when you visit say localhost/, it would look up for the default controller and for index method. If the controller can't be found or if the index method can't be found, CodeIgniter will generate an error for you.

#### Setting your own routing rules

##### We can also set custom rules. Let's say that when people go to localhost/awesome, we wanted the dojos controller and the ninja method to handle that URL request. We could set these custom routing rules by modifying application/config/routes.php to say...

$route['awesome'] = "dojos/ninja";

The important lesson is that just by looking at the URL structure, you can't be completely convinced which controller would be handling that request as we can always set custom routing rules for any URL type! In other words, don't always assume that the first word that appears after the domain name is the name of the controller that would be handling that HTTP request!

### FAQ

#### (IMPORTANT) How many controllers would I need to create? What are the best practices?

When you're building something very simple, you could put everything in a single controller. As soon as your application gets pretty complicated, it helps to create multiple controllers. For example, say you're building an application that needs to serve the HTTP request for 5 URL's. Would it make sense to have 5 different controllers to handle each URL request? Probably not.  What if you had about 50 URL requests.  Would it make sense for a single controller say Main Controller to handle all 50 URL requests? Probably not. **As a rule of thumb, you want to make sure that each controller has less than 10-15 methods**.  If you have a complex web application, you could say "Okay.I need to group different functionalities of my web application into different buckets."  You may then group certain functionalities to be handled by a Users controller and have the Users controller handle the URL requests that deal with User information. You could create a Session controller that handles the login/log off features.  You could also create say an Email controller that handles all the URL requests related to emails (receiving, sending emails, etc).

Really there's not just one way to build things. You could have, again, hundreds of methods in a single controller but when you have so many functionalities and many developers working on the project, it's always a good idea to group similar functionalities into different controllers so that it can be easily maintained and updated.

# Customizing Routes

Once you specify a default controller, you don't have to define a lot of routes as long as you stick [:name of controller]/[:method to be run inside the controller] convention to define your routes. What this means is that once you specify the default controller, then every time you create a new method in that default controller, a new route will be created for you in the following format [:name of controller]/[:method to run inside the controller]. For example, if we create a default controller of main and create a method called 'hello' then automatically, the route 'main/hello' will be added to the list of routes that our application supports. It will go to the main controller and run the hello method. Also, once we specify a default controller, if a user requests '/' then the index method in the main controller will be triggered.

// application/config/routes.php

$route['default\_controller'] = 'main';

// application/controllers/main.php

*class* Main *extends CI\_Controller* {

public *function* index() {

echo "Welcome";

}

public *function* hello() {

echo "I am the hello route";

}

}

Now, what if we want to make our URL look a little bit prettier? Instead of having the URL be 'localhost:8888/main/hello', what if we wanted it to just say 'localhost:8888/hello' ? If we don't want to use the default format of [:name of controller]/[:method to be run inside the controller] then we have to write custom rules in our routes.php file. If we make the following adjustments, then the user can access our hello method inside of our main controller using two different URLs: '/main/hello' and '/hello'.

// application/config/routes.php

$route['default\_controller'] = 'main';

$route['hello'] = 'main/hello';

Now, what if we wanted to pass a variable through the URL? What if a user requests 'main/hello/5' and we want to echo out '5'? If we are going to stick to the default format of [:name of controller]/[:method to be run inside the controller] we can just specify to the method to expect a GET variable and echo that out. We would only have to make changes in the controller file to look something like this:

*class* Main *extends CI\_Controller* {

public *function* index() {

echo "Welcome";

}

public *function* hello($num) {

echo $num;

}

}

But this doesn't make us too happy. What if we want our route to look like 'hello/5' and do the same thing? Now that we are once again diverging from the default format, we have to modify routes.php. To create a route like 'hello/5' and have the method in the controller use the GET variable, we would have to make the following changes in ***routes.php***:

// application/config/routes.php

$route['default\_controller'] = 'main';

$route['hello/(:any)'] = 'main/hello/$1';

You first specify which part of the URL you are expecting and how it matches up the to arguments you are passing in. You must specify it with *(:any)* to let the application know that you are expecting either a number or a string. There is an alternative *(:num)* if you know you are expecting a number, but by using (:any) you can work with **both** numbers and strings as GET variables. If you want to have a URL like '/hello/1/2' and use both of the GET variables 1 and 2, you can modify your routes and your controller like below:

// application/config/routes.php

$route['default\_controller'] = 'main';

$route['hello/(:any)/(:any)'] = 'main/hello/$1/$2';

*class* Main *extends CI\_Controller* {

public *function* index() {

echo "Welcome";

}

public *function* hello($num1, $num2) {

echo "You gave me " . $num1 . " and a " . $num2;

}

}

You specify in which order to pass on the variables using $1 for the first GET variable and $2 for the second GET variable. You have to use these variable names because CodeIgniter wants to know in which sequence you want to use the GET variables in your method in the controller.

## Custom Routes

**Do the following exercises without creating any view files or models.** We are only practicing how to set custom routes and have them routed to the desired controller and method along with any GET variables if necessary.

**Routing Exercises**

For the controller, although typically it is used to render a view file, it could simply echo a string.  For example,

<?php

*class* Main *extends CI\_Controller* {

public *function* index()

{

echo "Hi Class!";

}

}

?>

Before you start making your controller more complex, let's have you work on the following simple routing tasks:

1. When the user visits "/main/hello", have the request be handled by hello method in the class called 'Main' (in the controller folder).  Have this method simply echo "Hello World!"
2. When the user visits "/main/say/hi", have say method in the Main class (in the controller folder) handle this request.  Have this method simply echo "HI".
3. When the user visits "/main/say\_anything/\_\_\_", whatever was in \_\_\_, have this be simply be echoed.  For example, if the user visit "/main/say\_anything/awesome", the http response should simply be "AWESOME" all in capital.
4. When the user visits "/main", have this request be handled by the index method in the Main controller.  Have it simply say "I am Main Class!".
5. When the user visits "/" (e.g. localhost/), have this request be handled by the index method in the Main controller, echoing "I am Main Class!".
6. When the usr visits "/main/danger", have this request be handled by a method called 'danger' in the Main controller.  Have it simply redirect back to "/main"

**Views**

A *view* file in CodeIgniter is an HTML page with embedded PHP codes. The view files are located in the*application/views* folders.

Say that we had the following view file in /application/views/ ***dojo/hello.php***.

<html>

<head>

<title>Ninjas Page</title>>

<link rel="stylesheet" type="text/css" href="/assets/css/main.css">

</head>

<body>

<h1>Ninjas</h1>

<p>This view file is in /application/views/<b>dojo/ninjas.php</b></p>

<p>I am going to have php calculate 456 x 331115. The result is <?php echo 456\*331115; ?></p>

</body>

</html>

IMPORTANT - You'll note that CodeIgniter has an assets folder. This is where you should add the appropriate CSS, javascript, and image files. For example, the above view file calls main.css which is stored in*/assets/css.*

You could have the view file include other view files as well. For example. You could have */application/views/****partials/header.php*** or /application/views/***partials/footer.php*** that was included for each view file. To include these files, you would change your /application/views/***dojo/hello.php*** to look like

<?php $this->load->view('partials/header') ?>

<h1>Ninjas</h1>

<p>This view file is in /application/views/dojo/ninjas.php</p>

<?php $this->load->view('partials/footer') ?>

Now you could also have the controller load multiple view files but this it not a common practice and it's more common for the view file to load other partial view files themselves. It's not that this is the only way, but developers have adopted this as a common practice. Also you could require other PHP files by using require/include method but this only works for files that live in the same folder level as your view file and we think it's better that you use $this->load->view instead of using require/include.

The key thing to remember is that as you create a lot of view files and notice that a lot of your HTML is being repeated (for example the header and the footer), you could create separate view file just containing these repeated codes so that 1) your other view files look simpler/shorter and 2) so that when you have to make changes to the header/footer/etc, you can just change one file instead of changing each view file.

**Passing Data to the View**

A lot of times, the controller would prepare some data (data pulled from the database, data generated while validating the form, etc) and may want to pass this data to the view file. This can be done by **passing an array as the second argument to the view method**. For example:

public *function* ninjas()

{

$view\_data = array(

'topic' => 'CodingDojo Ninjas',

'description' => 'You guys rock. Keep up the good work!'

);

$this->load->view('dojo/hello', $view\_data);

}

Open the view file to be loaded (*/application/views/dojo/hello.php*) and change the text to variables that correspond to the array keys of*$view\_data.*

<?php $this->load->view('partials/header') ?>

<div id="container">

<h1>Topic: <?php echo $topic; ?></h1>

<p>Description: <?php echo $description; ?></p>

</div>

<p><?php $this->load->view('partials/footer') ?></p>

It's also a good practice to use *two-dimensional array* in passing data to the view, making it easier for us to organize our codes in the view.

//code in the controller

$view\_data['ninjas'] = array(

'topic' => 'CodingDojo Ninjas',

'description' => 'I love MVC!'

);

$this->load->view('dojo/hello', $view\_data);

//code in the view

<div id="container">

<h1>Welcome to <?php echo $ninjas['topic']; ?></h1>

<?php echo $ninjas['description']; ?>

</div>

**Reminder**:*$view\_data is not a constant variable to hold your data to be passed into a view, you can use any variable name.*

**Recommendations**

1. **group view files for each controller in a separate folder -**although this is not required, this is generally done by the developers community and it does help your view folder look more organized. For some of the simple apps you are doing, you may not see a lot of benefit doing it this way, but we recommend you follow this convention. This will help you when you go to more rigid MVC frameworks like Rails as again Rails will force all developers to follow this convention.
2. **naming the view file** - It's common to have the view file be named after the name of the controller's method. Again, this is NOT the only way to do this as earlier we demonstrated how Dojo->Ninjas method was loading up */application/views/dojo/hello.php* but this was to make sure you understand how the controller and the view work together (and how it's totally up to the controller which view file gets loaded). So for example, for Dojo->Ninjas method you could have the view file for that ninjas method saved as /application/views/*dojo/ninjas.php.*Similarly a view file for Posts->Create method (Posts Controller, Create method) you could name as /application/views/posts/create.php. This usually makes it a lot easier for developers to find your codes, which is a great thing.

**Redirect**

Another tool we have at our disposal is the *redirect()* function. Redirect() merely redirects the header to the URL we pass to it. A full URL to a separate website will act a lot like*this->load->view*, check out the example below:

if ($go\_to\_google == TRUE)

{

redirect("http://www.google.com");

}

Besides hard linking, we can use the *redirect()*to call methods within our controllers. *If you need to call one controller method from within another controller method, use redirect()!* ***Do not directly call public methods within your controller, always use redirect() for this!***

Consider the below example:

public *function* index()

{

$get\_data['data'] = $this->Data->getData();

$this->load->view('index', $get\_data);

}

public *function* logout()

{

$this->session->sess\_destroy();

$this->session->set\_flashdata('error\_message', "Logged Out successfully");

redirect('/');

}

Any time we need to send a user back to the index view, rather than writing out all of our code again, we can use the *redirect()*to call our *index* method. No matter how complex our code base becomes, we'll never need to rewrite *this->load->view* back to index!

**Input**

**Notice:  if your post data isn't working and you have Windows, read this:**

There is a wonderful bug with Codeigniter and Windows that for some people makes their post data (either $\_POST or *$this->input->post())* always evaluate to false. To fix this do the following:

1. Click your Wamp icon on the bottom right icon tray of your screen.
2. Hover over the 'Apache' tab and then the 'Apache modules' tab
3. Find the module in the list called '*rewrite\_mod'* and click it to check the box
4. This will restart your server and make the desired change and post data should now work!

**The input class**

The Input Class serves two purposes:

* It preprocesses global input data for security.
* It provides some helper functions for fetching input data and preprocessing it.

The Input class has the ability to filter input automatically to prevent cross-site scripting attacks. If you want the filter to run automatically every time it encounters POST or COOKIE data you can enable it by opening your *application/config/config.php* file and setting this:

$config['global\_xss\_filtering'] = TRUE;

**Using POST, COOKIE, or SERVER Data**

CodeIgniter comes with three helper functions that let you fetch POST, COOKIE or SERVER items. The main advantage of using the provided functions rather than fetching an item directly ($\_POST['something']) is that the functions will check to see if the item is set and return false (boolean) if not. This lets you conveniently use data without having to test whether an item exists first. In other words, normally you might do something like this:

if ( ! isset($\_POST['something']))

{

$something = FALSE;

}

else

{

$something = $\_POST['something'];

}

With CodeIgniter's built in functions you can simply do this:

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

**$this->input->post();**

1) The first parameter will contain the name of the POST item you are looking for:

$this->input->post('some\_data');

2) The function returns FALSE (boolean) if the item you are attempting to retrieve does not exist.

3) The second optional parameter lets you run the data through the XSS filter. It's enabled by setting the second parameter to boolean TRUE;

$this->input->post('some\_data', TRUE);

4) To return an array of all POST items call without any parameters.

5) To return all POST items and pass them through the XSS filter set the first parameter NULL while setting the second parameter to boolean;

6) The function returns FALSE (boolean) if there are no items in the POST.

$this->input->post(NULL, TRUE); // returns all POST items with XSS filter

$this->input->post(); // returns all POST items without XSS filter

**Forms**

To learn more about using the Input Class, let's work with forms and data submission. Let's create a view with the code below and save it in *application/views/add\_course\_page.php.*

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

<form action="/courses/add\_course" method="post">

<input type="text" name="title" placeholder="email">

<input type="text" name="description" placeholder="description">

<input type="submit" value="Add course">

</form>

</body>

</html>

Compared to basic PHP where the action of a form is usually a file *(action="login.php" etc)*, remember that in MVC Controllers are directly related to the URL. That is why *(action="/courses/add\_course")*would mean that we are going to submit the form into a Controller named as*Courses*with *function add\_course()*inside it*.*

*class* Courses *extends CI\_Controller* {

//displays the add course page

public *function* index()

{

$this->load->view('add\_course\_page');

}

//processes the adding of a course

public *function* add\_course()

{

$course\_details['title'] = $this->input->post('title');

$course\_details['description'] = $this->input->post('description');

$this->load->model("Course\_model");

$add\_course = $this->Course\_model->add\_course($course\_details);

if($add\_course){

echo "Course is added";

}

}

}

**Global Variables**

The Input Class can also be used to handle other global variables. If you are overwhelmed about these Global Variables, please go back and review the [PHP course](http://learn.village88.com/m/3/107/17287) where it was discussed.

//$\_GET

$this->input->get();

//$\_COOKIE

$this->input->cookie();

//$\_SERVER

$this->input->server();

**Session Class**

The Session class permits you maintain a user's "*state"*and track their activity while they browse your site. The *Session* class stores session information for each user as serialized (and optionally encrypted) data in a cookie. It can also store the session data in a database table for added security, as this permits the session ID in the user's cookie to be matched against the stored session ID. By default only the cookie is saved.

Other pointers

1. The Session class does not utilize native PHP sessions. It generates its own session data, offering more flexibility for developers.
2. Even if you are not using encrypted sessions, you must set an encryption key in your config file (*/application/config/config.php*) which is used to aid in preventing session data manipulation.

**How do Sessions Work?**

When a page is loaded, the session class will check to see if valid session data exists in the user's session cookie. If sessions data does not exist (or if it has expired) a new session will be created and saved in the cookie. If a session does exist, its information will be updated and the cookie will be updated. With each update, the *session\_id* will be regenerated.

It's important for you to understand that once initialized, the *Session* class runs automatically. There is nothing you need to do to cause the above behavior to happen. You can, as you'll see below, work with session data or even add your own data to a user's session, but the process of reading, writing, and updating a session is automatic.

**What is Session Data?**

A session, as far as CodeIgniter is concerned, is simply an array containing the following information:

* The user's unique Session ID (this is a statistically random string with very strong entropy, hashed with MD5 for portability, and regenerated (by default) every five minutes)
* The user's IP Address
* The user's User Agent data (the first 120 characters of the browser data string)
* The "last activity" timestamp.
* The above data is stored in a cookie as a serialized array with this prototype:

[array]

(

'session\_id' => random hash,

'ip\_address' => 'string - user IP address',

'user\_agent' => 'string - user agent data',

'last\_activity' => timestamp

)

If you have the encryption option enabled, the serialized array will be encrypted before being stored in the cookie, making the data highly secure and impervious to being read or altered by someone. More info regarding encryption can be found [here](https://ellislab.com/codeigniter/user-guide/libraries/encryption.html), although the Session class will take care of initializing and encrypting the data automatically.

Note: Session cookies are only updated every five minutes by default to reduce processor load. If you repeatedly reload a page you'll notice that the "last activity" time only updates if five minutes or more has passed since the last time the cookie was written. This time is configurable by changing the *$config['sess\_time\_to\_update']* line in your *system/config/config.php* file.

**Retrieving Session Data**

Any piece of information from the session array is available using the following function:

$this->session->userdata('item');

Where *item* is the array index corresponding to the item you wish to fetch. For example, to fetch the session ID you will do this:

$session\_id = $this->session->userdata('session\_id');

Note: The function returns FALSE (boolean) if the item you are trying to access does not exist.

**Adding Custom Session Data**

A useful aspect of the session array is that you can add your own data to it and it will be stored in the user's cookie. Why would you want to do this? Here's one example:

Let's say a user logs into your site. Once authenticated, you could add their username and email address to the session cookie, making that data globally available to you without having to run a database query when you need it.

To add your data to the session array involves passing an array containing your new data to this function:

$this->session->set\_userdata($array);

Where the *$array* is an associative array containing your new data. Here's an example:

$newdata = array(

'username' => 'johndoe',

'email' => 'johndoe@some-site.com',

'logged\_in' => TRUE

);

$this->session->set\_userdata($newdata);

If you want to add user data one value at a time, *set\_userdata()* also supports this syntax.

$this->session->set\_userdata('some\_name', 'some\_value');

Note: Cookies can only hold 4KB of data, so be careful not to exceed the capacity. The encryption process, in particular, produces a longer data string than the original so keep careful track of how much data you are storing.

**Retrieving All Session Data**

An array of all user data can be retrieved as follows:

$this->session->all\_userdata();

And returns an associative array like the following:

Array

(

[session\_id] => 4a5a5dca22728fb0a84364eeb405b601

[ip\_address] => 127.0.0.1

[user\_agent] => Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_7;

[last\_activity] => 1303142623

)

**Removing Session Data**

Just as *set\_userdata()*can be used to add information into a session, *unset\_userdata()* can be used to remove it, by passing the session key. For example, if you wanted to remove 'some\_name' from your session information:

$this->session->unset\_userdata('some\_name');

This function can also be passed an associative array of items to unset.

$array\_items = array('username' => '', 'email' => '');

$this->session->unset\_userdata($array\_items);

**Flashdata**

CodeIgniter supports*"flashdata"*, or session data that will only be available for the next server request, and are then automatically cleared. These can be very useful, and are typically used for informational or status messages (for example: "record 2 deleted").

Note: Flash variables are prefaced with "flash\_" so avoid this prefix in your own session names.

To add flashdata:

$this->session->set\_flashdata('item', 'value');

You can also pass an array to*set\_flashdata(),* in the same manner as*set\_userdata().*

To read a *flashdata*variable:

$this->session->flashdata('item');

If you find that you need to preserve a*flashdata*variable through an additional request, you can do so using the *keep\_flashdata()*function.

$this->session->keep\_flashdata('item');

**Helpful Tip Regarding Session**

We have seen some of our students try to modify the session data by doing something like below:

$this->session->set\_userdata('message') = "new message"

or

$this->session->set\_userdata('messages')[] = "new message"

Both will NOT work. Above basically retrieves the current message or messages(array) and sets it to be the new value but does not save the information back in the session. The proper way to do this is to do something like below:

$this->session->set\_userdata('message', "new message");

$this->session->set\_userdata('messages', array('new message'));

Some of our students have spent hours trying to fix problems caused by this so make sure you're using proper syntax to update the Session data.

**Running echo() or var\_dump() Before Changing the Session Data**

CodeIgniter seems to have a bug where if you echo anything or var\_dump any variable BEFORE you change the session variable, it refuses to change the session variable. For example, consider the codes below.

**Bad example of changing session**

echo "session counter is" . $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "changed session counter to be" . $this->session->userdata('counter');

When above codes run, it may seem like 'counter' is being updated to a new random number, however, regardless of how many times you reload the page and execute these codes, it will fail to update session['counter']. What happens is that any change in session after you echo or var\_dump is completely ignored... **To avoid this issue, always change the SESSION data and THEN echo or var\_dump the data.**For example below would be okay as you change the session and then echo something.

**Good example of changing session**

$old\_counter = $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "session counter was" . $old\_counter;

echo "changed session counter to be" . $this->session->userdata

copy

('counter');

**View Exercises**

Now, let's add some new functionalities to the Main controller (the one you worked during the Routing Exercises).

Add the following method:

* set it up so that accessing "/main/world", loads a world.php view file that has 3 beautiful pictures of some beautiful places in the world.  Make sure this view file is in the main folder (inside the views folder)
* set it up so that accessing "/main/ninjas", loads ninjas.php as a view file (again make sure this file is inside the main folder).  Have it display 5 pictures of awesome ninjas.
* set it up so that accessing "/main/ninjas/\_\_\_number\_\_\_", loads the same ninjas.php as a view file but where it shows the number of ninjas specified in the url.  For example, visiting "/main/ninjas/35" should display 35 Ninja pictures. It can display the same image multiple times.
* set it up so that accessing "/users/" would load views/users/index.php view file.  Have it say 'Hello Users!'.
* set it up so that accessing "/users/new" would load view/users/new.php view file.  Display a simple html form where a new user can be added and have this form be submitted to 'users/create'.  Have this form specify first name, last name and email address.  No need to have this form work on the back-end.
* set it up so that accessing "/users/create" directly in the browser will redirect to "/users".  However if accessing that url by the form created in "/users/new", have it say "This feature is coming soon!".  In other words, the only way the browser can access "users/create" is by posting some information to this url via a form (instead of visiting that url directly).
* set it up so that accessing "/users/count", would count how many times you visited that page and display that number.  For example, it should say "1" if you visited that that page for the first time.  It should say "3" if you visited that page 3 times.
* set it up so that accessing "/users/reset" would reset the count to 0 and display a message (through a view file) that the session has been reset.  Make it easy for the user to visit 'users/count' by displaying a button that the user can simply click and visit 'users/count' page.  After visiting 'users/reset', visiting 'users/count' should display 1.
* change is so that now the default controller is 'users'.  Now visiting 'localhost/" or "/", it should be handled by the index method in the users controller.
* set it up so that accessing "/users/say/\_\_\_\_\_", would display whatever is in the url but this time through the view file in /users/say.php.
* set it up so that accessing "/users/say/\_\_\_\_/\_\_\_number\_\_\_" would display whatever was in the url (the first part) in the view file but repeating the number of time specified in the 2nd part of the url.  For example, "/users/say/hello/10" should display hello '10' times.  Visiting "/users/say/sensei/500" should display 'sensei' 500 times.  If the 2nd part of the url is not a number, have it display an error message that says "Sorry.  This url does not meet our requirement."

As your final challenge, now, imagine that you had the following code:

*class* Users *extends CI\_Controller* {

public *function* mprep()

{

       $view\_data = array(

             'name' => "Michael Choi",

'age' => 40,

'location' => "Seattle, WA",

'hobbies' => array( "Basketball", "Soccer", "Coding", "Teaching", "Kdramas")

);

$this->load->view('users/mprep', $view\_data);

}

}

Set up users/mprep.php view file such that it displays a http response like this:

<h1>User Name: Michael Choi</h1>

<h2>User Age: 40, Location: Seattle, WA</h2>

<h3>5 Hobbies></h3>

<ul>

<li>Basketball</li>

<li>Soccer</li>

<li>Coding</li>

<li>Teaching</li>

<li>Kdramas</li>

</ul>

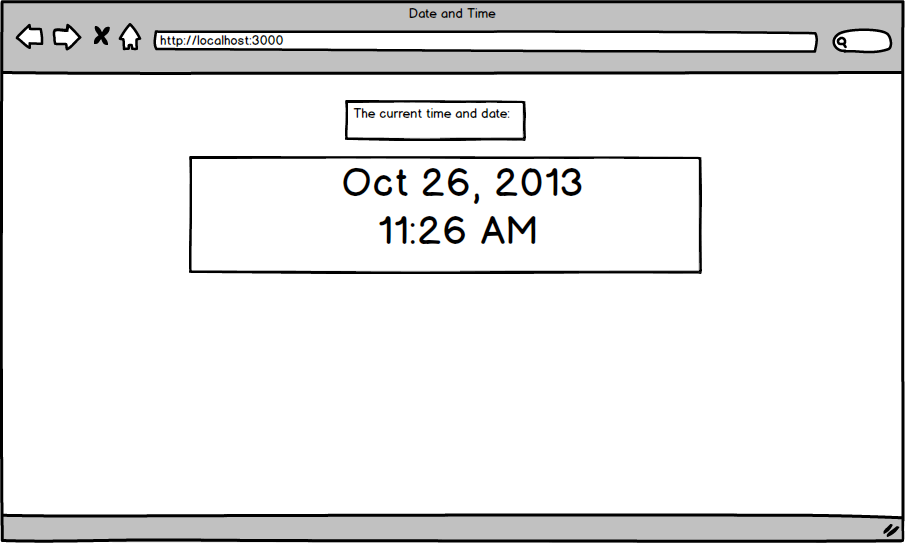
Set it up so that adjusting the data in the controller would also update what's shown through the view file.  For. example, if you added additional hobbies and changed the user's name and age, the updated information should show when the user visits "/users/mprep".   This exercise will be helpful as you later have your controller communicate to various models and where you'll be passing lots of information from the controller to the view file.

START WORKING ON THIS

## Assignment: Time Display

Create a controller called Times and have one method called main(). When the user goes to localhost:8888 the main method should render a view and pass that view a variable containing the current date and time and display that date and time on the view page.

When you're working with a web app, it's important that you understand how the data can be passed from the controller to the view. This assignment is to help you understand how this is done in CodeIgniter.



**IMPORTANT**: When you do this assignment, make sure that this page shows up when you visit just "http://localhost". In other words, figure out how you can have this view file render as default.

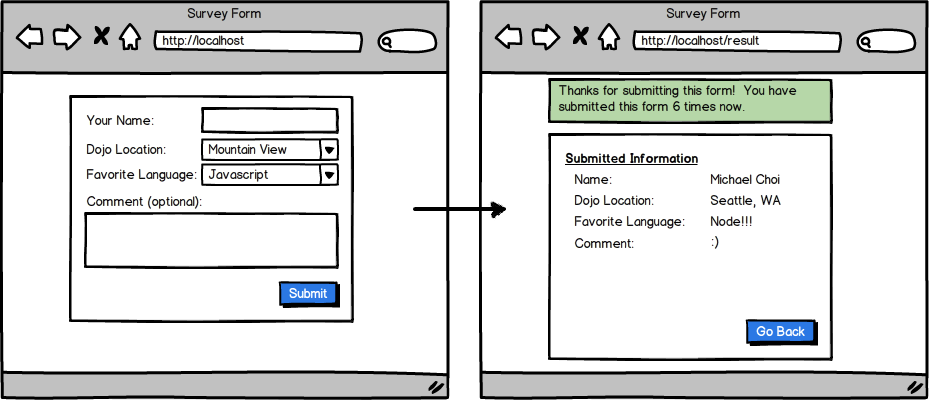
START WORKING ON THIS

**Assignment: CodeIgniter Survey Form**

Your next assignment is to build a new CodeIgniter (CI) application to do what is shown below.  Quick note: you'll need to download CI again for each new project or copy a fresh version each time you create a new CI project.  In fact, we recommend that you do this for each assignment you'll be working on and **NOT** have multiple controllers in the same CI folder for multiple assignments.

For any web app, it's critical that you understand how **form** data can be submitted, how **post** data and **session** data work. As you build the app described below, make sure you feel very comfortable with how information can be relayed between the form, controller, and the views **and** how session and post data are being handled.

Also for **ALL** of the assignments in this chapter, please do all assignments from scratch without relying on code from previous assignments.  Doing the assignments from scratch will 1) reinforce importance concepts you want to master and 2) get you to build things a lot faster as each assignment will help you optimize your process to make you a more efficient developer.



**IMPORTANT TIPS/GUIDELINES**

* Name your controller '**Surveys**' - note that we're using Plural for the controller name (not mandatory but it is helpful)
* Make sure that when you do this assignment that you don't use $\_POST nor $\_SESSION, but instead use *$this->input->post*and use*$this->session->userdata*or other session-related functions in CodeIgniter.
* No need to store this information in the database.
* Have your form submit to */surveys/process\_form*.
* Have the process\_form method in the Surveys controller redirect to '/result'.  Figure out how you can have the HTTP request of '/result' go to the Surveys Controller and Result method. (HINT: Change your *routes.php* so that '/result' is handled by the surveys controller's result method).
* **For PC users**: If your application is not able to handle POST data and if you're using WAMP, make sure you right click on the WAMP icon on the bottom right of your screen, go to Apache -> Apache Modules and make sure *rewrite\_module* is checked/enabled.

**IMPORTANT TIP (URL for this assignment as well as future assignments)** - Now, one thing we want you to be familiar with, and apply in this assignment, as well as future assignments is to NOT have a single URL handle BOTH the POST submission as well as render the view file. For example, the form that's rendered from *http://localhost* should be submitted NOT to /result but to say*/surveys/process\_form*.  The controller/method that handles /process\_form should do all the logic, process POST data, manipulate SESSION data, store things into SESSION, and redirect to another URL, say *'/result'.*  The reason we have a method to handle POST/SESSION and another method to handle the view file is mainly because it makes reading your code much easier but also because if the same URL handled both POST and the rendering of the view, when you reload that page, it RESUBMITS the form data, which is not a good thing as you probably intended that form data to be submitted just once.

**RUNNING ECHO OR VAR\_DUMP BEFORE CHANGING THE SESSION DATA**

CodeIgniter seems to have a bug where if you *echo*anything or *var\_dump*any variable BEFORE you change the session variable, it refuses to change the session variable. For example, consider the code below.

**Bad example of changing session**

echo "session counter is" . $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "changed session counter to be" . $this->session->userdata('counter');

When the above code runs, it may seem like *'counter'*is being updated to a new random number, however, regardless of how many times you reload the page and execute this code, it will fail to update session['counter']. What happens is that any change in session after you *echo* or *var\_dump* is completely ignored... **To avoid this issue, always change the SESSION data and THEN *echo* or *var\_dump* the data.**For example, below would be okay as you change the session and then *echo* something.

**Good example of changing session**

$old\_counter = $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "session counter was" . $old\_counter;

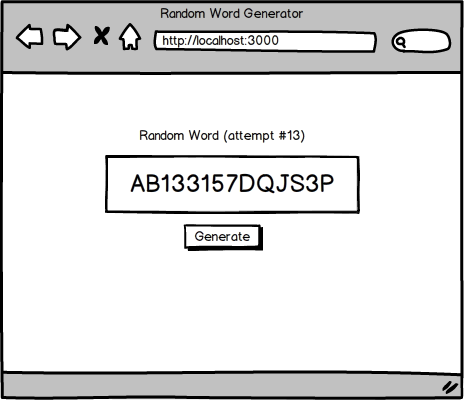
echo "changed session counter to be" . $this->session->userdata('counter');

START WORKING ON THIS

## Assignment: Random Word Generator

Create a new CodeIgniter project that shows a random word that is 14 characters long. The first time you use this app, it should say 'Random Word (attempt #1)'. Each time you generate a new random word, it should update the number inside attempt #\_\_\_.

The purpose of this assignment is to reinforce your understanding of form submission and the use of session.



We are also getting you ready to use Ajax with your CodeIgniter app in this assignment. Again make sure this shows up when you visit http://localhost and not any other URL (e.g. http://localhost/random etc).

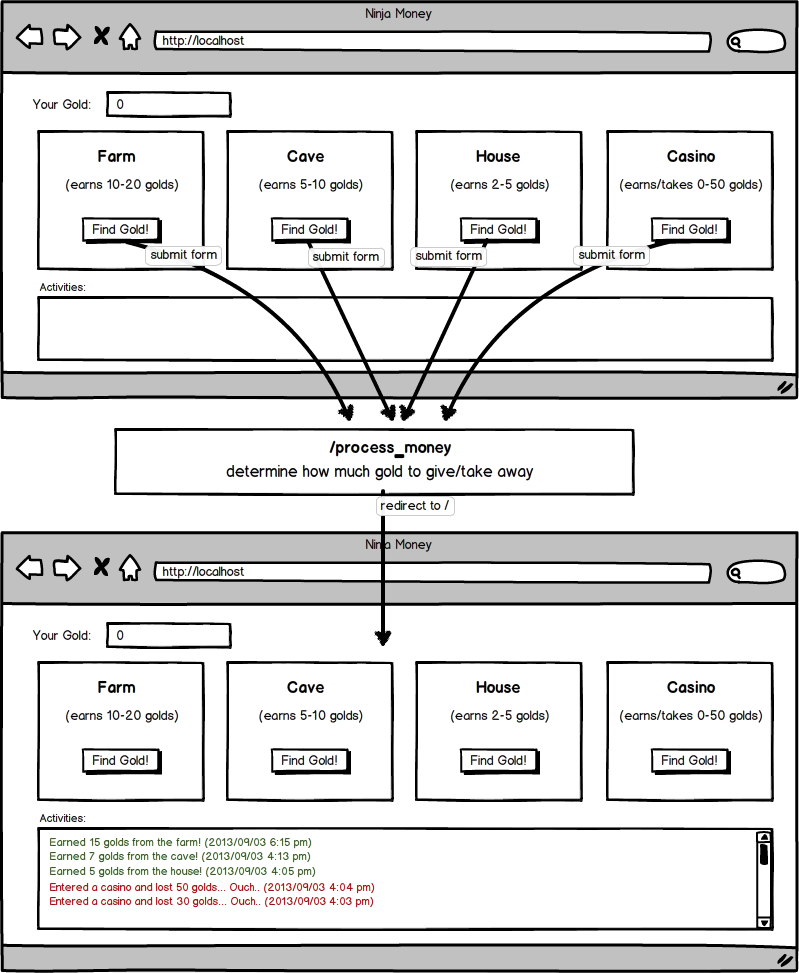
START WORKING ON THIS

**Assignment: Ninja Gold Game (CodeIgniter Version)**

You're going to create a mini-game that helps a ninja make some money! When you start the game, your ninja should have 0 gold. The ninja can go to different places (farm, cave, house, casino) and earn different amounts of gold. In the case of a casino, your ninja can earn or LOSE up to 50 golds. Your job is to create a web app that allows this ninja to earn gold and to display past activities of this ninja.

**Guidelines**

1. Refer to the wireframe below.
2. Have the four forms appear when the user goes to *http://localhost.*
3. For the farm, your form would look something like  
   *<form action="/process\_money" method="post">  
       <input type="hidden" name="building" value="farm" />  
       <input type="submit" value="Find Gold!"/>  
   </form>.*  
   In other words include a hidden value in the form and have each form submit the form information to /process\_money
4. Have a **process\_money** method determine how much gold the user should have (hint: you'll have to set up a custom routing rule)



Do NOT store the activity log in the database. Just save these logs in sessions.

Please make sure that...

1. when you visit, "localhost:8888/" you should see the page described above (in other words, we don't want to go to *"/gold/index"* or another URL to see this app).
2. make sure that the forms are sent to "/process\_money" and not any other URL. Again remember that you can modify the routing file to set custom URLs (have this custom URL).
3. make sure that you're not using*$\_SESSION* or *$\_POST* but that you're using *$this->input* and *$this->session*.
4. No need to store anything in the database. The activities should be stored in *$this->session*.

**Reminder:**

CodeIgniter seems to have a bug where if you echo anything or var\_dump any variable BEFORE you change the session variable, it refuses to change the session variable. For example, consider the code below.

**Bad example of changing session**

echo "session counter is" . $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "changed session counter to be" . $this->session->userdata('counter');

When the above code runs, it may seem like '*counter'* is being updated to a new random number.  However, regardless of how many times you reload the page and execute this code, it will fail to update*session['counter'].*  What happens is that any change in session after you*echo* or *var\_dump* is completely ignored... **To avoid this issue, always change the SESSION data and THEN echo or var\_dump the data.**For example, the code below would be okay as it changes the session data and then echoes something.

**Good example of changing session**

$old\_counter = $this->session->userdata('counter');

$this->session->set\_userdata('counter', rand(20,50));

echo "session counter was" . $old\_counter;

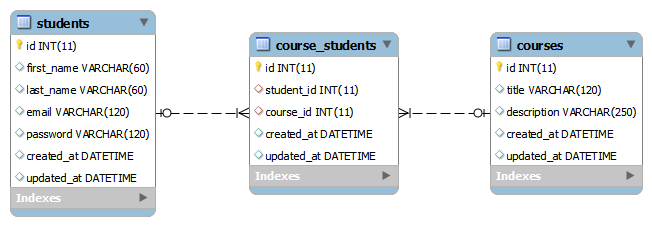
echo "changed session counter to be" . $this->session->userdata('counter');

START WORKING ON THIS

# Models

Models are PHP classes that are designed to work with information in your database. The class model contains functions to insert, update, delete and retrieve from your database.

For the sake of this topic and succeeding tabs, we will be using this database structure below. You can also import this database into your localhost using [codingdojo.sql](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3063/handouts/chapter3063_4019_codingdojo.sql" \t "_blank). This database is named as "codingdojo.sql".



## Connecting to Your Database

In order to run queries, we have to connect CI to the MySQL Server and select the database to be used. This is done by updating /application/config/database.php.

##### /application/config/database.php

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

$db['default']['password'] = 'root'; //use '' if you're using WAMP

$db['default']['database'] = 'codingdojo';

**Reminder:** If you are using WAMP, the default password is "", if you are going to put your site online, the hostname is usually the IP address of your server or your website's domain name.

## Anatomy of a Model

Model classes represent the tables in a database. When a new model class is created, it extends from the CI\_Model class included in CodeIgniter. By doing so, the new model class inherits certain methods and attributes from the default CI\_Model class. Let's create the Course model for codingdojo database with the code below and save it as course.php. This file will be in  application/models/course.php.  Put in the following code.

*class* Course *extends CI\_Model* {

*function* get\_all\_courses()

{

return $this->db->query("SELECT \* FROM courses")->result\_array();

}

*function* get\_course\_by\_id($course\_id)

{

return $this->db->query("SELECT \* FROM courses WHERE id = ?", array($course\_id))->row\_array();

}

*function* add\_course($course)

{

$query = "INSERT INTO Courses (title, description, created\_at) VALUES (?,?,?)";

$values = array($course['title'], $course['description'], date("Y-m-d, H:i:s"));

return $this->db->query($query, $values);

}

}

Here, we have three methods defined in the model: one to retrieve all courses, one to get a course with a particular id, and one to add a course.  Some key highlights:

* **result\_array()**- this method returns all the query results in an array.  If the query has multiple rows, it returns all of those rows in a multi-dimensional array.
* **row\_array()** - this method returns the first row from the query result.
* **Using ?** - in the query method, we use what CodeIgniter calls 'query binding' to have CodeIgniter insert the values specified in the second parameter of the query method AND also ESCAPE it properly to avoid any SQL injection. ALL of your queries should be done using this query binding.  For example running $this->db->query("INSERT INTO courses (title, description, created\_at) VALUES (?,?,?)", array('Dojo', 'Ninja', '2013-12-25')) would run the following query: "INSERT INTO courses (title, description, created\_at) VALUES ('Dojo','Ninja','2013-12-25')".

What you see above (running $this->db->query with result\_array(), row\_array() with query binding) is all you need to know to do everything you need. If you look at the CodeIgniter's documentation, it will give you dozens of other methods you can use with your model but you don't need any of them. Just spend time getting yourself familiar with the methods we teach above.

## Loading the Model in the Controller

The sample code below shows how Controllers and Models work together in order to get or add a new record in the database.

*class* Courses *extends CI\_Controller* {

public *function* show($id)

{

$this->output->enable\_profiler(TRUE); //enables the profiler

$this->load->model("Course"); //loads the model

$course = $this->Course->get\_course\_by\_id($id); //calls the get\_course\_by\_id method

var\_dump($course);

}

public *function* add()

{

$this->load->model("Course");

$course\_details = array(

"title" => "JavaScript",

"description" => "JavaScript Rocks!"

);

$add\_course = $this->Course->add\_course($course\_details);

if($add\_course === TRUE) {

echo "Course is added!";

}

}

}

## Highlights

1. **Loading the model** - We used '$this->load->model('Course')' to load the model Course in the Controller.
2. **Calling a method in the model** - To call a function from the loaded model we used $this->Model\_name->function\_name(parameter);
3. **Enable Profiler (IMPORTANT)** - CodeIgniter has a Profiler Class that can display queries that are run, SESSION/POST data as well as other useful information that makes debugging/troubleshooting much easier. We strongly recommend that you use this throughout your assignments. For more information, you may check this [CodeIgniter documentation](https://ellislab.com/codeigniter/user-guide/general/profiling.html).

For the create method, we passed a pre-defined array to the model. Although, in a real web application, your method in the Controller will most likely pass the POST data to the model, set some values into Sessions if needed, etc.

## Other Important Information

1. **MODELS and POST/SESSION** - Models should not directly access POST or SESSION data although the controller can pass POST or SESSION data for the models to access.  Again, the model's job is not to manipulate POST or SESSION (that's the controller's job) but merely to do the job necessary for updating the database or for retrieving proper information from the database.
2. **The Singular name for the models** - Models are usually named Singular while the Controllers are named Plural. Please use this convention without having to name your controller, say Controller\_User and your model Model\_User. Instead, name your controller 'Users' and name your model 'User'. This is a good convention and what most developers do.
3. **Have 'database' in your library autoloader** - If you don't have the database library in your autoloader, you won't have access to any of the db->query methods. Make sure you automatically load this library, especially if you've migrated to CodeIgniter 3. Check below for an example:

$autoload['libraries'] = array('database', 'session');

**Other Key Lessons**

1. **Role of Routing** - each URL request goes to*index.php*and is handed to the routing file. Routing file determines which controller/method should handle the URL request.
2. **Method in the Controller** - A method in the controller that handles the POST data should NOT render the view file. Although this is possible to do in CodeIgniter, it's strongly discouraged. Instead, have that method redirect to another URL, which is handled by a separate method that does render the view. For example if a form is submitted, say you submit to *'ninjas/form'*which handles the POST data, sets SESSION data, and redirect to say*'ninjas/success*' which is handled by success method in the ninjas controller that renders the view file.  Do NOT have the form method process POST, set SESSION and also render the view file (when the user reloads the page it submits the form each time which is not what you want to happen).
3. **Role of the Model**- The model should NOT access POST or modify POST data directly.  You can have the controller pass some or all of POST data to the model but in the model you should never directly access POST or SESSION data. Have the controller deal with SESSION data and not the model.
4. **What redirect does** - redirecting to another URL starts a new HTTP request, which means that new HTTP request is sent to *index.php*, routing file, controller and so forth. Using redirect in the controller does NOT automatically call the method in that controller, it just starts a new URL request.

# Validations

No matter what kind of web application you're building, eventually you're going to have to receive data from your users. In PHP and CodeIgniter, this data will come in from HTML forms. Much like how we did it back in PHP Procedural, we're going to need to validate this data and confirm it's in the correct format, as well as escape special characters and other hazardous code.

While we could do this validation with the if/else gates we wrote back in procedural PHP, there's a better way. We're working in MVC now, which means we can leverage other developer's code! **From now on, we'll use the form\_validation library to validate our CodeIgniter projects!**

Check out sample code below:

$this->load->library("form\_validation");

$this->form\_validation->set\_rules("first\_name", "First Name", "trim|required");

if($this->form\_validation->run() === FALSE)

{

//$this->view\_data["errors"] = validation\_errors();

}

else

{

//codes to run on success validation here

}

Note how the function set\_rules() of the Form Validation class takes 3 parameters, the name of the input field to be validated, a label for the error message, and finally the validation rules for that particular field.

$this->form\_validation->set\_rules("first\_name", "First Name", "trim|required");

Reading the above line of code, we learn that the **first\_name** field of our input form is **required** and it gets **trimmed**, meaning that excess white space characters get cleaned up. The string 'Brendan' would be the same as the string 'Brendan ', as the latter would have the extra space trimmed off.

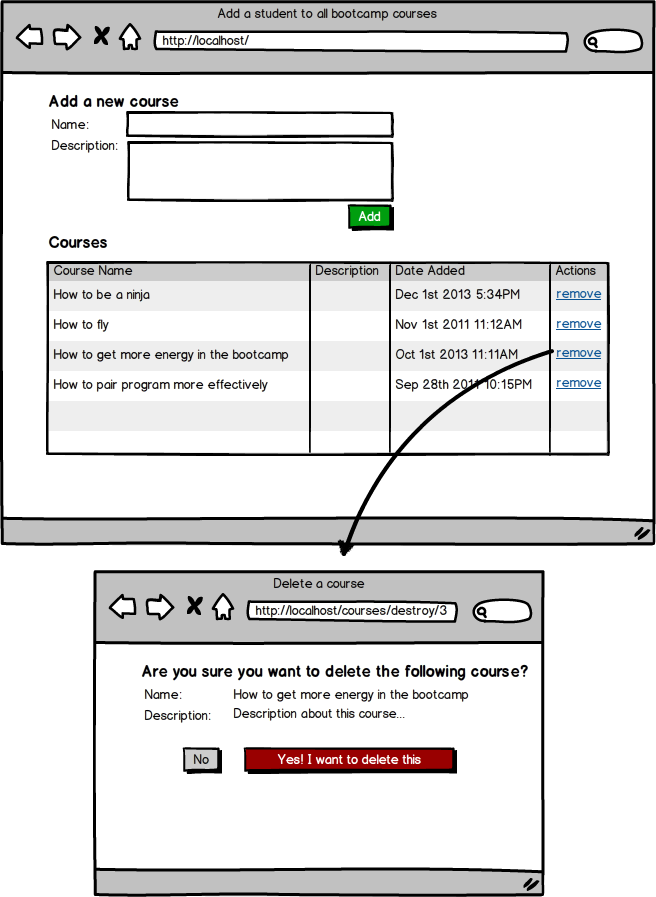
Next, we have the function run() contained within an if statement. run() executes all the validation rules, returning TRUE if all validation rules are met, otherwise it will return FALSE and place all validation errors in the function **validation\_errors()**.

$this->form\_validation->run()

To check out a full list of validation rules, please visit [this page](https://www.codeigniter.com/userguide3/libraries/form_validation.html#rule-reference). There's a ton of useful information there about the form\_validation library and how to further validate your code. Do not spend longer than 30 minutes to an hour reviewing this documentation.

**Assignment: Courses**

Create an application that implements the features outlined in the wireframe below.



**Guidelines**

* Have your model name be *'Course'.*
* Have this page show up when you visit *'localhost'.*
* Use *enable\_profiler* by going to your controller methods and adding the following line:
  + $this->output->enable\_profiler(TRUE);
* Have the form be submitted to '*localhost/courses/add'*.
* Make sure that when the form is submitted, it does NOT allow you to add a course without a course name. Make sure the course name is at least 5 characters. You can make the course description optional.
* When the remove link is clicked, note that the URL is '*localhost/courses/destroy/(course\_id)'*. The remove link is not a form/button but simply a URL link.
* Have the most recently added course appear on the top.
* (Optional) have a delete button next to the date added, which when clicked submits a form to /courses/destroy with the*id*of the course that needs to be removed. This is to get you prepared for RESTful architecture.

**Assignment: Semi-RESTful Routes**

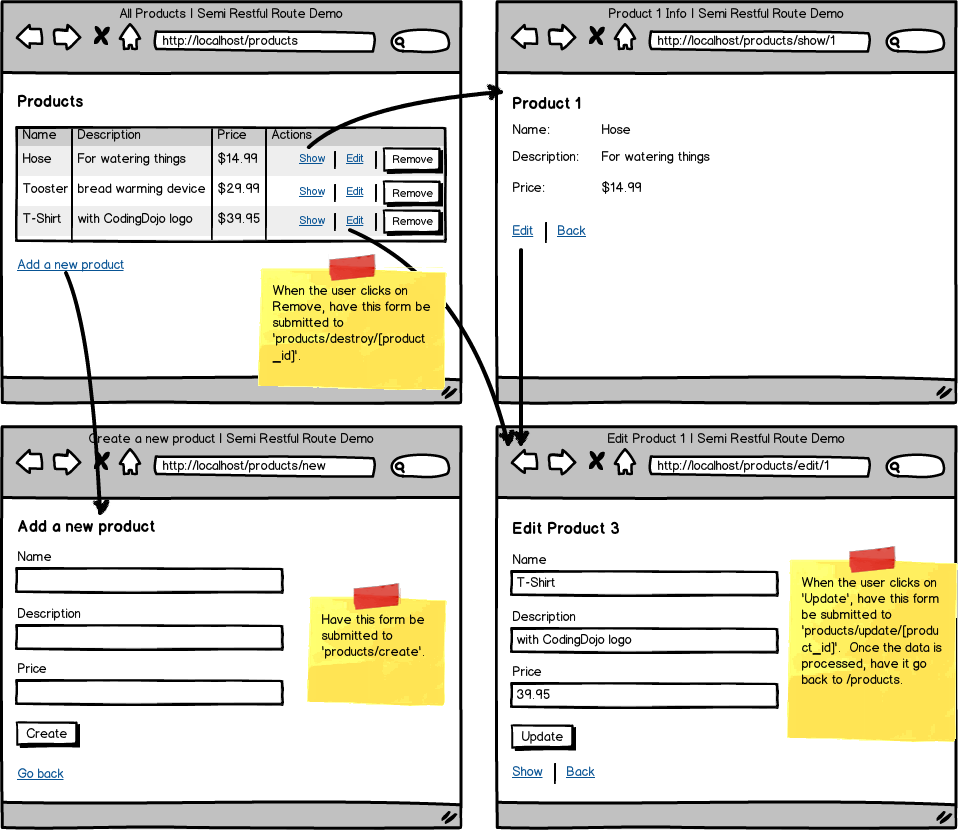
It's very common for a web application to provide the user interface for the users to create, read/retrieve, update, or destroy a 'resource'. For example, imagine you want to build a web application that allows the user to *create/read/update/destro*y products (think of a simple e-commerce website). There are many ways that you can build this web application. For example, you could have controllers called *product, products,* or simply *pd*. You could also have different methods that essentially do the same thing.  For example to display product information for product id 1, you could have the URL *'products/1'* provide this info or '*products/show/1*' or '*products/show\_info/1'* or *'products/display/1*' and so forth.

Since a lot of web applications are essential to do these CRUD (create/read/update/destroy) operations, you can imagine how confusing this could get if everyone followed different ways of creating controllers and methods names to do these operations.

REST or RESTful routing is a convention that was developed to outline some rules for everyone to follow. It's up to you whether you follow these rules or conventions, but it is strongly encouraged that you get familiar with how RESTful routes work as a lot of people in the industry are also following these rules.

Now, with CodeIgniter, it's not quite possible to do fully RESTful routing, so the exercise below is to help you get somewhat familiar with RESTful routes and so that when you get into another stack (MEAN, Rails, or Python), you're already somewhat familiar with the concept of REST.

Follow the instructions in the wireframe below to build this application in CodeIgniter.



**Make sure that...**

1. you name the controller plural - in this case*'products'*
2. name the model singular - in this case *'product'*
3. have 7 methods in the controller
   1. index method - to display all product info [this would need a view file]
   2. new method - to display a form that allows the user to create a new product [this would need a view file]
   3. edit method - to display a form that allows the user to update a product's info [this would need a view file]
   4. show method - to display a particular product's info [this would need a view file]
   5. create method - to process the form submitted from the new view [have this method process the POST data and redirect to *'/products']*
   6. destroy method - to process the form submitted from the index view to remove a particular product [have this method process the POST data and redirect back to *'/products'*]
   7. update method - to process the form submitted from the edit view to update that particular product's info.

First build all of this using HTML and CSS (if you have the time to style it), then create the appropriate methods in the controller as well as the model. Do not worry about adding validation rules in the beginning. Once all this is working without validations, add validation so that the price cannot be blank and must be numeric.

## Assignment: Login and Registration

Build a program using CodeIgniter that allows a student to register and login. Once the student logs in, display a page with information about the student (email, first name, last name). Have this be handled by a controller called **Students**.



#### **Validation Requirements**

1. Use the [CodeIgniter Form Validation](https://www.codeigniter.com/userguide3/libraries/form_validation.html#rule-reference)Library to validate all fields.
2. Check whether the email address already exists in the database.
3. Check whether the email address is in a valid format.
4. Check the password's length - make it at least 8 characters.
5. Check if the password and confirm password fields have the same value.

Refer to the sample code below in case you need a refresher on how to use the form\_validation library.

$this->load->library("form\_validation");

$this->form\_validation->set\_rules("first\_name", "First Name", "trim|required");

if($this->form\_validation->run() === FALSE)

{

//if you place your validations in the model, you can ship validation\_errors() back to the controller

//if you place your validations in the controller, you can put the validation\_errors() directly into session!

$this->session->set\_flashdata('errors', validation\_errors());

}

else

{

//code to run on successful validation here

}

The **set\_rules() function**of the Form Validation Library has 3 parameters (the name of the input field to be validated, a label for the error message, and the validation rules to apply to the particular field).

$this->form\_validation->set\_rules("first\_name", "First Name", "trim|required");

The **run()** function executes the validations and will return TRUE if all validation rules are met, otherwise it will return FALSE and all validation errors will be available in the validation\_errors() function.

$this->form\_validation->run()

More Features:

* When a student tries to login, check whether the email address and password combination is correct.
* Allow the student to log off.

### Other helpful Tips: Login and Logout with Session Class

Below is a simple implementation of utilizing **session**when handling the student Login and Logout features.

**Student Model**application/models/student\_model.php

*class* Student\_model *extends CI\_Model* {

*function* get\_student\_by\_email($email)

{

$query = 'SELECT \* FROM students WHERE email = ?';

$values = array($email);

return $this->db->query($query, $values)->row\_array();

}

}

**Student Login Page**application/views/student\_login\_page.php

<!DOCTYPE html>

<html>

<head>

<title>Login</title>

</head>

<body>

<?= $-rainbow">this->session->flashdata("login\_error") ?>

<form action="/students/login" method="post">

<input type="text" name="email" /-rainbow">-rainbow">>

<input type="password" name="password" />

<input type="submit" value="Login" /-rainbow">-rainbow">>

-rainbow"></form>

</body>

</html>

**Students Controller**application/controllers/students.php

*class* Students *extends CI\_Controller* {

//loads the login view

public *function* index()

{

$this->load->view('student\_login\_page');

}

//processes the student login

public *function* login()

{

$email = $this->input->post('email');

$password = md5($this->input->post('password'));

$this->load->model('Student\_model'); // or you can autoload

$student = $this->Student\_model->get\_student\_by\_email($email);

if($student && $student['password'] == $password)

{

$user = array(

'student\_id' => $student['id'],

'student\_email' => $student['email'],

'student\_name' => $student['first\_name'].' '.$student['last\_name'],

'is\_logged\_in' => true

);

$this->session->set\_userdata($user);

redirect("/students/profile");

}

else

{

$this->session->set\_flashdata("login\_error", "Invalid email or password!");

redirect("/students/index");

}

}

//simple profile page of a student

public *function* profile()

{

if($this->session->userdata('is\_logged\_in') === TRUE)

{

// load a view file to show them their information

}

else

{

redirect("/students/login");

}

}

//logout the student

public *function* logout()

{

$this->session->sess\_destroy();

redirect("/students/index");

}

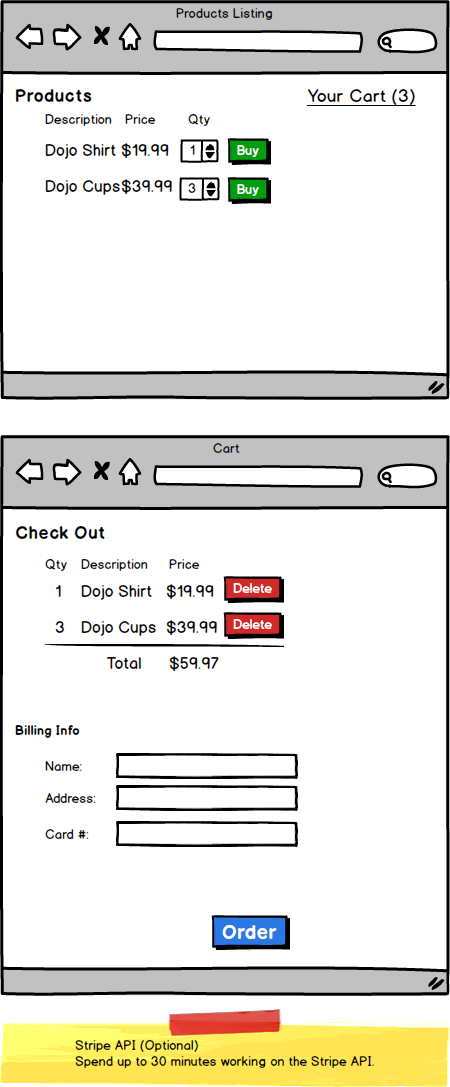
}

**Reminder:** Aside from using the Session class, the sample code snippets above also show how Models, Views, and Controllers work with each other along with the Input class ($this->input->post()) and URL Helper (redirect()).

## Assignment: E-commerce

This assignment is for you build a simple shopping cart website (see the wireframe below). Make sure you review the wireframe (determine which controller/methods you would create, build your ERD, and write your HTML/CSS before you do any coding). Make sure that the form that adds the item to the cart does not post the price in its form.

\*If you have time, connect the Stripe API.



**Form inputs**

For an e-commerce website, imagine that you had a form such as this

Buy this awesome T-shirt

<form action="..." method="post">

<input type='hidden' name='description' value='Tshirt' />

    <input type='hidden' name='price' value='19.99' />

<select name="quantity"> ... </select>

<input type='submit' value='Buy Now' />

</form>

If you were a hacker, can you think of how you could hack into this system and buy this T-shirt for 0.01?

One way you could easily hack a site like this is to open up the site, use Inspect Element, double click on the 19.99 field, change it to 0.01, and then to click on the submit button.  As the browser will allow you to modify any html output, you can easily adjust the value in the form, even if the form input's hidden.  When you modify the price and submit, the browser will send a http request with the following formation information:

* description - Tshirt
* price - 0.01

Is that what you wanted?

How could you protect your site from the user easily changing any of the input values?

**Industry insight**

Back in 2013, I remember teaching a group of students about this concept and how we can easily adjust any of the html values easily using Inspect Element.  Then I challenged the students to see if they could find an e-commerce website that had a form structured like this and to see if they really did allow them to buy items for $0.01 or whatever they wanted.  In about 30 minutes, students came back with multiple sites where sites had these security flaws.

Now, if you wanted to take advantage of these sites, you could easily build a crawler that stored all the html response and see if they had any input fields for the price.  Most likely, a lot of these sites would be vulnerable for these user modifications/hacks.

**Solution**

The key for fixing these type of security flaws is to not put any critical information in the form that the users can easily modify or put information in the form that's okay even if the users were to change it.  For example, instead of putting the price, what if you put in the product ID and also the quantity?  For example, maybe the form for buying the T-shirt could be as follows:

<form action="..." method='post'>

<input type='product\_id' value='100313' />

<select name='quantity'> ... </select>

<input type='submit' value='Buy Now' />

</form>

Now, for these forms, even if the user was to change the product\_id manually, it won't matter.  They would simply be buying a different item!

**Key takeaway**

In the next chapter, you'll learn about other security flaws and how to prevent these hacks.  As a general rule, it's never a good idea to completely trust the information that comes from the user.  Sanitize the data, put information that's okay for the users to change, and implement other security measures as you'll be taught later.

**CODEIGNITER II**

## Using Multiple Controllers and Models

On the previous assignments, you may have used one controller and model for all of your back-end code. As your projects get larger, you'll find that code will become difficult to organize and navigate when using only one controller/model. Instead, we can modularize our code by breaking it up into multiple files.  
  
The general rule-of-thumb is one controller and model for each table in your database. For example, if you are making a website that needs to keep track of users, products, and orders, you may have a controller and model for each. (A users controller and user model, products controller and product model, etc.)

Certain features or aspects of your app could also have their own controllers/models, such as an admin controller. You do not need to have a controller and model for every feature or piece of your database.

Having multiple controllers and models makes it much easier for you (and others!) to sort through your code. Your method names can now be more short and concise... and RESTful!

To navigate from one controller to another, simply redirect to a route that points to the desired controller/method.

There are multiple options for loading your models:

1. Load the model just prior to use:

$this->load->model("product");

$this->product->create($post);

2. Load the model in the controller's \_\_construct() method:

*class* Products *extends CI\_Controller* {

public *function* \_\_construct()

{

parent::\_\_construct();

$this->load->model('product');

}

}

3. Load the model in config/autoload.php:

$autoload['model'] = array('user', 'product', 'order');

#### Use multiple controllers and models on the rest of your assignments!

# Form Validations test

There are many ways to validate your form. Although there is no perfect way, there are best practices. We have separated the levels of form validations into three different levels.

## Level 1: Client-side Validation (View)



The first level is only validating your form from the client-side, which means validating from your view. We can do client-side validations (validations from a view file) by using JavaScript through jQuery. If this is the only validation that you have, your site is not very protected because a user might be able to bypass front end validations because all of the files are available for them in the browser. For example, let's say I know that the website is using jQuery to do the front-side validations. I can break all of that by opening up the JavaScript console and redefining the variable '$' to something else. Or there are some users who prefer to disable JavaScript on their browsers. This is why although having front-end validations provide great user experience for the user by providing instant feedback, it cannot be the only validations that you have.

## Level 2: Client-side Validation + Server-side (Controller)



By having the validations done in the controller, we are now doing validations on the server-side. Now you have validations on the client-side and also on the server-side. This makes your website secure because the user won't be able to bypass the server-side validation from the client-side. A lot of developers validate the form in the controller. In fact, even on CodeIgniter's documentation, they do their validations in the controller. If you do validations in the controller, you are at Level 2. This is a huge improvement from Level 1 because you are now doing validations on the server-side.

However, should the controller really be concerned if the data that we want to enter into the database is valid? Or, should that be up to the model? You will see that other popular frameworks such as Rails use the Model to validate data. The general rule of thumb you should follow is the skinny controller, fat model. The controller delegates task among the view and the model as much as possible.

## Level 3: Client-side Validation + Server-side (Model)



We don't want our Controller to deal with validations. The controller's job is to delegate all of its work. We can reach the third level of form validations by moving our validations from the controller to the model. **By doing so, we will be able to uphold the general rule of thumb: skinny controller, fat model.**

# Level 1: Client-Side Validation

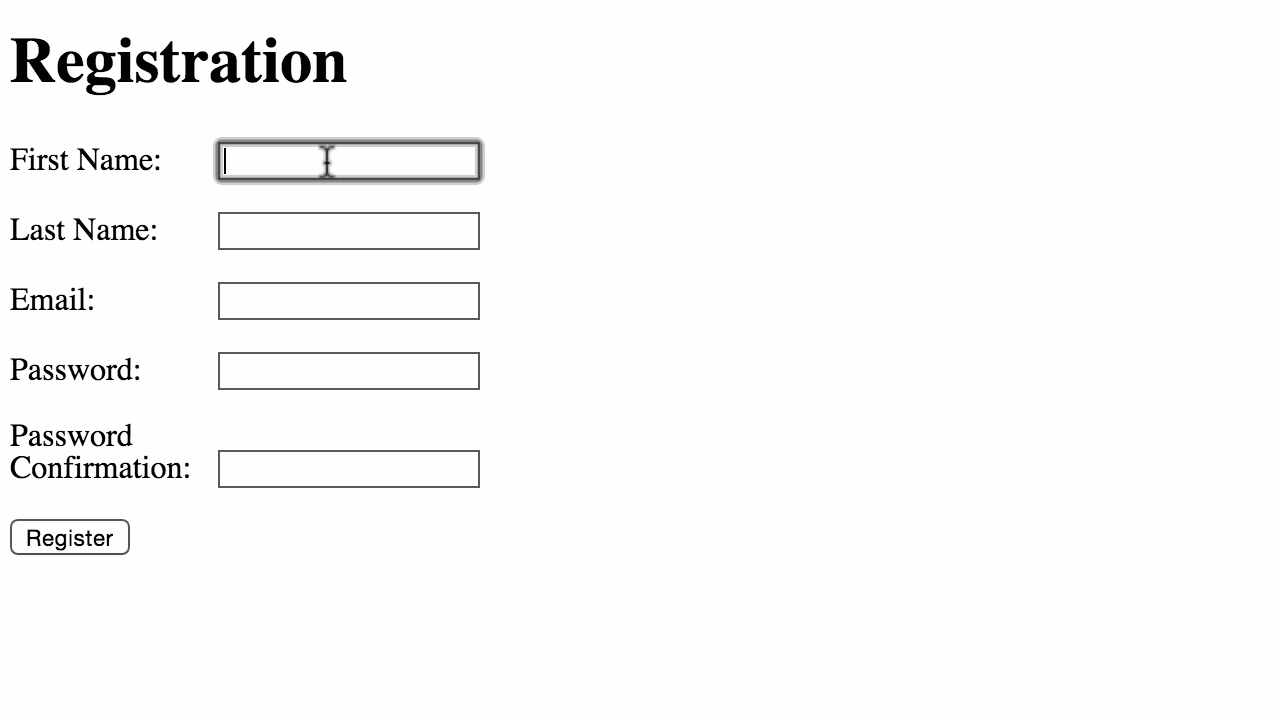
Client-side validations might work something like this. Each input type will listen to 'keyup' event and check every time the user types if what they typed was valid. This will give fast feedback to the users. In fact, this is what we are going to do with jQuery. **We will be cloning a repository on GitHub for us to see the difference between all three levels of form validations.** **Go ahead and navigate to a place in the computer where you want to clone the repository and run the following command:**

git clone https://github.com/andyrewlee/form\_validations.git

There are three different branches in the repository. We are only going to focus on the level\_one branch for now. Go ahead and navigate to that branch by running the following command:

git checkout level\_one

We will go over validations.js in the assets/javascripts directory. You can also point your MAMP to this folder and run the project on localhost to see what we will be going over. You should be seeing something like this:



The file is broken up into two parts. In the first part, we are defining the functions that we will be using. In the second part, we are adding a 'keyup' listener to the inputs that we have on the HTML and executing the functions that we wrote every time the user types into the specific input element.

# Functions

## validateEmail($obj)

This function has one parameter, $obj, which is going to be a jQuery object. We will be using this convention for the rest of the functions. If we are expecting a jQuery object then we will prepend the variables with '$'. This is important because we might be assuming a jQuery object during our implementation of the function and if it is not a jQuery object, our program will crash. JavaScript has a .test() method that can be called (to match patterns in strings) and it will return TRUE if the pattern is found. You can learn more about the .test() method here:  <http://www.w3schools.com/jsref/jsref_regexp_test.asp>

*function* validateEmail($obj) {

var regEx = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)\*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;

return regEx.test($obj.val());

}

## swapClass($obj, beforeClass, afterClass)

This function has three parameters, with the first parameter being a jQuery object, the second parameter being the class you want to remove, and the third parameter being the class you want to add. It is important that we pass in a jQuery object because we are calling methods .removeClass() and .addClass() which are methods that can only be called on jQuery objects.

*function* swapClass($obj, beforeClass, afterClass) {

$obj.removeClass(beforeClass);

$obj.addClass(afterClass);

}

## passwordsMatch()

This function compares two input elements in our view page with one having the id of password\_confirmation and the other being password. We are calling the .val() method on these jQuery objects to compare if the values of these inputs are the same. If the passwords are the same, the function will return TRUE, otherwise, it will return FALSE.

*function* passwordsMatch() {

return $('#password\_confirmation').val() == $('#password').val();

}

## hasPresence()

This function takes a jQuery object and returns TRUE only if the object does not contain an empty string or a null value. The function returns FALSE if the mentioned condition does not pass. We would have to make sure that the argument that we pass into this function is a jQuery input element so that we can call .val() method on it.

*function* hasPresence($obj) {

return $obj.val() !== "" && $obj.val() !== null;

}

## meetsLengthRequirements()

This function takes in a jQuery object and a number that specifies the minimum number of characters. This function will return true only if the value of the jQuery object that was passed in has a length that is greater than the specified minimum length and false otherwise.

*function* meetsLengthRequirements($obj, minLength) {

return $obj.val().length > minLength;

}

## fail($obj)

This function takes in a jQuery object and directly passes it to the swapClass function that we defined. This is going to change the $obj class from 'success' to 'fail'.` Elements with the 'success' class will have text color of lime and elements with the 'fail' class will have a text color of red.

*function* fail($obj) {

swapClass($obj, 'success', 'fail');

}

## success($obj)

This function takes in a jQuery object and directly passes it to the swapClass function that we defined. This is going to change the $obj class from 'fail' to 'success'.

*function* success($obj) {

swapClass($obj, 'fail', 'success');

}

# Listeners

We are going to add a listener to each of the inputs on our form. Every time the 'keyup' event is triggered from the user, we are going to run our functions and give the user instant feedback on whether what they are putting in is valid or not.

$(document).ready(function() {

$('#first\_name, #last\_name').keyup(function(){

hasPresence($(this)) ? success($(this)) : fail($(this));

});

$('#email').keyup(function(){

validateEmail($(this)) ? success($(this)) : fail($(this));

});

$('#password').keyup(function(){

meetsLengthRequirements($(this), 7) ? success($(this)) : fail($(this));

});

$('#password\_confirmation').keyup(function

copy

(){

passwordsMatch() ? success($(this)) : fail($(this));

});

});

# Level 2: Client-side Validation + Server-side (Controller)

A lot of developers validate the form in the controller. In fact, even on CodeIgniter's documentation, they do their validations in the controller. If you do validations in the controller, you are at Level 2. This is a huge improvement from Level 1 because you are now doing validations on the server-side. This is more secure because we are not trusting that the user will not mess with the front-end validations. However, should the controller really be concerned if the data that we want to enter into the database is valid? Or, should that be up to the model? You will see that other popular frameworks such as Rails uses the Model to validate data. The general rule of thumb you should follow is the skinny controller, fat model. The controller delegates task among the view and the model as much as possible. We will learn how to do validations in the model in the next tab.

Go ahead and check out the branch named level\_two. You can do this by running the following command:

git checkout level\_two

Here are some of the changes that we made since level\_one. On top of the validations on the client-side, we validate the post data in the controller. For this application to run you are going to need a database named "*formvalidations*" and one table in there of users with the columns first\_name, last\_name, email, password, created\_at, and updated\_at. Go ahead and create this database and run the CodeIgniter application.

Here are the changes made in the config/database.php:

if(ENVIRONMENT == 'production')

{

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

$db['default']['password'] = '';

$db['default']['database'] = '';

}

else

{

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

$db['default']['password'] = 'root';

$db['default']['database'] = 'formvalidations';

}

Here are the changes made in the application/controllers/users.php:

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

*class* Users *extends CI\_Controller* {

public *function* new\_user() {

$this->load->view('users/new');

}

public *function* create() {

$this->load->model('User');

$this->load->library('form\_validation');

$this->form\_validation->set\_rules('first\_name', 'First Name', 'trim|required');

$this->form\_validation->set\_rules('last\_name', 'Last Name', 'trim|required');

$this->form\_validation->set\_rules('email', 'Email', 'trim|required|valid\_email|is\_unique[users.email]');

$this->form\_validation->set\_rules('password', 'Password', 'trim|required|min\_length[8]|matches[password\_confirmation]');

$this->form\_validation->set\_rules('password\_confirmation', 'Password Confirmation', 'trim|required');

if($this->form\_validation->run()) {

$id = $this->User->create($this->input->post());

$success[] = 'Welcome! Registration was successful!';

$this->session->set\_flashdata('success', $success);

redirect('/users/show/' . $id);

} else {

$errors = array(validation\_errors());

$this->session->set\_flashdata('errors', $errors);

redirect('/');

}

}

public *function* show($id) {

$this->load->model('User');

$data['user'] = $this->User->find($id);

$this->load->view('users/show', $data);

}

}

Here are the changes made in the application/models/user.php:

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

*class* User *extends CI\_Model* {

public *function* create($post) {

$query = "INSERT INTO users (first\_name, last\_name, email, password, created\_at, updated\_at)

VALUES (?,?,?,?,?,?)";

$values = array($post['first\_name'], $post['last\_name'], $post['email'],

md5($post['password']), date("Y-m-d, H:i:s"), date("Y-m-d, H:i:s"));

$id = $this->db->insert\_id($this->db->query($query, $values));

return $id;

}

public *function* find($id) {

return $this->db->query("SELECT \* FROM users WHERE id = ?", array($id))->row\_array();

}

}

And we have also created a partial that we load at the top of the view in applications/views/partials/flash\_messages.php:

<div id='errors'>

<?php

if($this->session->flashdata('errors'))

{

foreach($class="keyword from-rainbow">this->session->flashdata('errors') as $value)

{ ?-rainbow">>

<p-rainbow">><?= $value ?></p>

<?php }

} ?>

</div-rainbow">-rainbow">>

<div id='success'>

<?php

-rainbow">-rainbow">if($this->session->flashdata('success'))

{

-rainbow">foreach($this->session->flashdata('success') as $value)

{ ?>

<p-rainbow">><?= $value ?></p>

<?php

}

} ?>

</div>

This is how we load the partial on top of our view:

<?php

$this->load->view('partials/flash\_messages.php');

?>

# Level 3: Client-side Validation + Server-side (Model)

We don't really have to do anything new to become Level 3. Our website is already secure because we do not only rely on client-side validations to make sure that the data we put into our database is valid. However, right now our Users controller has too many lines of code. The controller is doing too much when it should be delegating its work to the Model and the View. It makes more sense to have the Model deal with the validations because Models are the gatekeepers to the database. It makes sense to have the model be responsible for making sure that the data that is about to go into the database is valid. All we will be doing in this tab is moving our validation code from the controller to the model so that we will be able to uphold the general rule of thumb: **skinny controller and fat model.**Go ahead and check out the level\_three branch of the repository you cloned before by running the following command:

git checkout level\_three

Now, if you run the application on localhost you will realize that nothing has changed. The core functionality of the application is the same; we just moved some of the code to a different part of the application. Here is how the controller looks now:

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

*class* Users *extends CI\_Controller* {

public *function* new\_user() {

$this->load->view('users/new');

}

public *function* create() {

$this->load->model('User');

$result = $this->User->validate($this->input->post());

if($result == "valid") {

$id = $this->User->create($this->input->post());

$success[] = 'Welcome! Registration was successful!';

$this->session->set\_flashdata('success', $success);

redirect('/users/show/' . $id);

} else {

$errors = array(validation\_errors());

$this->session->set\_flashdata('errors', $errors);

redirect('/');

}

}

public *function* show($id) {

$this->load->model('User');

$data['user'] = $this->User->find($id);

$this->load->view('users/show', $data);

}

}

And here is how the Model looks now:

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

*class* User *extends CI\_Model* {

public *function* create($post) {

$query = "INSERT INTO users (first\_name, last\_name, email, password, created\_at, updated\_at)

VALUES (?,?,?,?,?,?)";

$values = array($post['first\_name'], $post['last\_name'], $post['email'],

md5($post['password']), date("Y-m-d, H:i:s"), date("Y-m-d, H:i:s"));

$id = $this->db->insert\_id($this->db->query($query, $values));

return $id;

}

public *function* find($id) {

return $this->db->query("SELECT \* FROM users WHERE id = ?", array($id))->row\_array();

}

public *function* validate($post) {

$this->load->library('form\_validation');

$this->form\_validation->set\_rules('first\_name', 'First Name', 'trim|required');

$this->form\_validation->set\_rules('last\_name', 'Last Name', 'trim|required');

$this->form\_validation->set\_rules('email', 'Email', 'trim|required|valid\_email|is\_unique[users.email]');

$this->form\_validation->set\_rules('password', 'Password', 'trim|required|min\_length[8]|matches[password\_confirmation]');

$this->form\_validation->set\_rules('password\_confirmation', 'Password Confirmation', 'trim|required');

if($this->form\_validation->run()) {

return "valid";

} else {

return array(validation\_errors());

}

}

}

Congratulations! You are now Level 3.

# Fun with Controllers

Let's get to know our Controllers a little better. We are going to go over the difference between redirect and calling a controller method by predicting the output in various situations.

## Example 1: Echoing before Redirecting

Here is our view file

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Fun With Controllers</title>

</head>

<body>

<a href="/welcomes/show">Click Me</a>

</body>

</html>

And here is our controller:

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

*class* Welcomes *extends CI\_Controller* {

public *function* index()

{

$this->load->view('welcomes/index');

}

public *function* show()

{

echo "2";

echo "25";

redirect("http://www.cnn.com");

}

}

What do you think the output is going to be when we click on the 'Click Me' link? It will echo out "225" and throw an error saying "Cannot modify header information." This is because we can't echo out things before we do a redirect.

## Example 2: Redirecting

Let's change our controller to the following. What do you think the output would be once we click the Click Me anchor tag that will execute the show instance method in the Welcomes controller?

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

*class* Welcomes *extends CI\_Controller* {

public *function* index()

{

$this->load->view('welcomes/index');

}

public *function* show()

{

redirect("/welcomes/hello");

}

public *function* hello()

{

echo "hello";

}

}

The show instance method in the Welcomes controller will be executed first. During the execution, we redirect to "/welcomes/hello" which triggers a whole new request. Our request will go to the routes file, which will then determine from the given route and our default controller that the hello instance method in the Welcomes controller should be run. The hello() method gets run which echoes out "hello" and that is the output we will see in the browser. Notice on which route "hello" is being echoed out. **The location is /welcomes/hello because we redirected to that route.**

## Example 3: Calling an instance method

Now let's change our previous code to the following. What would be the output?

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

*class* Welcomes *extends CI\_Controller* {

public *function* index()

{

$this->load->view('welcomes/index');

}

public *function* show()

{

$this->hello();

}

public *function* hello()

{

echo "hello";

}

}

The output would be the same as before, but there is one crucial difference between this example and the previous example. In the previous example, we used redirect which creates a whole new HTTP request. This means that the whole cycle starts over again with it going to index.php then routes to determine which controller is responsible and which method should be run. On the other hand when we are calling an instance method within our instance method, we do not start a whole new request. It just jumps directly to the function and executes it. **This explains why the route that shows "hello" in this case is /welcomes/show**because /welcomes/show was what we requested and we didn't make a new request so this is our final destination.

# Why Certain Routes Should Not Render a View

There are certain controller methods where you shouldn't load the view. For example, let's say a controller is both responsible for creating an order and also displaying the success page. If we have this on the same controller then whenever a user wants to load that page, something is going to get created in the database. If a user were to click refresh because the Internet was being slow, the order will be made twice. This is why controller methods that are responsible for updating the database should not be responsible for loading the view. They should do something with the database and redirect to another route where no database changes are made and only the view is loaded.

**Form inputs**

For an e-commerce website, imagine that you had a form such as this

Buy this awesome T-shirt

<form action="..." method="post">

<input type='hidden' name='description' value='Tshirt' />

    <input type='hidden' name='price' value='19.99' />

<select name="quantity"> ... </select>

<input type='submit' value='Buy Now' />

</form>

If you were a hacker, can you think of how you could hack into this system and buy this T-shirt for 0.01?

One way you could easily hack a site like this is to open up the site, use Inspect Element, double click on the 19.99 field, change it to 0.01, and then to click on the submit button.  As the browser will allow you to modify any html output, you can easily adjust the value in the form, even if the form input's hidden.  When you modify the price and submit, the browser will send a http request with the following formation information:

* description - Tshirt
* price - 0.01

Is that what you wanted?

How could you protect your site from the user easily changing any of the input values?

**Industry insight**

Back in 2013, I remember teaching a group of students about this concept and how we can easily adjust any of the html values easily using Inspect Element.  Then I challenged the students to see if they could find an e-commerce website that had a form structured like this and to see if they really did allow them to buy items for $0.01 or whatever they wanted.  In about 30 minutes, students came back with multiple sites where sites had these security flaws.

Now, if you wanted to take advantage of these sites, you could easily build a crawler that stored all the html response and see if they had any input fields for the price.  Most likely, a lot of these sites would be vulnerable for these user modifications/hacks.

**Solution**

The key for fixing these type of security flaws is to not put any critical information in the form that the users can easily modify or put information in the form that's okay even if the users were to change it.  For example, instead of putting the price, what if you put in the product ID and also the quantity?  For example, maybe the form for buying the T-shirt could be as follows:

<form action="..." method='post'>

<input type='product\_id' value='100313' />

<select name='quantity'> ... </select>

<input type='submit' value='Buy Now' />

</form>

Now, for these forms, even if the user was to change the product\_id manually, it won't matter.  They would simply be buying a different item!

**Key takeaway**

IAs a general rule, it's never a good idea to completely trust the information that comes from the user.  Sanitize the data, put information that's okay for the users to change, and implement other security measures as you'll be taught in a bit.

# XSS Filtering and CSRF

Cross-site scripting (XSS) is a type of security vulnerability found in a lot of web applications.  Spend up to 30 minutes to learn more about this including going through <https://en.wikipedia.org/wiki/Cross-site_scripting>.

Cross-site request forgery (CSRF) is another common security vulnerability found in a lot of web applications. Spend up to 30 minutes to learn more about this including going through <https://en.wikipedia.org/wiki/Cross-site_request_forgery>.

Now, most frameworks have tools to prevent your site from being attacked with CSS and CSRF.  For CodeIgniter, see how this is done by reading: <https://codeigniter.com/userguide3/libraries/security.html>

Going forward, for all CodeIgniter projects, make sure your application is secure from these two common attacks.

**SQL Injection**

SQL injection is a common security vulnerability that unfortunately a lot of websites are vulnerable for.  Please spend up to 30 minutes reading about SQL injection.  A good starting point is <https://en.wikipedia.org/wiki/SQL_injection>.

Imagine you had the following code in your model file:

function ...($post) {

    $this->db->query("SELECT \* FROM users WHERE email = '" . $post['email'])."'";

...

}

or if you were using {},

function ...($post) {

$this->db->query("SELECT \* FROM users WHERE email = '{post['email']}' );

...

}

If your controller was passing all the post information directly to your model above, and where your model was receiving all the post information as a variable $post, your site is vulnerable for SQL injection.

**As a rule of thumb, NEVER join any of the post information or any variable information directly in the query string.** Instead, remember that you were taught to use

$this->db->query("SELECT \* FROM users WHERE email = ?", array($post['email']));

By using ? inside the query string, you're protecting your site from SQL injection.

**Industry insights**

Surprisingly a lot of web applications are vulnerable for SQL injection.  Even if your application was protected from most of the SQL injection, all it takes for someone to hack into your system is one function in your code where the developer forgot to protect themselves from SQL injection and added a variable directly to the query string.

After teaching students about SQL injection, I have had numerous students test some of their favorite sites and found out that the site was vulnerable for SQL injection.  On one instance, they learned they could hack into their school website, which meant they could

* modify any records in the database without the site admin knowing about it (e.g. change their school records including grades)
* drop any database tables they wanted (e.g. drop the whole users table)
* modify the user record to give anyone admin access
* change one's password
* retrieve everyone's password

Luckily, these students didn't have any malicious intent so they didn't do any of the above and instead informed the school's IT about the SQL injection vulnerability the school website was exposed to.

As a developer, you should NEVER make the mistake of trusting the information received from the POST.  Always sanitize the user input to prevent your site from SQL injection as well as XSS.

**SQL Attack**

Imagine that a website had a simple login and in its controller, all the post information was passed to the model such as follows:

function ...($post) {

$this->db->query("SELECT \* FROM users WHERE email = '{$post['email']}');

...

}

For the purpose of this assignment, assume that there was no validation and the query was run for whatever email address the user specified in the login page.  Also imagine that you knew that you were user id = 135.

What would you enter in the email address field to:

1. Drop the entire users table?
2. Update everyone's password to your password (allowing you to login to anyone's account using your password)
3. Update your first\_name to be the email and password of all users who is an admin (assume that there is a field in the users table called is\_admin where it's 0 if the user is not an admin, or 1 if the user is an admin).  This way, when you log out and log back in, instead of saying Welcome [your first\_name], it would say Welcome \_\_\_what ever was stored in first\_name field\_\_\_.
4. Update your user access so that you now have an admin access.
5. Update all the user access of the admin so that they are no longer admin (this way, you are now the only one having admin access and you can threaten the site users).
6. Update your first\_name to now contain all the field names available in the users table (to see if. there are other sensitive information you can hack).
7. Update the first\_name of user id = 1 who is an admin to 'You have been hacked!'.  When this user logs in, instead of saying Welcome [first\_name].  It would say, "Welcome You have been hacked!".
8. Delete all users except yourself

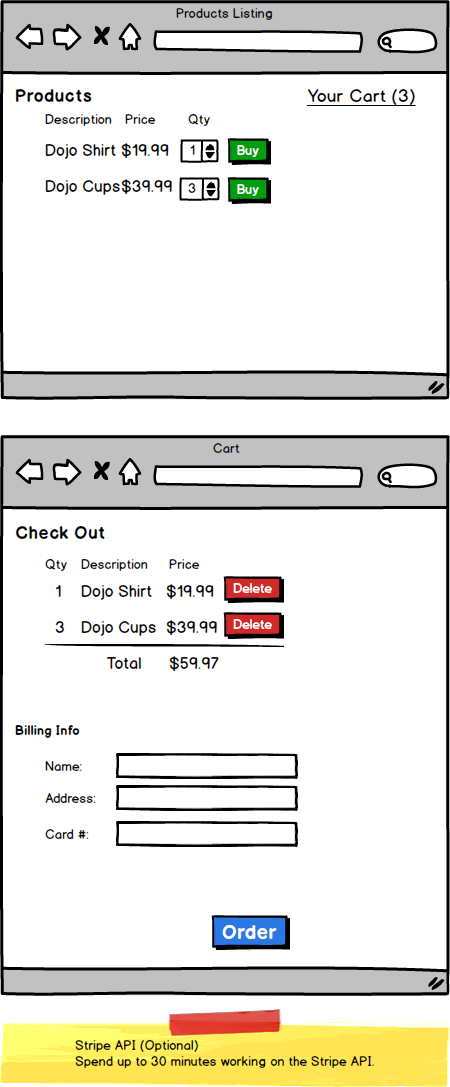
Can you think of other ways you can hack into this database?   Essentially, any code that's vulnerable for SQL injection allows the hacker to run whatever SQL query they want to run!  This could be extremely dangerous!

Create the 'email address' you would type in the web application to perform the tasks above.  Write a simple text file and upload your text file with answers below.

START WORKING ON THIS

## Assignment: E-commerce (revisited)

Remember the e-commerce assignment you did previously?  Revisit the code and make sure your application is secure from XSS attacks, CSRF, as well as from SQL injection.  In addition, make sure that you form does not contain sensitive information that the users can easily change (e.g. price).



START WORKING ON THIS

## Document your code and add meaningful comments

Aside from readable codes, detailed comments on complex codes are useful and important for those jumping into this project. Any functions without any comment above it, should not be accepted as valid code.

For each function, write a brief description as well as comments that would help other developers as well as yourself, when you're reviewing the code later. For complex function, add explanation on how the functions work, when it is called, and other important details that other devs should know.

### Comment Format

Use /\* instead of // for comments since the latter will have issues on code precompile.

Good

/\* DOCU: This function is triggered by Run and Save and Submit and Watch buttons in the Challenge Page. For run and save, it triggers checking of code via isolate and also sends the recorded keypresses of the user to S3. For Submit, this updates time spent details of the for the challenge.

Requires hero\_id, challenge\_id

Owner: PJ, Updated by Chris

\*/

function completeChallenge($post\_data){

...

Bad

//HERO SUBMIT CHALLENGE ANSWER

function completeChallenge($post\_data){

...

## General Rule of Thumb

Each method/function in the model and the controller should have comments documenting what the function is doing.  It should also layout what the function will be returning (its output) as well as what parameters are expected for the function to run (its arguments).

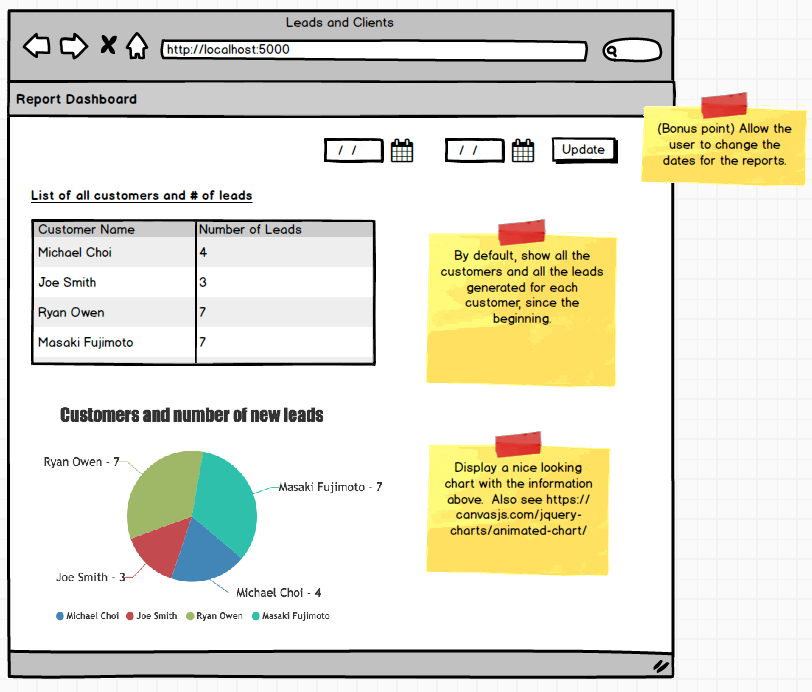
In addition, as a general rule of thumb, if for every 15-20 lines of code, if your code does NOT have any comments, it probably means you're not documenting your code enough.

# Leads and Clients

## Objectives:

* At a deeper level, understand how the server generates a http response which the browser can use to render the html/css or to give to the Javascript interpreter.
* Have you get exposure to generating beautiful graphs/plots using jQuery and know how to insert information from the database to Javascript.

In the [Lead Gen Business assignment](http://learn.village88.com/m/34/278/2856) from Web Fundamentals/MySQL, you wrote some complex queries for leads, sites, and clients. In this assignment, you'll use some of those queries to create a simple report dashboard:



The main goal of this assignment is to see how you can get data from your server and use it in script tags on the template. Start by pulling all the appropriate information from the database to generate the table and the graph.

If you have time, allow the user to change the reporting date range by adjusting the two dates on the top right. This assignment can be a bit difficult but teaches a lot of great lessons that could save you lots of time later when you're working on real complex projects.

#### Tips

* In the wireframe/assignment, customers and clients are used interchangeably.
* Test your queries in MySQL Workbench/MySQL command line to ensure you don't have any syntax issues and that your query is doing what you think it's doing.
* Use inspect element to view the data being rendered in your script tags.
* Don't forget that "" indicate literal strings, while words without quotes are read as variables.
* Play around with the code directly at <https://canvasjs.com/jquery-charts/animated-chart/> to make sure you understand what it's doing and get a sense for which parts you'll need to update.
* Make sure you add appropriate comments to make the code readable.

#### Video Overview

For this video, a tool called Jinja is mentioned.  This tool is used for printing javascript variables into the html and when you are using Node.js for the back-end language.  For example,

<h1>{{ ... }}</h1>

In PHP, Jinja is not used and as you know, we add PHP in the HTML by following this convention:

<h1><?= ... ?></h1>

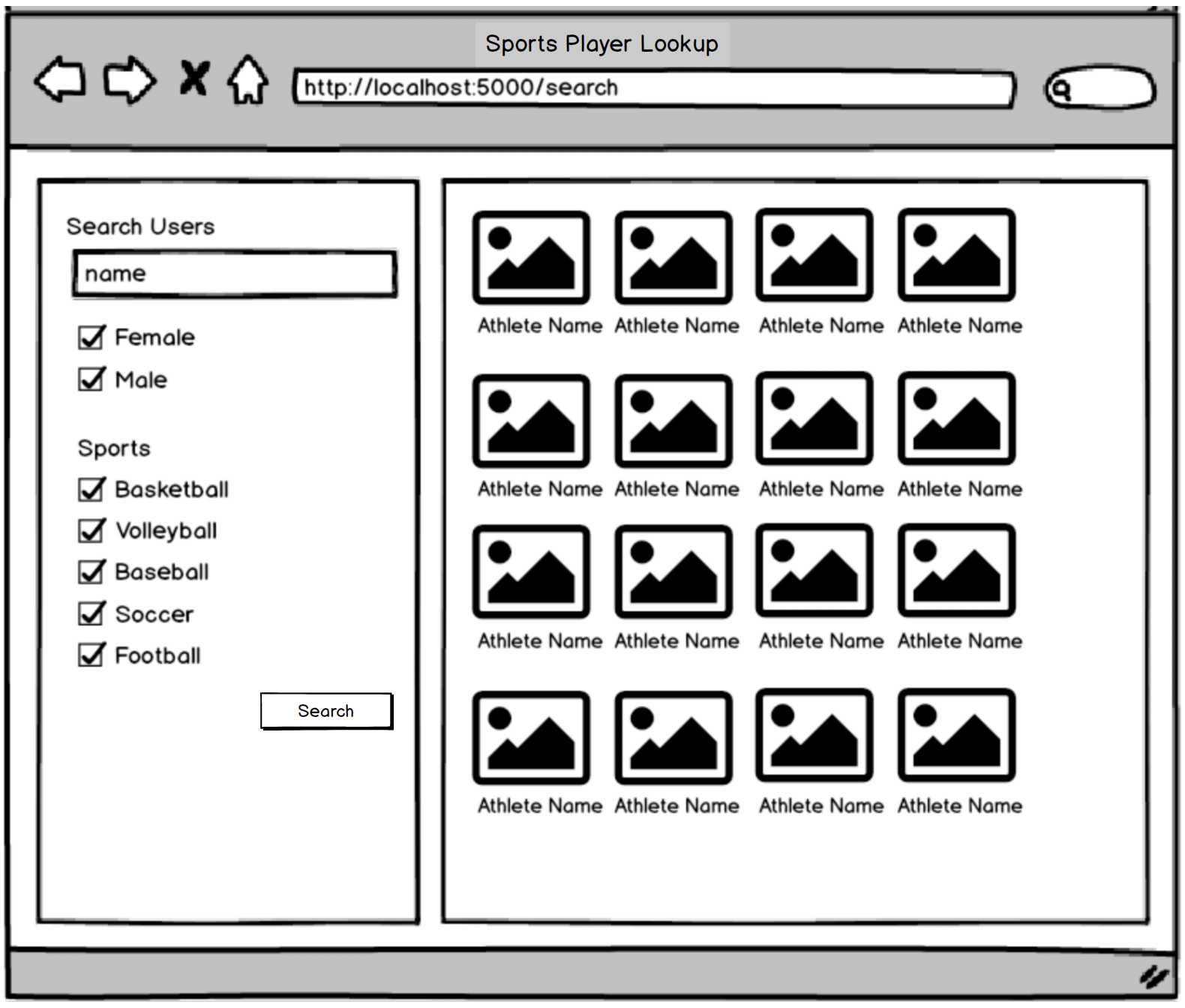
START WORKING ON THIS

# Sports Players

Build an application that allows the user to search through a database of sports players where they can filter the results by the name, gender, or which sports they are involved with.

Create your own ERD and pre-populate the database with some fake players information.  No need to store the images of the players in your database.  Instead, simply store the url of a remote image (an image hosted in another site) in your database (e.g. ).

Once you've populated the database with dozens of players information, make this application work.



As you do this, try to make your model as concise as possible.  In addition, document your code well so that it's readable.

**Debugger**

VSCode has great debuggers.  Instead of using var\_dump and echo statements throughout your application, using a debugger can save you enormous amount of time, often making debugging 3-10 times faster!  This is especially true for debugging a large complex application.

Learn about setting up a debugger for PHP in Visual Studio Code.  A good starting point is <https://code.visualstudio.com/docs/languages/php>

Once you set this up, make sure you're familiar with

* how to set breakpoints
* how step over / step into works
* how to use the 'Watch' feature to watch how variables change as you go through each line.

Once you use the VSCode Debugger, you'll never have to use echo or var\_dump.

Once you're familiar with this tool, see if you can use the debugger to trace how the HTTP request travels from your routing file -> controller -> model -> controller -> view.

In order to become a great developer, you must know get really familiar with these debugging tools.

# The Wall Bug

In this assignment, the objectives are to practice you from using Debugger tool and have you familiar on errors and solutions that possibly you may encounter in the future.

**Instruction**: Investigate the lines that are root cause of bugs specified in the list and correct it to make the whole web app functional. You can get the source code from this link: <https://github.com/hh-kigcasan/the_wall_bug>

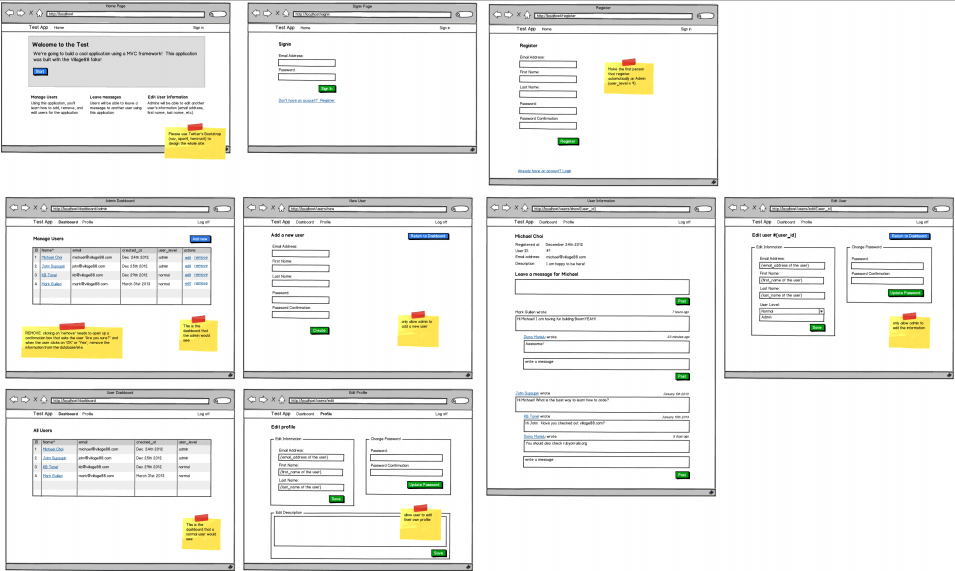
These are the possible error causes:

1. You may be using a function from a helper that is not loaded yet. (Ex. 'url', 'security')  
2. You may be using a function from a model that is not loaded yet. (Ex. 'comment', 'message')  
3. Encountering 403 (Forbidden) where form request is not allowed by the enabled CSRF Protection due to form's missing token.   
4. Calling a function but forgot to pass the required parameter/s.   
5. Missing form validation. (Ex. Input field 'email' should be valid, not null).  
6. Being redirected to a missing page which caused by wrong route path. (You may check the routes.php)  
7. You might be comparing values with different format. (Ex. Comparing a normal character string over the encrypted string). Check the model and see if you need to convert the string.  
8. You might initialized a variable with incorrect, hardcoded value. (Ex. hidden input field is initialized to a normal string instead of an id).  
9. The controller/model can't find the specific data from submitted form due to misspelled name in one of the form's input field.  
   
Tips:- Use breakpoints to highlight the lines you want to investigate and see if the program will go to it.- Use 'Watch' feature in the Debugger tool to monitor the value of variable/s.- Use 'Step Into' to dig through the called function. - Use 'Step Over' to go to next lines.

Make sure you read first the README.md file for more instructions. Finish this assignment no more than 2 hours and once you're done, kindly attach your fixed codes here. Good luck!

## Assignments: User Dashboard

**Note: Upload your work in GitHub and allow your collaborators to add/suggest new features for your work.**

Watch this [video](http://www.youtube.com/watch?v=90FPKvP_FiA) for details. Work on the pages indicated in the wireframes and make sure to follow same URL/links structure. For the full PDF, download the handout on the right or click [here](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3063/handouts/chapter3063_4024_user-dashboard.pdf).  Make sure you zoom into the PDF to see a larger view of each page.

This is one of the best assignments you can do to prepare for the qualifying exam as well as to improve your coding skills.  Many of our students have put this as part of their portfolio and have expressed how this assignment really helped them to understand the fundamentals of web programming.  We encourage all of you to finish this assignment if you have the time. Also, make sure you're using Git to back up your codes along the way.

GitHub is a social network hub for programmers. You can look at what other programmers are working on, check their code and make suggestions or changes. You may share projects you are working on and collaborate with other programmers for updates or additional features.

Upload your work for this exercise on GitHub and ask some of your cohort mates or other programmers to peruse your code and suggest changes/updates. Do the same thing for your cohort mates' work.

START WORKING ON THIS

**CODEIGNITER + ORM**

# Object Relational Mapper (ORM)

The Datamapper ORM was intended for CodeIgniter 2 use.  Seeing as how we are using CodeIgniter 3 in this curriculum, all of the ORM information is still relevant for general knowledge but only applicable to [CodeIgniter 2](https://github.com/bcit-ci/CodeIgniter/archive/2.2.6.zip) which you can download using the link.  The **Add Products** and **Product Listing**assignments are also optional.

Using an ORM is another way to retrieve information from the database similar to the Active Record Class using in Ruby on Rails. CodeIgniter has an ORM library called DataMapper which could make your life easier or more complicated (depending on what you want to do).

If you want to get into Ruby on Rails later, it would help you to get familiar with an ORM at this point. Otherwise, you may skip this particular section as you can still learn everything you need about Active Record Class in the Ruby on Rails course.

## Benefits

The primary benefit in our opinion is that if you have a lot of database tables that are related to each other (e.g. courses, chapters, modules, students, etc), when you end up pulling data from the database, an ORM can do this without having to write long queries with lots of join statements. In fact, you can do a lot of data retrieval without writing a single query!

In addition, for a simple database row insert or update such as adding a new row in the database: when a new user registers, their registration information is updated, a new activity is created, etc, you could use the ORM to call a simple function save() to do all this without writing a single query. If done properly, an ORM could save you lots of lines of code but it also can be very dangerous (read below for more info).

## When Not to Use an ORM

Sometimes it's better just to do things the way we have been with CodeIgniter models especially if you're doing complex queries.  It's just not worth it in our opinion to figure out a way to make something work with an ORM and spend hours when you can do it using what we already know in a few minutes.  Our preference is for you to do something very basic with the Datamapper (such as retrieving basic sets of data) and whenever you're doing some complex queries where you're grouping results or doing complex joins, to use your regular queries you've been using throughout the course.

## Danger of Using an ORM

Basically, it's really easy to write a lot of redundant queries, which slows your application and potentially creates an unnecessary load for the database.  How do we avoid this?  Always have enable\_profiler() on and check all the queries being run. See if there are unnecessary queries that are being repeated.  If there are, figure out a better way to do it using ORM or if you can't figure this out, just use your own manual queries.  This can be easier said than done, though.

Another danger of using Datamapper is that we believe it's hard to maintain a clear MVC structure as you have Datamapper objects in the views (in a true MVC framework all queries should be done through the "models" and not in the "views") and those without good backend knowledge would use Datamapper in the wrong way to generate lots of redundant queries.  We have seen numerous examples where someone was using Datamapper to retrieve data through 100+ queries when it could have been done using a single query to retrieve the data at least 100 times faster.  So again, if you do decide to use an ORM, make sure even those that are doing front end development know how to use the ORM and to have enable\_profiler activated to always check for any redundant queries.

## Installation

Step 1. Download the ORM code from this link: <https://github.com/WanWizard/sparks-datamapper>  
Step 2. Setup: <http://datamapper.wanwizard.eu/pages/installation.html> (follow the 'Long Version' of installation instructions, all 11 steps are required)

**NOTE: For MAC users and even Windows users, don't delete or remove CI files, just add the ORM Files to their respective folders. ORM Files will work with your CI Files. Don't remove any CI file.**

**REMINDER:** Load the Datamapper bootstrap file first in index.php before loading CI Bootstrap file at the very bottom of the file.

require\_once APPPATH.'third\_party/datamapper/bootstrap.php';

require\_once BASEPATH.'core/CodeIgniter.php';

## Models

Read more about ORM models: <http://datamapper.wanwizard.eu/pages/models.html>

## Relationships

Read more about relationships: <http://datamapper.wanwizard.eu/pages/settingrelations.html>

Read more about joins: <http://datamapper.wanwizard.eu/pages/getadvanced.html#Get.Advanced.Selection>

## Validations

Read more about validations: <http://datamapper.wanwizard.eu/pages/validation.html>

## References

**The following pages are helpful although you'll learn much more by using these functions to build something.**

1. <http://datamapper.wanwizard.eu/pages/gettingstarted.html> - very important introduction
2. <http://datamapper.wanwizard.eu/pages/get.html> - quickly read through once and only get yourself very familiar with the list of top Datamapper functions listed below.
3. <http://datamapper.wanwizard.eu/pages/getadvanced.html> - quickly read through once and only get yourself very familiar with the list of top Datamapper functions listed below.
4. <http://datamapper.wanwizard.eu/pages/save.html>
5. <http://datamapper.wanwizard.eu/pages/delete.html>
6. <http://datamapper.wanwizard.eu/pages/timestamp.html>

#### Top Datamapper functions we've used (only focus on learning these first):

1. get()
2. get\_by\_id()
3. get\_iterated()
4. all\_to\_array()
5. save()
6. result\_count()
7. count()
8. include\_related()
9. update()
10. delete()
11. delete\_all()
12. skip\_validation()
13. select()- similar to active record
14. where()- similar to active record
15. like()- similar to active record
16. group\_by() - similar to active record
17. order\_by() - similar to active record
18. limit()- similar to active record

This is easily one of those topics where you can spend days of research without making much progress. Our recommendation is for you to first master the above functions and forget about the other more advanced functions until you're feeling much more comfortable with the ORM.

**Assignment: Add Products**

The goal of this assignment is to introduce the ORM basic structure and on how to display/add/delete information to the database.

Create a single page application that displays the list of products saved on the database. This page also allows you to Add a new product and delete existing ones.

Make sure you've downloaded [CodeIgniter 2](https://github.com/bcit-ci/CodeIgniter/archive/2.2.6.zip) and [DataMapper](https://github.com/WanWizard/sparks-datamapper" \t "_blank) and ran through DataMapper's [long version of installation instructions](http://datamapper.wanwizard.eu/pages/installation.html).

Make these changes to your project's .htaccess file:

RewriteEngine On

# !IMPORTANT! Set your RewriteBase here and don't forget trailing and leading

# slashes.

# If your page resides at

# http://www.example.com/mypage/...

# then use

# RewriteBase /mypage/test1/

RewriteBase /

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

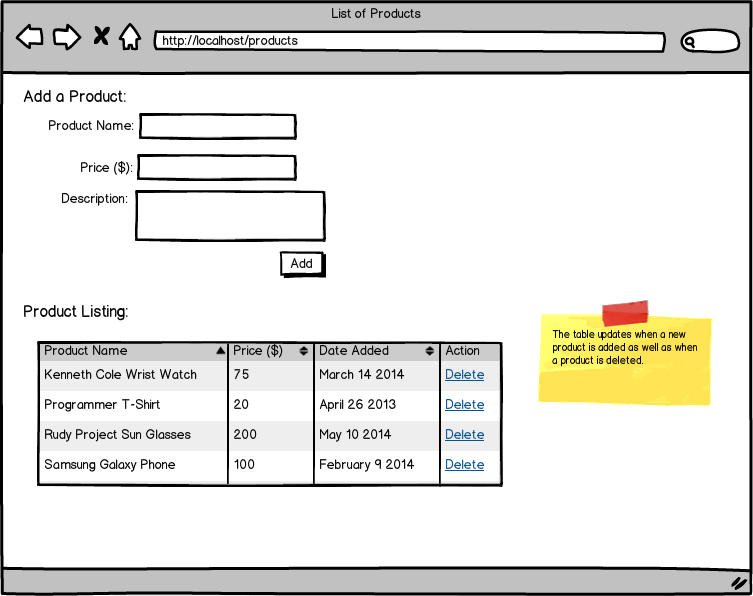
RewriteRule ^(.\*)$ index.php?/$1 [L]

# If we don't have mod\_rewrite installed, all 404's

# can be sent to index.php, and everything works as normal.

# Submitted by: ElliotHaughin

ErrorDocument 404 /index.php



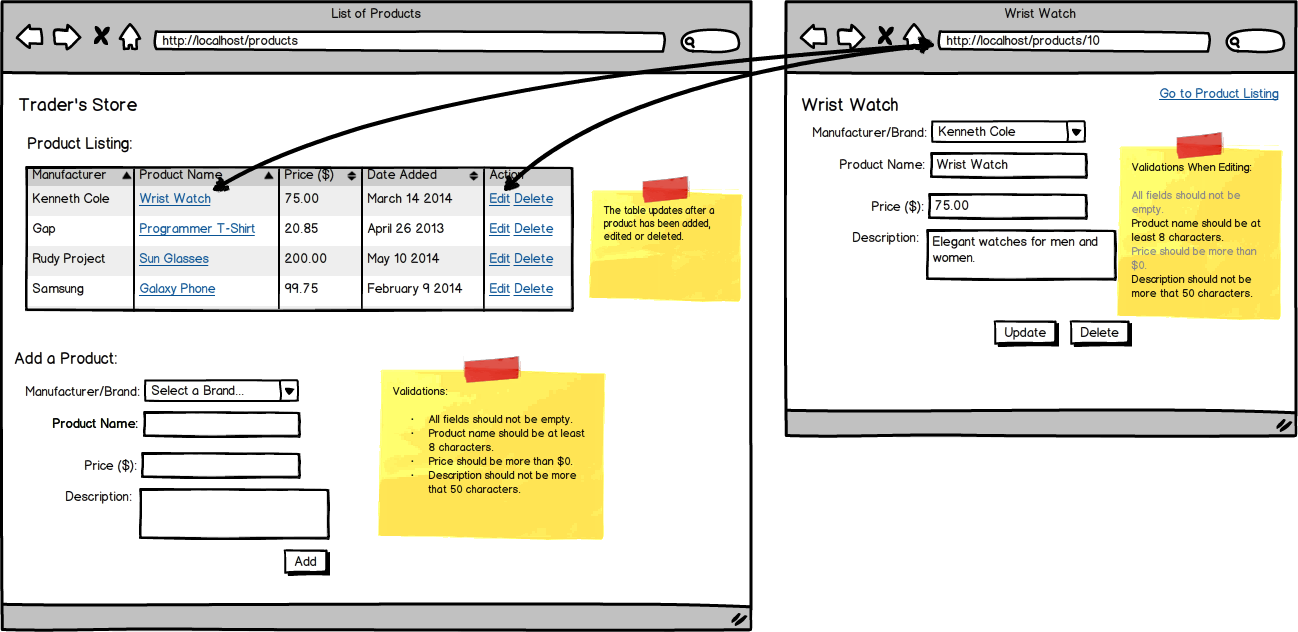
**Requirements:**

* Create a Product Listing page where you can display/add/delete products.
* Try to add new products without manually inserting the dates to the database. ORM will automatically generate the created\_at field which you can use in displaying the “Date Added” field in the table above.

START WORKING ON THIS

**Assignment: Product Listing**

The goal of this assignment is to help you practice basic CRUD using ORM. This exercise also includes ORM validations and DB table relationships.



**Requirements:**

* Create two tables on your database (products and manufacturers). You may add initial records for the manufacturers table.
* Make sure you add ORM validations when adding/editing a product.
* (Optional) Try to add a Manufacturers Page where it displays the manufacturer and the products (table format) of that specific manufacturer.