Glenn Stovall

# Building Maintainable Web Applications With CodeIgniter

**Glenn Stovall** glennstovall@gmail.com glennstovall.com

*Controlling Complexity is the essence of Computer Programming.*
*— Brian Kernighan*[1]

*Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges and it causes end-user and administrator frustration.*
*—Ray Ozzie*[2]

*Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius — and a lot of courage — to move in the opposite direction.*
*—Albert Einstein*

[1] Software Tools (1976), p. 319 (with P. J. Plauger)

[2] Former CTO of Microsoft, creator of Lotus Notes.

# Overview

Software is complicated by default. It should be the goal of software developer to make sure that the applications they are writing stay as simple as possible. By using a framework like CodeIgniter, you are already taking a step in the right direction. CodeIgniter provides a general structure to help organize your application. This document will help have a better understanding of how CodeIgniter is structured, and how you can use CodeIgniter to keep your project organized and maintainable. This document will also cover some general software principles that you can apply to any development project, and help you become a better developer that writes better code.

## Managing Change

The only thing that does not change in software development is that things never stop changing. As software is developed, business requirements can change, the specifications can change, and the code base must change to accommodate this. Even after the initial release of software, this continues to be true. Changing software is not always easy, and costs time and money.  The goal of this document is to try and reduce the cost of this change as much as possible. Have you ever had any of the following happen to you?

- You need to fix what seems like a small bug in an application, however due to the way the code is written, what seemed like a simple fix ends up taking you hours, if not days?
- You want to add a new feature, however adding this new feature requires so many changes to existing code that it isn't worth it?
- You add a new feature to a product, and it ends up breaking seemingly unrelated code in weird and unexpected ways?

# Principles of Software Design

If you take one thing away from document, let it be this: Making your code base more simple is the ultimate goal of software design. If you ever having trouble with a design decision, always favor the simpler choice.  Here are five principles you can apply while writing software that can help keep your code base clean.

## Single Responsibility Principle

The **Single Responsibility Principle** simply states that "a piece of code should only do one thing"[3]. Another way to think of it is that "a piece of code should only have one reason to change".  Your code will be easier to test and maintain if you have several smaller functions and objects that perform individual actions. As Ryan Singer once said, "*So much complexity in software comes from trying to make one thing do two things*"[4].

## Separation of Concerns

The principle of **Separation of Concerns** is for separating out parts of a software application into distinct sections. CodeIgniter follows The Model-View-Controller paradigm, commonly called "MVC". The name refers to three distinct parts of the software system, each of which deals with a separate concern.

- **Models** are designed to handle working with data. Most of the time, this will be interacting with a database on your server. Your models will be responsible for handling all of the common actions you perform on data, such as creating, reading, updating, and deleting records. This is commonly called CRUD. Models are the *only* place you should be writing or even thinking about database queries.
- **Views** are responsible for handling how you present information to the end user. Views should contain all of the HTML in your application, and as little PHP code as possible.
- **Controllers** manage requests from the user. Controllers should be the simplest part of your application. Controllers are simply the messenger in the application. They receive a request from the user, usually in the form of a URL, and then form the response to send back. If you aren't sure if something belongs in a controller or not, changes are that it doesn't.

[3] http://en.wikipedia.org/wiki/Single_responsibility_principle

[4] http://37signals.com/svn/posts/2316-so-much-complexity-in-software-comes-from

## Don’t Repeat Yourself (DRY)

The principle of **Don’t Repeat Yourself** (commonly called DRY), means to reduce duplication in your code base as much as possible. If you ever find yourself duplicating code, it is usually a sign that one of these other principles are being broken. Since PHP is object-oriented, you can use Inheritance to extend classes, and put code that is shared amongst several classes into classes that each of them can extend. DRY can also refer to you repeating code that is already part of the application. CodeIgniter contains several helpful libraries and helpers. It is better to use these to solve your problems instead of writing new functions that duplicate the functionality CodeIgniter already has.

## Interface Segregation

The biggest strength of object-oriented programming is that it allows for **Interface Segregation.** This is also commonly referred to as ‘Separating the interface from the implementation’. Think about when you order food at a restaurant. Let’s say you go to your favorite Italian place and order a plate of the chicken caprese. After a brief wait, you receive your food. You do not know how the food was made. Is there a chef? two? ten? Where did the ingredients come from? If this is one of the cheaper places in town, maybe you don’t *want* to know. What’s important though is that you don’t have to worry about it, as the restaurant, the server, and the employees take care of all that for you.

In computer programming, this hiding of information is called **encapsulation.** It works in a similar way with your users, They don’t know how the web application works. They don’t know what language it’s written in, or what database schema you use. When it comes to writing code, you apply the same principle to methods, functions, and classes. If I call `place_order(‘chicken caprese’)`, It does not matter *how* the function does what it does, even if it changes over time. As long as this function returns a chicken caprese object, I’m happy. A good exercise to practice this is to try writing just the method names of a class with arguments before you write any of the implementation code. This way you are designing your class based on how it is going to be used, instead of having to figure out how to use it based on how it is implemented. This can make for much more readable and reusable code.

## Low Coupling, High Cohesion

**Coupling** refers to how much different objects rely on each other. When writing objects, they should strive to be able to work as independently as possible. For example, each of your controllers should not any references to any other controllers. Then you could say that your controllers have **Low Coupling** among themselves.

**Cohesion** refers to how similar various parts of the code base are. functions with high cohesion should be kept together in objects. For example, if you were writing an application that dealt with managing construction equipment, you might want a page that lists of your equipment, and each piece of equipment might have it’s own individual page. Since these are both related to equipment, and perform similar functions, you would say that they have **High Cohesion**. It would make more sense to have one controller that can handle both of these requests than to have two distinct controllers.

Coupling and cohesion are some of the harder concepts to understand as a software developer. If you have a hard time decided when things should be decoupled, and when they should be grouped together, don’t worry. It’s often subjective, and it takes time and experience to learn how to handle these cases.

# Structure of a CodeIgniter App

Inside of the application folder of your CodeIgniter app, there are several directories for each of the different types of files you will be writing in your application. Understanding what each of the different types are can help you understand where you should place the code you write.

## Models

Models will handle the data of your application. This will usually be interacting with a database of some sort. all models contain an object called '`db`', which can be used to write queries.[5] Models can also be used to interact with other sources of data, such as files, sessions, cookies, and 3rd party web services.

## Views

Views are mostly files that are mostly written in HTML. These will handle the look and output of your application. PHP code in a view should be limited to variables, single function calls, conditionals, and loops.

## Controllers

Controllers will handle all of your URL requests. Controllers also act as the 'glue' of your application. Controllers are responsible for loading up models, views, helpers, and libraries. It contains a `load` object that helps with this. If two different components of your application need to interact, then it should probably be taking place in a controller.

## Helpers

Helpers are files that just contain functions. Helper files are mostly used to abstract out complexity in a view, so that you can keep your view files as free of PHP as you can. CodeIgniter comes with several helpers as part of the core application.

## Libraries

Libraries  are objects in your application that aren't models, and aren't controllers. Libraries may also be something you are using from a 3rd party. They can be a helpful way to share code amongst controllers, and keep your controllers simple. Much like helpers, CodeIgniter has several useful libraries already built in.

## Config

The config directory is useful for handling various set up options inside of your application. There are several config files included by default in CodeIgniter which you can use to declare how parts of your application will work, and store data such as database usernames and passwords. You can also create your own custom config files if you need some of your own set up for your application.

[5] This is called the 'Active Record Class'. You can see of the functions available here: http://ellislab.com/codeigniter/user-guide/database/active_record.html

Company

## Core

The core folder allows you to set up parent classes for models and controllers. Core objects can be very powerful, and allow to easily share code between multiple models and controllers.

## Hooks

Hooks are files that allow you to modify the inner workings of the core CodeIgniter framework, without having to actually modify those files. These are not used as often as some of the other components, but they can come in handy.

## Third_party

CodeIgniter 2.0 added support for packages, which can be placed here. packages are structured just like a standard CodeIgniter application folder, and can include models, libraries, and helpers or its own.

# Creating Core Classes

When starting a new CodeIgniter application is to set up a core model and core controller. By doing this, we will have a place to put common functionality between models and controllers.

## Setting the Prefix

In CodeIgniter, all of it's core classes have a prefix of `CI_`. This tells whoever is working on the application that those classes are part of the CodeIgniter framework, not your application. We will want our own prefix as well. By default, and for these examples, the default prefix is `MY_`. You can change the prefix by editing the `application/config/config.php` file, and changing the `$config['subclass_prefix']` variable.

## Creating Your Classes

You can create the two files `MY_Model.php` and `MY_Controller.php` inside of the `application/core` directory. Once you have done this, just change how your controllers and models are declared, so that they extend `MY_Model` and `MY_Controller` instead of `CI_Model` and `CI_Controller`, respectively.

## MY_Model Example

Let's say we are working on a project management software application, and we have three types of data: users, projects, and tasks. We would have a model for each of these. We would also likely want to be able to fetch any one of these by it's id, so our three classes may look something like this:

```
class User_Model extends MY_Model {
  public function get_by_id($id) {
    $results = $this->db->select('*')->from('users')->where(‘id’,$id);
    return $results[0];
  }
}

class Project_Model extends MY_Model {
  public function get_by_id($id) {
    $results = $this->db->select('*')->from('projects')->where(‘id’,$id);
    return $results[0];
  }
}

class Task_Model extends MY_Model {
  public function get_by_id($id) {
    $results = $this->db->select('*')->from('tasks')->where(‘id’,$id);
    return $results[0];
  }
}
```

You can see that there is some duplicate code in these three classes. The only difference is that each of these refer to a different table. We can set a variable in each of these to refer to the table, and then write a more generic function in our `MY_Model` class:

```
class MY_Model extends CI_Model {
  protected $table = ‘’;

  public function get_by_id($id) {
    $results = $this->db->select(‘*’)->from($this->table)->where(‘id’,$id);
    return $results[0];
  }
}
//In our other files..
class User_Model extends MY_Model {
  protected $table = ‘users’
}
//Etc.
```

Now we can have this functionality in all of our models, and new models we write will have this functionality already built in.  This allows us to write less code, and it also makes our code easier to maintain. There is a bug in the above examples. Did you spot it? The issue is that if we give any of models an id for a class that does not exist, this will cause an error since `$results[0]` will not exist. In the first example, we would have had to make three changes to our code base. Now, we can only make one. Again, this fix not only solves the bugs in existing classes, but it will prevent the bug from appearing in future classes we add.

```
class MY_Model extends CI_Model {
  protected $table = ‘’;

  public function get_by_id($id) {
    $results = $this->db->select(‘*’)->from($this->table)->where(‘id’,$id);
    if(empty($results)) {
      return null;
    }
    return $results[0];
  }
}
```

# Using Libraries

Libraries are a great way to break pieces of functionality down into more digestible, reusable chunks.  Libraries are classes that can be called inside of controllers by using the `load` object once you have loaded a library, you can use it the same way you would a model or any other class, as a property of the current controller:

```
$this->load->library('form_validation');
$this->form_validation->run();
```

Libraries are very useful in applications that have a lot of complex business logic. It shouldn't go into the views, since they are only for display. They also shouldn't go into model classes, since we want them to only have a single responsibility, and that is connecting to our database sources. We also would not want this code to placed into controllers, since they already have a single responsibility, and controllers should be the simplest classes of all. Libraries give us another option for a place to put this kind of code.

## Built-in Libraries

CodeIgniter comes with several built in libraries you can use. [6] Here are five of the most useful ones, in my experience.

- **Email:** http://ellislab.com/codeigniter/user-guide/libraries/email.html
- **File Uploads:** http://ellislab.com/codeigniter/user-guide/libraries/file_uploading.html
- **Form Validation:** http://ellislab.com/codeigniter/user-guide/libraries/form_validation.html
- **Image Manipulation**: http://ellislab.com/codeigniter/user-guide/libraries/image_lib.html
- **Sessions:** http://ellislab.com/codeigniter/user-guide/libraries/sessions.html

## Extending Built-in Libraries

CodeIgniter Libraries can be extended if you want to add your own functionality. This is done in a similar fashion to extending models and controllers. If you want to extend the functionality of existing libraries, go to your `application/libraries` folder and create a new file with your custom prefix, and after that named the same as the existing library. For example, if we wanted to extend the form validation library, so that we can write our own validation rules, we could do so like this:

---

[6] A complete list can be found in the CodeIgniter user guide: http://ellislab.com/codeigniter/user-guide/toc.html

```
class MY_Form_Validation extends CI_Form_Validation {
  //Place your custom functions here
}
```

CodeIgniter's load library is smart enough to look for these. So in this example, if you call `$this->load->library('form_validation')`, CodeIgniter will now load your extended class instead of the native one.

## Creating your own Libraries

Creating your own libraries are simple. Simply create a file in your `application/libraries` folder, then create a class with the same name as the file. That's all there is to it! Now you can load those libraries using the `$this->load->library()` function, of by calling them in the `application/config/autoload.php` file.

## Using Third Party Libraries

Whenever you can, you should design libraries to be agnostic to the rest of your application. What that means is that each library is written as a class that can stand on it's own (loosely coupled). They should also have as little knowledge about the other classes in your application (encapsulation). When libraries are written in this manner, it makes them much more reusable, as you can place them into any application and they will still work.

Several other people have written add-ons for CodeIgniter in this fashion. These can be added to the third_party directory in your application. Third party packages are structured like an application, and can include not just libraries, but helpers and models as well.  There are several sources for finding modules:

- **Sparks[7]:** sparks was an attempt to create a package manager for CodeIgniter. It is no longer maintained, but you can still find some useful modules there.
- **Composer[8]:** Composer is one of the best new tools that has been developed for PHP developers. Composer is a package management system that makes it easy to find and install packages not just for CodeIgniter, but for any PHP application. There are several CodeIgniter packages available[9].

If you building generic functionality into your application, it's a good idea to look and see if there are any existing packages that suit your needs. This can save you a lot of time writing and testing code, and allow you to focus more clearly on the problems your application is trying to solve, instead of writing the internet's 600th Authentication protocol[10].

[7] http://getsparks.org/

[8] http://getcomposer.org/

[9] More information on using composer + CI: http://philsturgeon.co.uk/blog/2012/05/composer-with-codeigniter

[10] If you do need Authentication, I highly recommend Tank Auth: http://konyukhov.com/soft/tank_auth/

# Using Helpers

While libraries are used when you want to add classes to your CodeIgniter application, helpers are used when you want to add functions to it.  There are two main use cases for when you would want to use helpers. The first is for generic functions that you would want to use in several places in your application. The second is for simplifying views. If you find yourself writing complex logic in views, then chances are you should be placing that logic inside of a helper function, and calling that function from your view. Helpers are loaded in a similar way to libraries:

```
$this->load->helper('email');
valid_email('email@example.com');
```

## Built-in Helpers

Like libraries, CodeIgniter comes with several useful helpers built in[11]. Some of the more useful ones:

- **Email:** http://ellislab.com/codeigniter/user-guide/helpers/email_helper.html
- **File:** http://ellislab.com/codeigniter/user-guide/helpers/file_helper.html
- **Form:** http://ellislab.com/codeigniter/user-guide/helpers/form_helper.html
- **URL:** http://ellislab.com/codeigniter/user-guide/helpers/url_helper.html

## Writing your own helpers

Creating your own helpers is just like creating libraries; Just create a file in the `application/helpers` directory, and then you can load them in your controllers using the `$this->load->helper`() function, or in the `application/config/autoload.php` file. Even if you using a helper exclusively in a view, The controller is still responsible for loading the helper.

## Extending Existing Helpers

Even though helper files are just files, they can still be extended in a similar way to libraries. By creating a helper file with the same name as a native helper, but with your custom prefix (i.e. `MY_email_helper.php`), then CodeIgniter will include these functions as well as the native ones when that helper is loaded.

[11] A complete list can be found here: http://ellislab.com/codeigniter/user-guide/general/helpers.html

# Managing Assets

Asset files refer to any files you plan on serving directly to the client. Assets typically include CSS stylesheets, JavaScript files, fonts, and images.  They could also include files that you intend to offer for download, such as a PDF.  One of the shortcomings of CodeIgniter is that it does not offer any way of handling assets by default. In this section I'll show you a way to keep your assets organized.

## Asset File Structure

To follow separation of concerns, I prefer to give each type of assets it's own directory.  To follow with the convention of CodeIgniter, we will also include a `third_party` directory, if we want to use any additional libraries or frameworks, such as jQuery or Twitter Bootstrap. The example file structure would look like this:

```
assets/

   stylesheets/

   javascripts/

   images/

   fonts/

   third_party/
```

I personally prefer the longer directory names as I find them to be a bit clearer. Many people prefer shorter names such as '`css`', 'js', or '`img`'. Both are fine.

## Minifying Assets

You development process can be made easier by keeping your CSS and JavaScript broken up into several smaller files. However, this can cause an increased page load, as each individual file creates another HTTP request. The best practice is to keep several smaller files for development, and then when you deploy your application, serve a single minified CSS file and a minified JavaScript file. There are many tools to help with this[12].

[12] 'Minify' is a good one written in PHP: https://code.google.com/p/minify/

# Conclusion

By learning these principles and putting them into practice we these techniques, I hope you are able to write better CodeIgniter applications. ‘Better’ is a pretty subjective word, so what do I mean by that?

- **Your code are more readable.** By following the principles of single responsibility and decoupling, Your application should end up being made up of many smaller parts instead of a few big ones.  By doing this, it makes your code easier to understand. Developers working on the project will only have to focus on smaller, more independent part of the application.
- **Your application is more reliable.** By breaking things into smaller, simpler components, you achieve two other benefits. One is that since your code is smaller and simpler, there is a lower change of bugs, and since your parts or more decoupled, it is less likely for new bugs to be created due to unforeseen side effects that could happen when you change the code.
- **Your application is more extendable.** Since we have made our code easier to organize, manage, and maintain, this means that when it is time to start adding features to your application, it will be much easier to do so.  This allows projects to grow faster, and helps avoid technical debt[13].
- **Your application is simpler.** This is the biggest benefit of all. Complexity is the enemy of software design, and your primary job as a software developer is managing complexity. By following these principles, you can greatly reduce the amount of complexity in your application. The benefit of this may not be clear when you are first starting to work on an application, but by the third or sixth month of working on a large project, you will start to see the benefits, and when something that would have taken you weeks before now only takes days or even hours, You’ll be glad you took the time to read and understand this principles.

## Questions? Comments? Concerns?

I hope you’ve found this guide helpful.  If you any questions or comments, feel free to contact me at glennstovall@gmail.com , Or for more information, visit http://www.glennstovall.com

[13] Basically the opposite of what are trying to accomplish:  http://en.wikipedia.org/wiki/Technical_debt