Practical CodeIgniter 3

From the trenches advice on practical ways to develop web applications with the latest version of this time-tested PHP framework.

Lonnie Ezell

Practical CodeIgniter 3

From the trenches advice and techniques for making the most out of CodeIgniter.

Lonnie Ezell

This book is for sale at http://leanpub.com/practicalcodeigniter3

This version was published on 2016-08-15

Tweet This Book!

Please help Lonnie Ezell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #practicalci3.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#practicalci3

Contents

Introduction

This book is for you–the *developer*–who needs to get a quick answer to "How can I wire this up?".

It's also for you–the *designer* or *entrepreneur*–who has been tasked with building out the skeleton of a new application while you wait for funding to get a hired gun in to take care of the hard-core specifics of the app.

It's not meant as a replacement for the detailed user guide[1]. It won't go into all of the nitty-gritty about each part of the framework. Instead, it is intended to be a practical "field guide" that you reference when you need ideas about how to do things, structure things, or where to put things. It tackles a number of questions that have shown up over and over again in the forums or on StackOverflow, and provides clear, actionable solutions you can start working with today.

If you're an experienced software architect who revels in discovering the *perfect* way to structure your application, and follows all the best practices you learned in school, you might be disappointed. That's not me. Personally, I find that many of the things that you're taught in school are wonderful ways to organize code for maintainability and clarity. But I also think you need to take it all with a grain of salt and not simply jump on the latest bandwagon.

Many of the techniques that have come to be best practices in the PHP community over recent years were originally conceived in a software environment where the code was compiled. Once compiled, it performed very speedily and everyone was happy. PHP isn't a compiled language, though, and those design decisions can, if not carefully used, result in quite slow code when run through interpreted languages like PHP. Yes, I know you can offset that in a server environment with proper caching, proxy caches, and yada yada. That comes with its own cost, though. The more layers you add on top of PHP, the more complex the overall system becomes, and the harder the system as a whole is to maintain. If ease of maintenance was the whole reason we started architecting our apps in that way, it becomes self-defeating at times.

The best answer, in my eyes, is to keep it simple wherever you can. Keep it *practical*. That doesn't mean abandoning all hope of a structured, easy to maintain application, though. Far from it.

This book isn't here to sell you on the gospel of "My Way". It's here to provide quick solutions and inspiration. You should take the knowledge you get here and bend it and reshape it in the way that works best for you and your organization.

[1] http://www.codeigniter.com/userguide3/

The Book At A Glance

If you need to know a solution *right now* then this is a great place to start, since what follows is a very quick overview of what each chapter covers. Hopefully, it can point you in the right direction quickly.

Introduction

You are here. See the red 'X' on the CodeIgniter treasure map? Yup. That's you.

Chapter 1: Where Does This Go?

While everyone seems to have a slightly different variation of MVC definitions, this chapter will start the book off by describing how I like to think about the different file types CodeIgniter provides: models, controllers, views, libraries, and helpers (Oh, My!), and why they work best that way. It also briefly touches on how to map your application into controllers for easiest maintenance.

Chapter 2: Environmental Protection Codes

This chapter will examine the all-too-underused capabilities of Environments. I'll show you how to set them up for maximum flexibility and ease-of-use, and we'll discuss why they are so necessary. Especially when working in teams and using version control.

Chapter 3: Controlling Traffic

You've already learned what should and should not live in a controller, so here I'll show you how to create multiple controllers to keep your code DRY[2] and separate different areas of your application. We'll show you how to load them both with, and without, one of the common HMVC[3] solutions.

Chapter 4: Showing Your Work

This chapter digs into working with views and view data. Once you understand the basics, I'll show you the way that EllisLab intended the views to work. Next, I'll show you a simplified version of my convention-based theme system I borrowed from Rails years ago. From there, we'll explore the parser and how to integrate other template engines, particularly Twig.

Chapter 5: Working With Data

In this chapter, we'll dig deep into using the database. Everything from basic setup to using multiple databases to separate read and write queries. We'll look at setting up a basic `MY_Model` to provide utility functions to increase flexibility and productivity. We'll take a look at how to validate data in the model, how to use the migration system, and even how to create our own Seeder.

[2] DRY or "Don't Repeat Yourself" is a principle aimed at reducing repetition of information and/or code. https://en.wikipedia.org/wiki/Don%27t_-repeat_yourself

[3] HMVC or Hierarchical Model-View-Controller is a variation of the MVC architecture. https://en.wikipedia.org/wiki/Hierarchical_-model%E2%80%93view%E2%80%93controller

Chapter 6: AJAX is Easy

I often see questions about working with AJAX in the forums, so this chapter provides the necessary knowledge to integrate AJAX into your applications with ease. We'll use the Output class to ensure our data types are sent correctly, build some utility methods into our `MY_Controller`, and explore some simple jQuery setup and utilities.

Chapter 7: Working With Files

We'll look at the fabled multiple-file upload problem that plagues the forums. We'll also explore using FTP with some real-world examples, and working with remote files quickly and easily.

Chapter 8: Multiple Applications

In this chapter, I'll show you how to use CodeIgniter's multiple application setup to address several application needs, like separating out sections of the site for server needs or simple desires, and even how to share common code between different applications, or an API and Admin area.

Chapter 9: Security

In this chapter, we'll scan through a number of topics that are crucial to understand if you wish to maximize the security of your application. We will make sure you understand when and why to use tools like the built-in CSRF and XSS protection, how and when to filter and sanitize your data, and more.

Chapter 10: Performance

Here we examine a number of things we can do to help increase the performance of our application, some obvious and some maybe not-so-obvious. We will cover several caching strategies and the types of caching available to you, database system tweaks for production environments, and even when and where to use CodeIgniter's autoloading features for the best performance.

Chapter 11: Fun At The Terminal

While CodeIgniter has always seemed to shun the terminal, there are times you can't get away from it. We'll explore how to work with CodeIgniter from the command line, the basics of creating CLI scripts of your own, and even how to make a simple, but flexible, cronjob runner.

Chapter 12: Composing Your Application

You've probably heard of Composer and how it's shaking up the way PHP is written and shared. This chapter starts by exploring how to use it within your application to simplify using some of the many high-quality packages available. We will look at `codeigniter-installers` and how to use it to share your CodeIgniter-specific code. Then we'll take a look at how to use Composer to provide a completely different way of working with CodeIgniter than what you're used to: a way that is more flexible, powerful, and more future-proof.

1. Where Does This Go?

A common question in the forum of any framework is, “Where do I put this information?” The answers are usually fairly varied, though most do stick close enough to traditional MVC patterns. The reason it varies so much, though, is simply due to the types of projects each developer has worked on and the environments those projects were created in. Freelance developers might have stumbled onto the way they use the data based on sheer luck or hours reading blogs. Large companies might have someone dictating the organizational patterns used by the rest of the developers, or even internal base classes which make a specific structure easier or more productive.

In this chapter, I’m going to show you the way that I approach this subject. If you’re new to CodeIgniter, I believe that adopting these methods will help keep your projects organized and easy to maintain. If you’ve been using it for a while, or are coming from another framework, and your habits are slightly different, that’s ok. I’m not here to convert you, but I will let you know *why* I think this approach works best. From there, you’re free to choose how you want to use it. I won’t come to your business and critique your work. And CodeIgniter doesn’t mind, either. It’s flexible enough to support almost anything you can throw at it.

MVC - Easy as 1 2 3

CodeIgniter is based loosely around the MVC pattern - Models, Views, and Controllers. There are many variations on exactly what goes in each, so let’s first cover what each of those are so that we’re all on the same page.

Models

Models are the central piece of the puzzle. In traditional definitions of MVC, they were responsible for notifying the views of changes and letting the controllers know so that it could change the available commands. In our web world, the model layer is responsible for the data itself and handling any business logic around that data.

In some frameworks the model is split into two layers: the data access layer, which only handles getting the information from the database; and the business logic layer which handles validation of data, ensuring formatting is correct on the way in and out, making sure that other items are updated when an action happens in this model, etc. In CodeIgniter, all of this has a tendency to be handled within the Model itself, though I’ll make a case for splitting it between Models and Libraries later.

Views

Views are simply displays of data. They are normally "dumb" HTML files, though newer thick-client applications may have very smart views that handle updating themselves in a very dynamic way. Even though they may provide a lot of interaction with the user, they are still a way to view that data, though JavaScript may add additional layers of MVC at that point.

While they are primarily used to display HTML to the user, they can also be used to format an XML sitemap or an RSS Feed, CLI output, or any other format you desire. Views can represent a whole page, but will more often represent fragments of the information to keep repetition to a minimum.

Controllers

Controllers take care of moving the data between the model and the views, and directing traffic based on user actions. For example, when a user submits a form, the controller collects the data and shuffles it off to the model, which should ensure the data is valid. The controller then tells the View (or JavaScript layer) whether it was successful.

There are a number of reasons to keep your controllers light and relatively dumb, but I'll focus on one big reason. It keeps all of your application-specific rules in one central place: the Model layer. That model can then be used in different applications with the same result.

You might have an application that is already up and running. The client comes back and asks for an API for that data so they can make a mobile application. With the data and business rules all handled in one location, you simply need to share those models with the new API application and any changes in business logic will be ready for both applications.

Or you might be working for a large insurance company. Because of the way their data needs to be handled, they have a number of separate applications that all use the same data. And that data needs to be handled the same way, whether it's generating millions of monthly reports to send to their clients, or allowing the administrators to run "what if?" scenarios on the data to see how potential changes might affect the company.

So, What Goes Where?

Now that we've covered the basics of MVC and have a common way of looking at things, let's look at 2 more types of files that CodeIgniter supports: Libraries and Helpers. Then, we'll explore some common use-cases and look at ways to organize these files to help solve these problems.

Helpers

Helpers are intended to hold procedural functions. These can be any functions that are outside of a class.

"Wait!" I hear you yelling. "This is supposed to be OOP programming. What gives?!" First, nothing will ever be 100% object-oriented. You always have to have your bootstrap file that runs through and instantiates all of the needed classes. So forget the idea that you'll ever get rid of procedural code and functions. It's a lie.

Helpers, then, are used to provide new functions that can be used easily to provide custom functionality. You can use these functions to provide compatibility functions for when the command doesn't yet exist in your version of PHP. They can be used in form validation callbacks to provide custom validation methods. They can also be used to do some simple or common formatting within views since, once loaded, they're always available. You'll also find third-party code that is implemented as a function that you want to use. A common example, for me anyway, is the Markdown Extra libraries. For years I've used those, though I've recently switched over to the PHP League's CommonMark libraries. (No, I didn't switch because it was an OOP solution.) The fact is that functions still have their place, and helpers provide a simple way to load them. Simple as that.

Libraries

Libraries were EllisLab's way of making it easier to load any class that was not a controller or model. This was before the days of namespacing and good autoloading in PHP. Libraries can be used to incorporate third-party code, though more often than not, you'll simply use autoloading and won't need anything special for third-party code.

Libraries can be used to provide common functionality that you want accessed by different parts of your app. They help provide a single point of entry that makes it simple to maintain consistency in business rules. They're ideally used for creating focused, single-responsibility classes, like Benchmarking, UserObject, Article, etc.

The thing I want to emphasize here, though, is that they were the way to handle **any class that was not a controller or model**. I think this is a very important concept when using CodeIgniter so that you don't paint yourself into a corner when it comes to using other classes. Libraries are not just a file that must contain stand-alone information that can be passed from project to project, though that is the traditional role in CodeIgniter applications. It's just not the *only* use for libraries. Many new paradigms are becoming common in PHP programming that have come from more traditional software development. You can now have Entity Models, Factories, Adapter classes, Decorators, and many other classes that help fulfill different design patterns. The one thing they all have in common is that they are a class. In CodeIgniter, the only way provided to load these classes is as a library. The other options are to hardcode `require` statements or use an autoloader like Composer. Personally, I highly recommend Composer, and will discuss that more in a later chapter.

Earlier, I said I'd make a case for splitting the business logic across Libraries and Models. Let's tackle that now. The more I work with CodeIgniter, and PHP in general, the more I find situations where following this simple rule of splitting it into two layers would have saved me time refactoring. So, to save you that time, here's the primary reason I recommend splitting them up. Depending on how you structure your app, you might find other benefits along the way.

By keeping the Library as your object's unchanging, backwards compatible API, and using your Model as a basic data-access layer, you never have to worry about *how* you get that data. If your company decides that they need to switch this one resource from a persistent MySQL storage base over to using a "temporary" Redis-backed solution, you're ready because you can hook up to any model. You could even provide both in different situations, maybe with the lightning fast Redis as your primary store that's backed up occasionally in the background to MySQL for permanent storage.

The Library can also provide formatting of objects before they get to the rest of your code. Then, if your database schema changes, it won't automatically break your application because the library is keeping a consistent format. Alternatively, you might have a reporting resource that needs to pull in data from several database tables, and process the results, before returning it to the rest of your app. While you could do that in your Model layer, a library provides a cleaner mental picture of how the app works and makes changes that much clearer.

One method of implementing this separation is known as the Repository Pattern[1], and becomes more important as your application and team get larger and more complex.

As with anything, though, application architecture is more art than science. Splitting up the layers in smaller situations might be overkill. On large projects I'm becoming a big believer in it.

Third Party Code

There are two common methods for handling third-party code effectively within CodeIgniter. Third-party code could be a single class or several classes making up a full-featured suite of related code, like Monolog[2].

The first, and becoming increasingly more popular, method is to use Composer to bring in and manage third-party code. In this case, it will all be handled for you behind the scenes and you don't have to worry much about it. More details will be given in Chapter 13: Composing Your Application.

What about libraries of code which are not packaged up nicely for Composer? One example of this might be libraries provided by a credit card processing company. If they don't provide a Composer package, then you're stuck trying to figure out how it fits within your application's structure, and how to load it. My advice? Forget about CodeIgniter, and remember that you're simply programming in PHP with CodeIgniter to help out.

In these cases, place the code in its own folder within `/application/third_party` and simply use `include()` or `require()` to load the file, then you can make a new instance and use it however you need to. To make this a little easier and less fragile, be sure to use the `APPPATH` constant when you include the file.

[1] http://www.sitepoint.com/repository-design-pattern-demystified/
[2] https://github.com/Seldaek/monolog

```
require APPPATH . 'third_party/mypackage/myClass.php';
$package_class = new myClass();
```

Use Cases

So, let's dig into a few examples of how we'd implement some code for some real-world use cases.

Simple Blog System

For the first example, let's use the standard Blog system. This isn't anything very fancy, just the basics. We are going to look at the front end part of the module, ignoring any admin code.

Controller - The Controller will simply handle moving things back and forth between the library and the views. It might have the following methods:

- **index** to display the 5 most recent posts with their body text.
- **show** to display a single post with all details, comments, etc.
- **category** to list all posts belonging to a single category.
- **archives** to display all posts published within a year and month passed in the URI.

In addition to these page-specific issues, it should take care of any common data that will need to be displayed on all of the blog pages. It would likely need to grab a list of the most recent posts and make it available for the sidebar. It might also need to grab a list of tags or categories for the sidebar.

Library The library would contain very specific methods for grabbing the data from the database, by way of the model, making sure it's in the proper format, and returning the data back to the controller. It might have methods like `recent_posts()`, `posts_by_category()`, `posts_by_month()`, etc. This would need a method to display a simple list of recent posts with a headline, image, and URL for sidebars.

Model The model would contain a set of basic methods for getting data out of the database, like `find()`, `find_many()`, `insert()`, etc. It would validate each field whenever a new element was inserted or updated. It might contain logic embedded within the various methods to destroy a cache whenever a record was updated or created, etc.

Views There would be several views, but you want to minimize repetitive display of posts as much as possible. This means you want to create a single view responsible for displaying the post itself. You could have a little bit of logic here to determine if it is showing a full post or just an excerpt. This post view could then be re-used by the category page, the archives page, the overview page, and the individual page. You would also have smaller views to handle the recent posts, or a popular posts block that would end up in the sidebar.

Views should not have any business logic in them. The logic in a view should be restricted to simple things related to the display of the data, like `if...else` blocks so that you can provide good blank

slates[3] when no data exists. Almost everything else should be handled by your controller, a library, or your model.

Survey Generator App

This app allows a user to create a survey by selecting from a number of question types, like yes/no questions, text-only answers, multiple choice, etc. The questions must be reusable between surveys, have their own scoring rules, etc. What you end up with is a number of different special cases for each question type.

For this example, we won't focus on the entire application, but instead drill down into how we might use the standard file types for this situation.

Libraries To handle the different question types elegantly, we will need to create a single `BaseQuestion` class. This class would implement default, generic versions of the functions needed by a question, like generating an answer based on the scoring rules for that question, determining maximum possible score, etc. Each question type would then need its own class that extends from that `BaseQuestion` to customize the way it handles scoring questions, etc.

In this case, I would be tempted to use CodeIgniter's `load->library()` function to load the individual questions, but instead I use a `require_once` in each of them to pull in the `BaseQuestion` class. This is mainly because I don't feel the `Loader` really needs to know about the base class and use storage and memory for that.

If I was using Composer[4] in the project, then I would skip the libraries and make them simple namespaced classes, since it would make everything a little cleaner. If you're not familiar with Composer, I highly recommend you take some time to learn it, because it can change the way you code. A later chapter is devoted to using Composer with CodeIgniter, and there are many, many resources available on the internet to get you up to speed with using Composer in general.

Additionally, you will need a central library for handling and formatting your question data consistently. This might need to pull in data from several models in order to provide quality statistics and other reporting features.

Models While you will often end up having a single model represent a single database table, when you are dealing with applications that need to return statistics, like this app would provide to the survey owner, you will often need to switch your thinking. Instead of a single table, a model can represent a single *domain*, or area of responsibility. This means it can pull in data from a number of tables to generate the results it needs.

In this example, we could have a `Survey_model` that, in addition to its standard required functions, would be able to pull in data about statistics for each survey. You could also create a separate `Survey_stats_model` that you would use within the `Survey_model` to keep a clean separation of responsibility.

[3] http://blog.teamtreehouse.com/tips-for-creating-a-blank-slate

[4] http://getcomposer.org

Views While you would have all of the standard views for displaying the various forms for making the survey, and for the user to fill it out, a little special consideration would be needed for the questions. Since each question will have different needs when it comes to displaying them, you will need to create new views for each question in three different modes: create/edit, answer, and reporting. Then you can loop through the available questions and pull in a view for each depending on its type.

Controller Design

While we're talking about use-cases, I want to touch an issue for less-experienced developers: controller design. What I mean by this is how controllers are broken up, which methods they contain, etc.

It's all too easy to put too much in a single controller. Like anything else, though, you need to keep controllers clearly organized and simple to understand. Both for you, when you come back to debug something in a few months, or for new developers on a project that are forced to try to understand what is where.

The simplest way to look at this is to write out a list of what you want your URLs to be, then work from there. This helps to keep you thinking about the problem clearly, without getting caught up in the intricacies of the code required for the application.

Let's run through an example, and I'll explain more as we go along.

This application is going to be a Customer Relationship Management software that allows you to manage your clients, potential clients, supplier companies, and cold leads. Since we want to be able to view and manage the individual people as well as the companies, we could start our URL mapping with something like the following.

```
/people
/people/add
/people/remove
/people/{id}/contact
/people/{id}/history
/companies
/companies/add
/companies/remove
/companies/{id}/people
/companies/{id}/people/add
/companies/{id}/people/remove
```

With this starting point, it would make sense that you have one controller called `people` and one controller called `companies`. Inside each of those, you would have a method to handle the various

actions, using the router to allow for the `object/{id}/action` URLs to map to a class and method like `People->contact($id)`.

In addition, you would likely want some AJAX methods for adding notes to people or companies. You might also want AJAX methods for adding reminders to the app's calendar so that you'll know to follow up with the client in 3 months. I've seen a natural tendency to want to create a single AJAX controller to handle all of that. However, that solution tends to create very large classes that are difficult to maintain, and it occasionally makes it difficult to locate the proper method.

Instead, you should create separate controllers for each of the areas of concern, like *notes* or *reminders*. In smaller applications, these controllers could hold both the AJAX and non-AJAX methods. Once your application starts to grow, you should consider creating an AJAX directory located at `/application/controllers/ajax/` and putting any AJAX-specific controllers in that folder. This makes everything very clear for you and any other developers. It also makes it very simple and logical to locate the method you have to edit.

Another way to start discovering the structure of your controllers is to look at your site's main navigation from the designer's mockups. This can be especially helpful if your actions are not all based around *nouns* (like people or companies). You might have a client area with main navigation focused around a Dashboard, Survey Results, and Leaderboards. In this case, you might be tempted to simply combine those four or five pages into a single controller called clients. I think a better way would be to create a new directory to hold several new controllers.

```
/application/controllers/clients/
/application/controllers/clients/Dashboard.php
/application/controllers/clients/Results.php
/application/controllers/clients/Leaderboards.php
```

While it might seem at first glance that each page will only have a single method in it, you will often find that to be untrue. The Leaderboards controller would have the method to display the leaderboards, the `index()` method. It might also have additional methods to help view other aspects of the boards, for example by a certain year, a certain client, etc. It is possible to handle all of that with a single method and filter the results based on `$_GET` variables, but it is clearer to the end_user if you handle each aspect in a different method within the same controller, because the URL will tell them exactly what's going on.

```
Leaderboards->index()
Leaderboards->by_year($year)
Leaderboards->by_company($company)
```

It also encourages you to wrap your logic up into a neat library or model so that you don't repeat code. Which is what you should be doing in the first place.

As already mentioned, the purpose of all of these separations are to make things as easy to find and digest, mentally, as possible when you come back to the project, or when you have multiple developers working on the project together, or a new developer takes over the project from you. All of your decisions should be based around those aspects, and how you can make things clearer and more apparent to anyone involved. It doesn't hurt your reputation, either, to have people talk about how your code is easy to work with and clean.

Packages

While we're talking about where your code should go, it would be remiss to overlook packages. Packages[5] allow you to bundle nearly any of the standard CodeIgniter files into a "package" that you can easily redistribute. You can include any of the following file types: libraries, models, helpers, config, and language files.

You'll notice the one missing file type is controllers. This means that you cannot route user actions to anything within that package, but they still have great uses. EllisLab originally added packages to CodeIgniter when they decided to use CodeIgniter as the base of their Expression Engine CMS. They needed a convenient, high-performance way to include third-party modules without interfering with the application or the system.

Why Packages?

I've seen three primary reasons for using packages. The most common reason is simply to keep code which shares a common purpose wrapped up in a neat package that can be easily distributed and reused. You might make it available on GitHub for others to use, or you might want to keep a library of common code that you've used on projects for your own use. With each new project you can simply pull in the functionality, build a controller to interact with it, and you're set. You might even have a number of smaller sites on one server that all reference shared code for additional functionality.

Some teams of developers like to use packages or modules as an organizational tool. A single developer, or a small team, would be tasked with working on one dedicated part of the application. This keeps their code isolated and simple to maintain.

The last reason is to separate out common code that you want to share between multiple applications[6]. You might be working on a large site for selling automobiles with an admin area, a dealer area, and the front-end customer area. To keep things prepared in case your application takes off and you need to move each area to its own server, you might choose to keep each of those three areas as a separate application. You want them to share the libraries and models, though,

[5] http://www.codeigniter.com/user_guide/libraries/loader.html

[6] http://www.codeigniter.com/user_guide/general/managing_apps.html

to avoid any duplication. You could create a new `common` directory and make sure that your other applications know about this new package. Then, when you deploy, you can have a build script delete the unused application from that server, or simply copy only the required application and the common folder into place. This will be discussed in detail in Chapter 9.

Using Packages

While the user guide[7] gives you all of the details, we'll quickly cover them here so that you understand not just how to set them up, but how they work and why they were designed this way.

A new installation of CodeIgniter doesn't know anything about your packages or where to find them. It also won't look for them when trying to find a library or other file, and this is great for performance. The fewer directories that the system has to scan for files, the faster it can find what it's looking for. So, you have to explicitly tell the system where to find the package, either through autoloading or at runtime.

When you tell it about the package, it will add a new directory to scan for any requested files (except controllers, of course). These directories are then scanned on each load request, starting with the main application folders, then branching out into the packages, in the order they were added. This keeps any packages from overloading core functionality, which is generally a good thing.

Autoloading

You can tell CodeIgniter to "autoload" packages by adding the entry to `application/config/autoload.php`.

```
$autoload['packages'] = array(APPPATH . 'third_party', '/usr/local/shared');
```

Autoloading of packages is extremely light-weight since all it does is add the directories to the paths to be searched when loading files. No files are read, it doesn't even confirm that the directories exist before adding them. This is the perfect solution if you have a common package that you will always need to use. Set it once in the autoload config file and forget about it.

Runtime Package Management

If you want to add packages at runtime you can use the `add_package_path()` method.

```
$this->load->add_package_path(APPPATH . 'third_party/' . $path);
```

This can be used a couple of different ways. You could load up a list of modules that are installed and active, then loop over them, adding each module's package path in the controller's constructor.

[7] http://www.codeigniter.com/user_guide/libraries/loader.html

```
foreach ($modules as $module => $path)
{
    $this->load->add_package_path($path);
}
```

However, this creates an unneccessarily large number of folders that have to be scanned through each time. To increase performance, you can limit the packages used in some fashion that allows you to add the package, call its functionality, then remove the package path immediately. This way no extra paths are stored unless they are needed.

```
$this->load->add_package_path($path)
           ->library('foo_bar')
           ->remove_package_path($path);
$this->foo_bar->do_something();
```

Modules

The next logical transition that many people move on to from the built-in Packages are Modules. Modules simply allow you to add and route to Controllers. Other than that, they are the same thing as CodeIgniter's Packages. These carry all of the same benefits as packages, so I won't go over them again here.

There are currently two solutions for adding module support to CodeIgniter 3. The first is WireDesignz' Modular Extensions - HMVC[8]. It's the oldest and most well-known of the two. The second is Jens Segers' CodeIgniter HMVC Modules[9], which takes a slightly different approach, but achieves the same goals.

After running both of them through a series of un-scientific tests, I found the performance of both of them to be close to native CodeIgniter speeds. They do both have a small impact, but not enough to worry about in the real world. When running under PHP 5.5.9 with Opcode caching turned on, the difference was less than 10% fewer requests served than on raw CodeIgniter. Of the two, Jens had the edge at being slightly faster.

So, for storing modules, either one works very well, and both have their perks. WireDesignz' has a few more features, like integrated autoloading of files from the libraries and core directories, while Jens' has more options for how the router locates the controller and method to run. WireDesignz' has a known complication with the Form Validation library, but the solution is listed on his site and fairly easy to implement. Jens' didn't say anything about that on his site, though I have had the same issues with it in the past.

The biggest difference between the two are how they handle the HMVC, or Hierarchal Model View Controller setup.

[8] https://bitbucket.org/wiredesignz/codeigniter-modular-extensions-hmvc
[9] https://github.com/jenssegers/codeigniter-hmvc-modules

HMVC

HMVC is simply a fancy way of saying modules that contain Models, Views and Controllers, and allowing you to call the controllers from other controllers like a library. In practice there are two reasons that people like to use HMVC solutions with CodeIgniter, instead of being happy with simple modules.

The first is for more of a "web services" architecture that allows modules to behave as mini-applications in their own right, and be easily moved to another server if you need to scale your application[10]. Jens' solution supports this type of HMVC, but in a limited manner. For a true HMVC solution to work with web services like this, you'd have to be able to route across some form of connection, like a standard HTTP connection. Jens' is not designed to do this. Instead, it merely provides the ability to organize your code in a hierarchy of MVC triads. Depending on who you ask, this can be a good thing or a bad thing. WireDesignz' solution does something similar here, by allowing you to load a controller and then use it like you would any other library.

The second use is to create reusable "widgets" that make it simple to insert small chunks of HTML into multiple views, but without the headache of having to load that information in every controller. This is handled by WireDesignz' solution but not by Jens'.

So it boils down to one question: should you use HMVC instead of just plain modules? My answer is no. While I'm a huge fan of modules for code reuse and distribution, the other abilities of HMVC seem to pale among their complexities, especially in the CodeIgniter world.

CodeIgniter was never designed to have what amounts to child requests being processed in a parent request. At its core is the way everything belongs to one central core element (or all controllers/models through the Controller instance) that is a singleton. Whenever you load the controller again, it creates a nested version of this controller. It quickly becomes a messy situation where you cannot always trust that `$this` is what you think it is. This is very obvious when you have to start tweaking things to get Form Validation to work. As a workaround, the HMVC solutions create another instance of the singleton and pass that around. While that can work well in most cases, it's also, I believe, what has caused the issues you see with the Form Validation library not working correctly.

The headache that it can cause during debugging is not worth the additional benefits to me. While I like the idea of reusable "widgets" in the view layer, it's simple to code a solution in a single file, and we will cover that in Chapter 4: Showing Your Work.

Closing

Hopefully, this chapter has helped clear up any confusion about where things go in your application. As you're working on your applications, though, remember that you don't need to stress overly much about it. I'm sure people will argue with me, but as long as things are in the "generally correct"

[10] http://techportal.inviqa.com/2010/02/22/scaling-web-applications-with-hmvc/

location, you will be fine. There is no perfect answer for every situation, but I think the advice I've given here will help to prevent situations in which major refactoring due to bad structure will be needed.

Just keep striving to learn from your mistakes, implement those changes in the next project, and keep moving forward. Having stuff in the "wrong" place usually won't break your application. Besides, you can always refactor later if you need to.

2. Environmental Protection Codes

CodeIgniter supports the idea of multiple Environments[1] in which your application may run. This allows you to configure your application to behave differently in different server environments. It's not as scary as it sounds, and it's extremely useful.

The environment itself is simply a way of letting the application know whether it's running on a developer's personal server at the office, or the production server that's running the live site. Depending on your workflow, you might have one or more staging or QA servers that function like the real environment, but use different databases, etc.

First, we will look at how to determine which environment is running. Then, we'll take a look at what changes are made to the system. Finally, we will run through a few different ideas for using the different environments in your workflow.

Determining the Environment

The environment is determined on each page run at the top of the main `index.php` file.

```
39  /*
40   *---------------------------------------------------------------
41   * APPLICATION ENVIRONMENT
42   *---------------------------------------------------------------
43   *
44   * You can load different configurations depending on your
45   * current environment. Setting the environment also influences
46   * things like logging and error reporting.
47   *
48   * This can be set to anything, but default usage is:
49   *
50   *     development
51   *     testing
52   *     production
53   *
54   * NOTE: If you change these, also change the error_reporting() code below
55   */
56          define('ENVIRONMENT', isset($_SERVER['CI_ENV']) ? $_SERVER['CI_ENV'] : 'develop\
```

[1] http://www.codeigniter.com/user_guide/general/environments.html

```
ment');

/*
 *---------------------------------------------------------------
 * ERROR REPORTING
 *---------------------------------------------------------------
 *
 * Different environments will require different levels of error reporting.
 * By default development will show errors but testing and live will hide them.
 */
switch (ENVIRONMENT)
{
        case 'development':
                error_reporting(-1);
                ini_set('display_errors', 1);
        break;

        case 'testing':
        case 'production':
                ini_set('display_errors', 0);
                if (version_compare(PHP_VERSION, '5.3', '\>='))
                {
                        error_reporting(E_ALL & ~E_NOTICE & ~E_DEPRECATED & ~E_STRICT & ~E_USER_NO
E & ~E_USER_DEPRECATED);
                }
                else
                {
                        error_reporting(E_ALL & ~E_NOTICE & ~E_STRICT & ~E_USER_NOTICE);
                }
        break;

        default:
                header('HTTP/1.1 503 Service Unavailable.', TRUE, 503);
                echo 'The application environment is not set correctly.';
                exit(1); // EXIT_ERROR
}
```

The first section, labeled **APPLICATION ENVIRONMENT** is where the actual environment is set. Unless you tell it differently, it will default to an environment named `development`. There are a few different ways that you can tell CodeIgniter which environment it is currently running in.

No matter how the environment is detected, the end result is always a newly defined constant named `ENVIRONMENT` that can be used anywhere in your application to modify how the app works

at run-time. We'll cover a few different use cases for this below.

Environment Setup under Apache

If you look at line 56, you'll see that it's checking for an environment variable called `CI_ENV` in the `$_SERVER` array. You can set that pretty easily in your site's main `.htaccess` file (or other server configuration files) with the `SetEnv` command. You would add this line anywhere outside of any `<IfModule>` or similar blocks.

```
SetEnv CI_ENV production
```

This would set the environment to `production`.

Environment Setup under nginx

Under nginx, you must pass the environment variable through the `fastcgi_params` in order for it to show up under the `$_SERVER` variable. This allows it to work on the virtual-host level, instead of using `env` to set it for the entire server, though that would work fine on a dedicated server. You would then modify your server config to something like:

```
server {
    server_name localhost;
    include     conf/defaults.conf;
    root        /var/www;

    location    ~* "\.php$" {
        fastcgi_param CI_ENV "production";
        include conf/fastcgi-php.conf;
    }
}
```

Manual Setup

A third option is to manually define the environment by changing line 56 for whatever server you are running on.

```
define('ENVIRONMENT', 'production');
```

The biggest drawback with this method, though, is that you must manually change the variable here whenever a new set of changes is pushed to the server. This is far from ideal, and is only recommended if you have some form of automated deployment script setup that can modify the file for you on the fly as it is deployed to each server.

Dynamic Manual Setup

The most flexible way to handle this is to have the script check the host it's running on and compare it to a set of known domains to determine the correct environment. You might also provide a fallback to look at the domain name for common patterns that you and your team use to name your virtual hosts on your development machines.

```
39 $domains = array(
40         'test.myapp.com' => 'staging',
41         'myapp.com'      => 'production'
42 );
43
44 // Have we defined a server for this host?
45 if ( ! empty($domains[$_SERVER['HTTP_HOST']]))
46 {
47         define('ENVIRONMENT', $domains[$_SERVER['HTTP_HOST']]);
48 }
49 // Or is it a development machine, like myapp.dev?
50 else if (strpos($_SERVER['HTTP_HOST'], '.dev') !== FALSE)
51 {
52         define('ENVIRONMENT', 'development');
53 }
54 // Else - be safe...
55 else
56 {
57         define('ENVIRONMENT', 'production');
58 }
```

In this version, we create an array of allowed hosts and their corresponding environments, then we check to see if the current host matches one of these. If the current host isn't found, we check the hostname to see if it ends in `.dev`, like `myapp.dev`. This matches my local naming scheme, but you could easily modify this to match `myapp.local`, `dev.myapp.com`, or anything else your team might already use. Finally, if no match is found, we assume it's production for the strictest default settings.

This is my preferred method for determining the environment, simply for its flexibility and the ability to "set it and forget it". You don't have to worry about maintaining multiple versions of the .htaccess files to specify production or staging servers. Once you setup the rules, you save the index file to your version control system, and things simply work as expected whenever you push code to any of the servers.

Environment Configuration

Once you have decided how to detect and set the current environment, you need to setup your application to really take advantage of this. Out of the box, CodeIgniter sets the error reporting to sane defaults based on the environment (see lines 66-90). These make great default settings, but you should feel free to modify them to meet your company's (and application's) needs.

The most common use for server environments is going to be setting up configuration files based on the environment. Common things you would want to vary based on environment might be:

- Database settings, so you access the right database(s) for each environment.
- Email library configuration, so you can use different email addresses/servers based on domain name, or even simply use `mail` in development but setup an `SMTP` connection everywhere else.
- Third-party API settings, so you're not mixing test API calls into your live data.
- Different cache settings in development, where you want to always see the changes instantly, or in production where long-term caching is a benefit.
- Prefix the page title with an abbreviation for the environment so you can tell with a glance at the browser tab which environment you're looking at.
- And many more uses depending on your exact application.

A Few Uses

Configuration Differences

The simplest use-case for environments is to change configuration settings based on the environment, and we just went over a few examples of this. Here's **how** you do it, though.

You can set default configuration values in the standard CodeIgniter configuration files. You can opt to use the settings for your production site, or to leave the default settings and keep things in their separate environments for clearer organization. The latter method is the method I generally like to use, because there's less chance for confusion. Everything is in its place and clear for you, your team, and any future developers that might work on the project. It also keeps accidents from happening when production settings are accidentally used in development environments.

For each environment which will have its own settings, create a directory named after that environment within the config directory. For any settings you need to change, you would copy or create a new file with the same name as the standard CodeIgniter configuration file, but under the directory named for the environment. This file is read in after the main configuration file, so you really only need to copy the changed values into the new file.

To setup the database settings for our development environment, we first create a new directory, `/application/config/development`. This directory's name matches the environment. Then we

would copy `/application/config/database.php` to the new directory (`config/development/database.php`) and modify the new file's settings for our local, developer-specific database.

Don't Commit Development Config Files

When using a VCS system, like git (and if you're not, you should be!) NEVER commit your development environment settings. These settings should be different for each developer on your team, and committing them would only cause frustration between team members. You should use the `.gitignore` file (or your system's equivalent) to always ignore files in that directory so they don't get accidentally committed.

Auto-Migrations

One thing I will commonly do is to setup my system with the capability to automatically perform database migrations[2] when the page is visited. However, I only ever want this to happen in the development environment. Sometimes on the staging server, depending on the client I'm working with at the time, but never on the production server. We like to maintain total control over the process on the production server, and each client has slightly different needs when pushing to that environment. Plus, performing the database migrations automatically may have some performance implications, so you may want to have more control over when this occurs.

Setting this up is fairly easy, and something we will go over in detail in Chapter 6.

Development Tool Routes

I have had some sites where we had some tools that we needed available to the developers, but would be dangerous to make available in production. These might be tools that:

- Allow us to easily do some maintenance or browsing of the database.
- Allow us to "fix" some data that occasionally becomes corrupt, allowing the site to continue until we find the true fix.
- Allow us to push the current database data to one of the other servers.
- Allow us to browse raw data feeds from external sources to detect changes in schema, or help while creating data imports.

In times like this, I would check the environment in `/application/config/routes.php` and either show or hide the routes as needed.

[2] http://www.codeigniter.com/user_guide/libraries/migration.html

```
if (ENVIRONMENT === 'development')
{
        $route['devtools/(:any)'] = 'developer/tools/$1';
}
else
{
        // Don't allow access to these routes at all.
        $route['devtools/(:any)'] = '';
}
```

Asset Compilation

While I typically use a tool like CodeKit[3] for this now, it's not always feasible. At times, the team you're working with might use something else. You can add a method to the constructor of `MY_Controller`, or even use a pre-controller hook[4], to fire off commands that will compile and compress any CSS stylesheets or Javascript files.

Debugging Tools

If you use tools like PHP Error[5], Whoops![6], or Kint[7], you will probably want those only in your development environment. It doesn't make sense to even load them up in the other environments as it's an unnecessary waste of resources. You can modify your autoload file to check for the environment, too.

```
1  /**
2   * Load PHPError
3   */
4  if (ENVIRONMENT === 'development' && ! is_cli() && config_item('use_php_error'))
5  {
6          require(__DIR__ . '/../php_error.php');
7          \php_error\reportErrors(array(
8                  'application_folders' => 'application',
9                  'ignore_folders'      => 'system',
10                 'enable_saving'       => false
11         ));
12 }
```

[3] https://incident57.com/codekit/
[4] http://www.codeigniter.com/user_guide/general/hooks.html
[5] http://phperror.net/
[6] http://filp.github.io/whoops/
[7] http://raveren.github.io/kint/

Conclusion

Environments are a powerful feature that every developer should be using to make their lives a little bit easier. My hope is that this chapter has not only shown you how to use them, but sparked your imagination with different ways to use them.

If you have other ideas of how to use environments, I'd love to hear them!

3. Controlling Traffic

There are two main parts of the framework that are hit when a user visits your web site or application: the Router and a Controller. The great news is that you have control over the path which the request follows through your application.

The first part of this chapter will be a look at the Routing portion of the request. The second part will examine Controllers in all of their magical glory. Finally, we'll look at a number of common ways I've found that are handy to tweak your controllers to make them a pleasure to use.

Routing Traffic

When a user visits your site, CodeIgniter will run the URI through the router, looking for any matches in the routes you specified in the routes config file at `/application/config/routes.php`. These routes are really just patterns to match the URI against, along with instructions of *where* to send the requests which match each pattern.

If no match is found, it will try to find a controller and method that match the URI.

If this fails, it will go one step further and look for a "default controller" to process the request.

Failing that, it will throw a 404 error, telling the world that the requested page cannot be found. Don't worry, you can control how the 404 is handled if you need to.

Magic Routing?

Lately, the matching of the URI against the controller/method pairs has taken on the name "magic routing" in the forums and–while not at all proper in Computer-Science speak–I kind of like it, so I will use it here.

In the following sections, we will take a detailed look at how each of the elements work, how you can take control of them, and the enourmous amount of flexibility they provide.

Default (Magic) Routing

If not given any other orders in the `routes` config file, the router will attempt to match the current URI against an existing controller and method. Let's look at a few examples to make sure this is all clear. More specific details are also available in the CodeIgniter user guide[1].

[1] http://www.codeigniter.com/user_guide/general/routing.html

- **/users** will look for the file `/application/controllers/Users.php`. Since no additional segments exist, it will try to use the `index()` method in that controller.
- **/users/list** will look for the file `/application/controllers/Users.php` and try to use the controller's `list()` method.
- **/manage/users** will also work if you have a controller in a sub-directory at `/application/controllers/manage/Users.php`. In this case, it will try to run the `index()` method, since no method is listed.
- **/admin/manage/users** will work if you are even further nested in the controllers directory, looking for `/application/controllers/admin/manage/Users.php`. Once again, the `index()` method will be used.

Note: **/admin/manage/users** can also be found at `/application/controllers/admin/Manage.php` if the `admin` directory contains a `Manage` controller which has a method called `users()`. Similarly, **/manage/users** could be found at `/application/controllers/Manage.php`. Keep this in mind when using directories to organize your controllers and magic routing, as it can be fairly easy to inadvertently override an existing route if you're not careful.

Default Controller

The first of the reserved routes at the top of your `routes.php` config file, the `default_controller` setting allows you to specify a controller name which the router will attempt to use if no other match can be found for the given URI. This is commonly used to determine what will be shown when the user visits your site but doesn't supply any information for the router, like just visiting your domain at `http://mydomain.com`. By default this is set to the `Welcome` controller:

```
52  $route['default_controller'] = 'welcome';
```

So, when the user visits `http://mydomain.com`, the router will look for the file `/application/controllers/Welcome.php` and call the `index()` method.

While that's quite handy, it doesn't stop there. If you have any directory in the controllers directory that contains a controller with that name, it will also be displayed. This can be quite handy for building out the "dashboard" of an admin area, for example.

When a user visits `http://mydomain.com/admin` you could have a file at `/application/controllers/admin/Welcome.php` and the router would call its `index()` method.

404 Override

The `404_override` setting specifies a controller to be used whenever the requested controller isn't found.

Beware of this Gotcha!

The `404_override` setting only applies to a situation in which the router has identified a matching route, but it can't find the controller/method to handle the request. If a matching route isn't found and/or the `show_404()` function is called, it will not use the `404_override`. Instead, it will display the view located at `/application/views/errors/html/error_404.php`. While this often causes a great deal of confusion, it is the difference between the router's `404` *override* functionality and the framework's handling of `404` errors. At the very least, it allows you to configure a single location to add special handling for routing errors before calling `show_404()`.

Like the other routes, you set `404_override` to the name of the controller and method that you want to use. The format should be like any other route, with the controller and method names separated by a forward slash.

```
$route['404_override'] = 'errors/page_missing';
```

This will look for a controller named `Errors` and a method named `page_missing()`. It will not pass anything to the function. You can use any of the standard URI[2] methods to determine which route generated the error.

Here are a few ideas of how you might use this new controller to your benefit:

- List other possible pages the user may have wanted
- Attempt to determine whether a page has moved (if you keep track of that within the app)
- Provide a search box to search your site.
- Provide links to popular content on your site.
- Log the missing pages so the admins can try to spot problems and/or patterns.
- Try to redirect the user to a similar page (like a category page on a blog, if that can be determined). You'd want to provide them a note indicating why they wound up on that page instead of the page they requested, though.
- Check if it's a bot and end there, or, if not a bot, then show one of the other options. This could help to relieve a bit of server usage for bots following bad links.
- It could probably be used as part of a hacking-detection scheme, though I've never actually tried that.

To help design your page, consider these links for good info:

- Bing's 404 Best Practices[3]

[2] http://www.codeigniter.com/user_guide/libraries/uri.html

[3] http://www.bing.com/webmaster/help/404-pages-best-practices-1c9f53b3

- Google's 404 Best Practices[4]
- Some Creative Inspiration[5]
- Some SEO Best Practices for 404 pages[6]

Translate URI Dashes

The `translate_uri_dashes` setting simply converts dashes (-) to underscores (_). For example, the URI `articles/some-great-article` is converted to a method name compatible with PHP's naming requirements, like `articles/some_great_article`.

Why would this matter? A few years back this was a big deal as people were attempting to maximize their SEO rankings. It turned out that Google and Bing differed on how they handled them[7]. As far as I can tell, this is still an issue[8], so using dashes could help your site's SEO, while underscores could hurt it. Underscores can also be misinterpreted for spaces by users.

Turning this on will not magically make your links use dashes, though. You will still need to ensure your links are created apropriately for this to work for you.

Blocking Magic Routing

What do I mean when I say "Blocking magic routing"? Remember, "magic routing" is a term for CodeIgniter's ability to look at a URI string and match it to controllers and methods based on the URI segments. In recent years, especially since Laravel came to town, the idea of using your routes as documentation and requiring that all routes are explicitly specified, rather than determined by convention, is taking hold with many developers. CodeIgniter presently has no built-in method to disable its "magic routing" feature.

Why Block Routes?

I find that I like having all routes defined for documentation purposes whenever I work on an API. When I'm working on more traditional applications, though, I usually don't bother. Blocking can come in handy when you want to ensure that protected URIs cannot be reached in more than one way (perhaps for SEO reasons).

If you want to turn "magic routing" off, you must resort to tricking the system a little bit. Don't worry, though, it's pretty simple to do.

In order to make this work for you, you need to add the following route to the bottom of your routes config file:

[4] https://support.google.com/webmasters/answer/93641?hl=en
[5] https://econsultancy.com/blog/9525-16-creative-404-pages-to-inspire-you-to-overhaul-yours
[6] http://www.fallingupmedia.com/404-page-examples-seo/
[7] http://searchengineland.com/google-bing-handle-underscores-dashes-differently-89672
[8] http://blog.woorank.com/2013/04/underscores-in-urls-why-are-they-not-recommended/

```
$route['(.+)'] = 'something';
```

There are two parts to this solution. The first part is the regular expression within the route's key, `(.+)`. This will match any character, number, etc., which means it will match every route that you pass to the application. Then, in order to get it to send us to a 404, we provide a name which doesn't exist anywhere else on the system. It can be junk text, the name of your favorite band, or even "midi-clorians", (unless you're running a wiki on Star Wars).

Defining New Routes

Now that you know the basics of the routes file and how it's used, it's time to look at the various ways you can define new routes. We already did a quick refresher on how basic routes work, above. Now it's time to dive into a few ways to customize the working of these routes without needing to modify any framework code, or even extend the core classes.

Wildcards

Wildcards are placeholders in your routes. They may or may not be inserted into the final method at your discretion. In essence, they are simply regular expressions that are applied to your route when trying to match it against the current URI. While the user guide[9] already provides some great info on wildcards, we will show a few examples here as a quick refresher.

```
$route['product/:num'] = 'catalog/product_lookup';
```

The placeholder here is `:num`, which will match a URI segment containing only numbers. One thing to notice about this example is that the product ID that you're matching is not passed to the final destination. Instead, you'd have to use `$this->uri->segment(2)` to retrieve the product ID.

If you want to pass the product ID to your method, you need to wrap the wildcard in parentheses and use it as a back reference in the destination.

```
$route['product/(:num)'] = 'catalog/product_lookup/$1';

// In your Catalog controller:
public function product_lookup($product_id) { . . . }
```

What happens if your product IDs are not just numbers, but contain some alphabetic characters as well, like `pid1234` or `tshirt001`? Then you would use the `(:any)` wildcard. This actually matches any character except for the forward slash so that it restricts itself to a single URI segment.

[9] http://www.codeigniter.com/user_guide/general/routing.html#wildcards

```
$route['product/(:any)'] = 'catalog/product_lookup/$1';
```

You can use more than one placeholder, at any place in the URI, and match it in any order in your destination.

```
// Match /api/v1/products/ts123
$route['api/v(:num)/products/(:any)'] = 'v$1/products/$2';
```

Optional Wildcards

To really wrangle routes to your needs, you only have to remember that all route wildcards are simply regular expressions. As such, we can do a lot of crazy pattern matching here. The user guide covers a couple of good examples, but one thing not covered there which shows up in other frameworks is the optional segment. By crafting the regular expression to match 0 or more characters with the asterisk (*) we can easily create wildcards that will match whether the segment is there or not.

```
$route['product/[^/]*'] = 'catalog/product_lookup';
```

This converts our previous product lookup URI to match either with or without a product ID, like `/product` or `/product/ts123`.

This can be used with back references in the destination, as well, provided your method is defined with a default value.

```
$route['product/?([0-9]*)'] = 'catalog/product_lookup/$1';

// In your Catalog controller:
public function product_lookup($product_id = null) { . . . }
```

Slash Aware!

When passing an optional wildcard, your segment separator, the forward slash, must be identified as optional by placing a question mark after it. Otherwise, CodeIgniter won't recognize the route correctly. Incorrect: `product/([0-9]*)` Correct: `product/?([0-9]*)`

For your reference, here are the optional versions of CodeIgniter's two provided placeholders.

- `[^/]*` - optional version of `:any`
- `([^/]*)` - optional version of `(:any)`
- `[0-9]*` - option version of `:num`
- `([0-9]*)` - optional version of `(:num)`

Custom Wildcards

Once you start customizing your routes with regular expressions, you will find that your routes become pretty fragile and hard to read. After all, regular expressions weren't made for easy readability. Is there anything that you can do about this? Absolutely! You can create custom variables to hold the new custom placeholders with semantic names which make reading the routes much easier for you, your team, and whoever inherits the project.

For example, if you use a unique id in your URI instead of a user id, and this unique id can contain both letters and numbers, you can add the following line to the top of `/application/config/routes.php` and then use it in any of your routes.

```
// Define the wildcard
$uuid = '[a-zA-z0-9]*';

// Us it in your routes:
$route["users/({$uuid})"] = '/users/show/$1';
```

In this example I didn't include the parentheses in the `$uuid`, so the identifier could be used with or without back references, but you could always include the parentheses in the variable if you intend to always use back references.

```
// Define the wildcard
$uuid = '([a-zA-z0-9]*)';

// Use it in your routes:
$route["users/{$uuid}"] = '/users/show/$1';
```

When using variables as custom wildcards, remember that the string containing the wildcard must be enclosed in double-quotes, as the variable will not be expanded within single quotes. Wrapping the variable in curly braces ({ and }) is not always necessary, but doing so acts as a reminder that you're using a placeholder and also prevents maintenance headaches if someone modifies the route later in a manner which requires the braces. If you later decide to encapsulate your custom routes in a class or an array, having the braces already in place makes it easier to perform a simple search-and-replace in the routes config file.

Callbacks

PHP Requirement

This method requires PHP version 5.3 or higher.

Callbacks[10] allow you to modify the destination portion of your route dynamically at run time. While the use of these is probably pretty limited, you might find some great uses for them and, if you do, please let me know so that I can share them.

One of the most common uses that I can think of would be to create dynamic destinations to reduce your route creation. For example, if you're creating an e-commerce site and you want to be able to route to all of the different categories in your site, with a separate function in your controller for each one, you could do something like:

```
$route['category/(:any)'] = function ($category) {
        return 'categories/' . strtolower($category) . 'List';
}
```

This would take any category as the back-reference, say "shirts", and send it to the `Categories` controller, with a method named `shirtsList`. This allows you to only write a single route to handle all of the top-level categories, instead of creating 50 different routes (one for each category), or having to modify the routes if you add a new category.

Versioning Your Routes

There are times when you need to have different versions of your application. This is especially true when you have an API that is live, and you're working on the next version. You don't want to break anything that uses version 1, but you want to provide additional functionality for version 2 users. This can be easily handled with a little PHP in your routes file.

The first step is to determine the version of the API you're going to use. With the changes made to the URI class from v2 to v3, this is a really simple step. At the top of your routes file, collect the current API version (you'll need to modify this to match your route). Once you have the version, you can use that anywhere you would normally put the version number.

```
59 global $URI;
60 $version = $URI->segment(2);
61 if (empty($version))
62 {
63         $version = 'v1';
64 }
65
66 $route["{$version}/users"] = "{$version}/users/list_all";
```

Since higher routes take precedence, you could include another file that has your version-specific route overrides and new routes. To reduce memory usage and improve performance, you would want to only include the current version. This would change the previous code to something like this:

[10] http://www.codeigniter.com/user_guide/general/routing.html#callbacks

```
59 global $URI;
60 $version = $URI->segment(2);
61 if (empty($version))
62 {
63         $version = 'v1';
64 }
65
66 if ($version !== 'v1')
67 {
68         require APPPATH . "config/routes_{$version}.php";
69 }
70
71 $route["{$version}/users"] = "{$version}/users/list_all";
```

Here I don't check to see if the file exists primarily because I want it to throw an error during development so I know when I forget the file. If we are in production, the errors won't show up, and either the API will send back its error, or the application's error-handling process will take over to display a nice error page.

HTTP Verb-Based Routing

There are times when you need to respond to a request based on the HTTP verb that was used, whether it is a standard one (GET, POST, PUT, DELETE, PATCH) or a custom one (like PURGE). This is most often used when creating a RESTful API, but could be useful at other times, also.

CodeIgniter has a built in method[11] of doing this which is very simple to use.

In order to define a route that responds only to one HTTP verb, you add the verb as an array key to your route. The verb is not case-sensitive.

```
$route['products']['get'] = 'product/list_all';
$route['products/(:any)']['get'] = 'product/show/$1';
$route['products/(:any)']['put'] = 'product/update/$1';
$route['products/(:any)']['delete'] = 'product/delete/$1';
```

This makes it very easy to setup your entire API pretty quickly. If you want to speed things up even more, you can create a small helper function in your `routes.php` file to quickly create a set of standard resources for you.

[11] http://www.codeigniter.com/user_guide/general/routing.html#using-http-verbs-in-routes

```
59 function map_resource($resource)
60 {
61         echo "$route['{$resource}']['get'] = '{$resource}/list_all'\n";
62         echo "$route['{$resource}']['post'] = '{$resource}/create'\n";
63         echo "$route['{$resource}/(:any)']['get'] = '{$resource}/show/$1'\n";
64         echo "$route['{$resource}/(:any)']['put'] = '{$resource}/update/$1'\n";
65         echo "$route['{$resource}/(:any)']['delete'] = '{$resource}/delete/$1'\n";
66 }
67
68 map_resource('products');
69 map_resource('photos');
70 map_resource('users');
```

Controllers

At their core, controllers are very simple to understand, and the user guide[12] does a great job of telling you the basics. However, it doesn't talk much about the practicalities of their usage, so this chapter will focus on understanding and using controllers as you will likely use them in your applications.

No matter how deep your understanding of their inner workings are, though, knowing the best ways to organize your controllers, and distribute the methods throughout them, is essential to create a pleasant project to work in. If you have not read Chapter 1: Where Does This Go? yet, then I highly recommend you read that to get a better understanding. Especially the section on Controller Design.

CodeIgniter's Magic Instance

The first thing you need to understand about controllers is that they form the central hub of all of the "magic" that happens in your application. Whenever you call `$this->load` or `get_instance()` you are actually working with an instance of the **active controller**. This can be crucial to understand if you want to take full advantage the system, or just debug some especially problematic cases.

Whenever you call `get_instance()` it is grabbing a reference to the `CI_Controller` class that's loaded into memory. Your controllers will extend `CI_Controller`, or one of its children. This means that `get_instance()` is not providing you with some stand-alone singleton class that handles loading, etc. Nope. It means that it is returning your controller.

What ramifications does this have? Any library, model, or helper that uses `get_instance()` has full access to any public properties in the controller, and can call any public methods in the controller. Most of the time, you are going to want to avoid using the controller's properties and methods just to keep things clean. There may be times that you could take advantage of this, though. Let's take a look at a couple of them.

[12] http://www.codeigniter.com/userguide3/general/controllers.html

"Global" methods

You can use `MY_Controller` to implement some methods that can then be used in any controller, library, or model. While this is similar to loading up a helper with some common functions in it, it has the benefit of still having direct access to information in the controller that might be protected.

You might use this within an API to provide a method to get the current user's permissions, role, paired client companies, etc. The controller might verify information with the Auth module, check any paired client access, verify role, etc. during instantiation. It could then provide a method that can be used pretty much anywhere to retrieve that information.

"Global" Objects

Any class that has been loaded through `$this->load` is available within your other classes. The best use I have found for this is being able to access authentication classes. I actually discovered this by accident one time while debugging why the heck a call to get the current user's id within a model was working, even though I had never loaded the appropriate class within the model.

I do want to warn you, again, to be very careful when you decide to use these types of "global" capabilities. They can be handy, but also have the potential to cause confusion and make things harder to debug. They can also complicate the testing process dramatically. Only use this functionality after carefully weighing the pros and cons.

While you can use these features, do it rarely. It's not the cleanest method, and can cause complications down the road. It is usually better to have a function as part of a library so you can pass the dependencies (classes) in as part of the params array[13].

More than anything, this section should serve as a set of examples that you can look for in your code when things are misbehaving. Then, armed with the understanding of *why* your application is doing what it's doing, you can decide what to do about it.

Remapping Methods

Controllers can define one special method that CodeIgniter will look for to help you wield some additional control over how your controllers handle a request. If the `_remap()` method exists, it will always be called and override anything that the URI might otherwise specify. The `_remap()` method has the ability to route a request however you wish.

The parameters passed to the `_remap()` method can be very helpful in determining how the request should be resolved. The first parameter is the name of the method specified in the URI. The second parameter is an array of the remaining segments in the URI. Together, these parameters can be used to emulate CodeIgniter's default behavior, or do something completely different.

[13] http://www.codeigniter.com/userguide3/general/creating_libraries.html#passing-parameters-when-initializing-your-class

```
public function _remap($method, $params = array())
{
    if (method_exists($this, $method))
    {
        return call_user_func_array(array($this, $method), $params);
    }
        show_404();
}
```

So why use the `_remap()` method instead of routing? In many cases, it is a matter of preference, since they can both handle many of the same things, as long as the action can get to the controller. They both have their special uses, though. The router can direct to different controllers. The `_remap()` method, though, can also manipulate the parameters being passed to the class.

Here are a few example uses.

RESTful Methods

If you want to simplify your routes file, and not specify every single variation of HTTP verb and action for every controller, you could remap your methods based on the method name and HTTP verb being used.

```
public function _remap($method, $params = array())
{
        $method = strtolower($_SERVER['REQUEST_METHOD']) . '_' . $method;
    if (method_exists($this, $method))
    {
        return call_user_func_array(array($this, $method), $params);
    }
        show_404();
}
```

This would look for methods like `get_list`, `get_show`, `put_update`, etc.

Infinite Methods

Imagine you're creating an e-commerce store. You have a categories controller that you are using to display all of the products based on a category. That should be simple enough - just one method would handle all of it, right? In some cases, absolutely, but what happens when you have a few categories that need to pull or display different information, while the rest of them can happily run through the single method? In that case you're going to need new methods in the `Categories` controller for each of those special cases.

You could create a new route for each category that mapped to the new methods, but that could quickly get out of hand if you have many of them. Plus, that uses additional memory that really isn't needed. Instead, we can use the `_remap()` method to check if the category exists as a method and run that method, or run the generic `index()` method for all other categories.

```
public function _remap($category, $params = array())
{
    if (method_exists($this, $category))
    {
            return call_user_func_array(array($this, $category), $params);
    }
        $this->index($params);
}
```

MY_Controller

CodeIgniter allows you to extend any of the core classes by attaching a prefix to the classname. By default, this prefix is `MY_`, though it can be changed in `/application/config/config.php`. Using a `MY_Controller` class is the easiest way to create a custom workspace for your application. Since all of your other controllers will extend from `MY_Controller`, you can setup some utility methods to use across your site, load common classes, configure pagination, and the list goes on. I find a reason to use it for every project that I've worked on, so I make it a point to create one from the beginning so that it's ready.

```
class MY_Controller extends CI_Controller {
        public function __construct()
        {
                parent::__construct();
        }
}
```

Some of the items that are handled well in `MY_Controller` include:

- Load and setup the Cache library so that it's always available. You can use properties and/or methods to make it simple to override on a per-controller basis.
- Setup a way to turn on/off profiler, but only in development environments.
- Setup a simple theme system.
- Attempt to authorize the current user through "Remember Me" functionality, and setup a `$this->current_user` property.
- Integrate flash messages into custom JSON-rendering methods for use in AJAX.

- For API-only sites, I've added handy methods for working with Guzzle and handling oAuth authentication.
- Provide per-controller auto-load functionality for language and model files.
- Provide the ability to automatically run migrations to the latest version.
- Provide simple methods to output different content types, like text, javascript, JSON, external files, and HTML, and ensure the content type is set correctly.
- Provide methods to redirect out of AJAX calls by sending javascript to redirect the browser.

True, some of these items could be handled as libraries, but sometimes it's much simpler to provide that functionality across all controllers, where it's going to be used anyway. Once a feature reaches a decent size, though, it's best to move it to its own library for maintainability. If you are using PHP 5.4+ some of these could also work well in a Trait.

We will step through some of these items in detail later in this chapter. We will even create a simple, but very flexible, template system in Chapter 5: Showing Your Work.

Multiple Base Controllers

While `MY_Controller` is extremely handy, I often find that it makes sense to include several base controller classes to help provide different types of functionality. These different classes are used as base controllers across the site. The most common base controllers are:

- **AuthenticatedController** - Loads the authentication/authorization classes and ensures that the user is logged in.
- **AdminController** - Extends the `AuthenticatedController`, and additionally confirms that the user has administrative privileges. Frequently, this will also setup various items like the correct theme, some pagination defaults so the pagination library can have different front-end and back-end configurations, and more.
- **ThemedController** - Sets up the theme engine and provides a number of handy methods for working with the themes. Then, I'll use a trait I've created to handle auth functions so they can be used across multiple types of controllers.
- **APIController** - Provides helpful setup for an API, detects request-specific information, and provides a number of utility methods for returning data in success and failure states, and more.
- **CLIController** - Useful for making command-line-only controllers, used with CLI tools or cron jobs.

These controller examples are just ideas of common uses. Each project may benefit from very different base controllers, but they demonstrate a couple of common solutions to common problems. They might create a more enjoyable experience when developing a specific type of controller, like an API or an admin area. Additionally, they might add new functions which are available site-wide, or help with integration, like integrating theme engines.

The biggest problem then, is how to conveniently load these additional controllers. Unlike `MY_Controller`, these classes would not get auto-loaded on each request. There are a few different solutions, each with its pros and cons.

Single File Solution

The first solution is to simply include each of them in `MY_Controller.php`. It's fine, it works, and I've done it a number of times. I don't recommend it anymore, though. It's harder to maintain, and it takes up extra memory by loading classes which are not used within the current request.

```
class MY_Controller extends CI_Controller { . . . }

class AuthenticatedController extends MY_Controller { . . . }

class AdminController extends AuthenticatedController { . . . }
```

Multiple File Solution

The next step would be to move each class into its own file. This matches what CodeIgniter's style guide requires, and the current best practices across the PHP community.

The method to load the additional class which results in the best performance is to simply `require()` the file at the top of each controller which uses it. This ensures that only the required classes are loaded, and minimizes both the time it takes to instantiate classes and memory usage. However, it does mean that you need to do it on every controller. In a very performance-oriented situation, it might be worth it.

More often, though, I think simply including the different controllers from within the `MY_Controller` is the best solution, because you don't have to think about it once you set it up. That means no mistakes, and fewer errors caused by forgetting to include the class.

```
include APPPATH . 'libraries/AuthenticatedController.php';
include APPPATH . 'libraries/AdminController.php';

class MY_Controller extends CI_Controller { . . . }
```

Composer-based Solution

If you are integrating Composer into your workflow, then you already have the perfect solution at hand. Composer allows you to place the files anywhere you want to store them and load them only when needed. It solves both of the problems with the other methods discussed above.

We will go into much more detail about using Composer in later chapters, but we will quickly cover the basics of setting this up so you can get up and running with this method. It's my preferred method and extremely handy when you start using it. This does assume that you already have Composer working on your machine and are familiar with the basics. If you're not, then head over to Composer's site[14] and familiarize yourself.

[14] https://getcomposer.org/

The first thing to do is decide where to put the new controllers. I typically place them in `/application/libraries`, though it might also make sense to put them under `/application/core` so that they'll be next to `MY_Controller`.

Next, we need to tell Composer how to find them. To do this, we need to edit the `composer.json` file that CodeIgniter ships with. This file currently just holds some basic information about CodeIgniter itself, and one requirement for use by the test suite. We need to add a new section to the file, called "autoload".

```
20 "autoload": {
21         "psr-4": {
22                 "App\\": ["application/core", "application/libraries", "application/models"]
23         }
24 }
```

This sets up a new namespace[15] called "App" that you can use in your classes. When looking for class with that namespace, Composer will look in `/application/core`, `/application/libraries`, and `/application/models` for the file. You can always customize the directories it looks in to fit your situation.

If you are going to use namespaces that match up to the directories, like `App\Libraries` or `App\Models`, you can optimize the autoloader performance by as much as 35%[16] by simply being more specific with the namespace mapping. Instead of using an array of directories, we can split each one into its own listing:

```
20 "autoload": {
21         "psr-4": {
22                 "App\\Core\\": "application/core",
23                 "App\\Libraries\\": "application/libraries",
24                 "App\\Models\\": "application/models"
25         }
26 }
```

The last thing to do before we can successfully extend one of our custom controllers is to actually create the base controllers. So, create a file at `/application/libraries/BaseController.php`:

[15] http://php.net/manual/en/language.namespaces.basics.php
[16] http://mouf-php.com/optimizing-composer-autoloader-performance

```php
<?php namespace App;

class BaseController extends \CI_Controller {
        // Your custom stuff goes here
}
```

This new class extends CodeIgniter's own `CI_Controller` to ensure we still use the same `get_instance()` and don't run into any conflicts.

Finally, you just need your new controllers to extend this `BaseController`.

```php
<?php

class Welcome extends \App\BaseController {
        // Your custom stuff goes here
}
```

Any class that doesn't have a specific namespace set is considered part of the global namespace, so you do need to make sure to prefix the `App` namespace here with a backwards slash so that it can find it.

Dump Autoload

When working with Composer, you will occasionally find that it can't find a class, even though you know you have everything setup correctly. The first step to debug is always to jump to the command line in your project root, and have Composer rebuild its autoload mapping files, with `composer dump-autoload`. This will fix the problem most of the time. When it doesn't, look back at your code, because you probably have something wrong, like a missing backwards slash.

Want More?

Wait! There's more of this chapter than what you've seen here, but only in the full product. The rest of this chapter deals out some powerful tweaks to your controller that can make your day-to-day experience coding your applications quite a bit more enjoyable.

I hope you've enjoyed what you've seen so far and will jump right on over to https://leanpub.com/practicalcodeigniter3[17] and buy the full book now to take your CodeIgniter 3 knowledge to the next level. Or at least consider it.

Either way - thanks for reading this far. See you in the next pages!

[17] https://leanpub.com/practicalcodeigniter3

4. Showing Your Work

Now that you have total control of how the application handles the user's request, it's time to look at how you present information to the user.

View Basics

Code which controls how CodeIgniter presents information to the browser is usually stored in view files[1]. Views are simply PHP files that contain the HTML you want to display to your user.

They are stored (by default) in the `/application/views` directory.

Views are loaded with the `$this->load->view()` command. The first parameter is the name of the view to display.

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

This displays the view located at `/application/views/welcome_message.php`.

You could easily store that view under a sub-directory for each part of the application, allowing you to split out admin views from app views, or views used by different controllers. Just add the name of the sub-directory to the name of the view.

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

This would display the view located at `/application/views/admin/welcome_message.php`.

View Data

You can make data available to use within the view by passing it in as the second parameter. This should be an array of key/value pairs.

[1] http://www.codeigniter.com/user_guide/general/views.html

```
$data = array(
        'user_name' => $username
);
$this->load->view('welcome_message', $data);
```

In the view, you would access this data as a variable named after the `key` of the array.

```
<?= $user_name ?>
```

If you pass an array or an object in as one of the values, you have full access to that object or array within the view.

```
// In the controller:
$user = array(
        'first_name' => 'Darth',
        'last_name' => 'Vader'
);
$this->load->view('profiles/basic_info', array('user' => $user));

// In the view:
<h2>Welcome back, <?= $user['first_name'] .' '. $user['last_name'] ?>!</h2>
```

To keep things organized, and easier to use in a mixed AJAX/non-AJAX request-based controller, you might find it works best to split out parts of your data into different class methods, or into libraries. This makes it simple to pull the information together in the controller.

```
$data = array(
        'user' => $this->auth_lib->current_user(),
        'profile' => $this->collect_profile_info($this->auth_lib->id())
);
$this->load->view('profiles/basic_info', $data);
```

There are times, though, when you need to add items to the view from locations other than your controller, or maybe you want to make some variables available from within your controller's constructor. No problem. You can add data to be displayed in the view from any place that has access to the controller instance with the `$this->load->vars()` method.

```
$data = array(
        'current_user' => $this->auth->current_user(),
        'user_theme' => $this->user_model->get_user_theme()
);
$this->load->vars($data);
```

This is similar to using the second parameter of the `$this->load->view()` method, but can be called multiple times from different locations before loading the view. If keys in the data passed to either `$this->load->vars()` or `$this->load->view()` match, the later value(s) will replace the previous value(s).

Constructing Pages

What you want to display to the user, though, is typically not just a single view. There will be common elements, like page headers, footers, and sidebars, that need to wrap around the content. There are a few ways that you'll see mentioned in the forums, but I've never found any of them to be robust enough for my needs in a world where clients like to change their minds all too often.

I will go over them here, because for some very simple sites they may be all that's needed. With each method I go over, you'll notice things get abstracted more and more until it becomes a simple theme system that works with you to keep your code organized in a way that can be understood by anyone on the team.

Method 1: In Method

The first method I saw recommended years ago, which is fine for small brochure-type sites, is to load all of the views in sequence:

```
public function index()
{
        // Make all of the data available for all views...
        $this->load->vars($this->data);

        $this->load->view('header');
        $this->load->view('site_navigation');
        $this->load->view('the_actual_page_content');
        $this->load->view('footer');
}
```

While this works, it becomes tedious if you have to do that in every controller method which needs to display something to the user.

Method 2: In View

The next most common method I've seen used in a couple of different CMSs, is to load views at the top and bottom of each view. This allows several other views to be grouped in one file, like a header file that calls the HTML head view, a view that contains the page header, and the main navigation, and perhaps even the sidebar. Then a second `footer` view might include the page footer, another view to collect and output JavaScript, etc.

```
// In the controller
$this->load->view('the_contents');

// In "the_contents" view
$this->load->view('theme/header_collection');

page content goes here...

$this->load->view('theme/footer_collection');
```

This reduces a lot of our calls by combining the different elements into collections of templated content which surrounds our page content. Then, if anything needs to change in the template, you can simply alter those template views and your entire site is updated. For many small sites, this can work just fine. However, once the number of pages you need to display grows, this becomes a bit burdensome.

There's a better way, based around some conventions that help keep your code organized, and give you the ability to swap out themes at will, even using different themes per controller, or method, all while reducing the amount of work you have to do while building the site. This is a simplified version of what I tend to use now, and is very close to a version I've used on quite a few different sites. That's what this next section is all about.

Want More?

Wait! There's more of this chapter than what you've seen here, but only in the full product The rest of this chapter builds a simple theme system, explaining every step and showing you all of the code. The ThemedController file even gets bundled with the code samples that ship with the book.

I hope you've enjoyed what you've seen so far and will jump right on over to https://leanpub.com/practicalcodeigniter3[2] and buy the full book now to take your CodeIgniter 3 knowledge to the next level. Or at least consider it.

Either way - thanks for reading this far. See you in the next pages!

[2] https://leanpub.com/practicalcodeigniter3