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.

The Book At A Glance

You should have received the code samples as part of the purchase. If you did not, or cannot find the samples that you've downloaded, the larger code samples from this book can be downloaded from my site.

The latest version will always be available at: http://newmythmedia.com/files/PracticalCI3Code.zip[4]

[4] http://newmythmedia.com/files/PracticalCI3Code.zip

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.

Controller Tweaks

In this section of the book, I'm going to share some common, helpful methods that I've used over the years to make the workflow a bit smoother and more enjoyable. Not everything will be something that you will want in your projects, but they should, at the very least, get your ideas flowing about how you could enhance your `MY_Controller` file.

Flash Messages

CodeIgniter's implementation of flashdata is very helpful for status messages. I use them all of the time. However, it was created with a flow that assumed you would have a separate method handle

any form submittals that would redirect to other methods on success or failure, reloading the page in the process. I prefer not to hit that additional page load and, instead simply display the results at once, if possible. That wasn't possible with CodeIgniter prior to version 3 because of the way the sessions worked. In 3, you can set flashdata and immediately access it in the view without a problem, but I still use this method because of the additional features it provides.

```
public function set_message($message = '', $type = 'info')
{
    if ( ! empty($message))
    {
        if (isset($this->session))
        {
            $this->session->set_flashdata('message', $type . '::' . $message);
        }

        $this->message = array(
            'type' => $type,
            'message' => $message
        );
    }
}
```

This accepts the message to show the user, as well as a message `type`. The type is used for CSS classes to affect the displayed status message. It checks if the session has been initialized and, if so, saves the information to the session. It also stores a copy in a class var. While this part is not technically necessary due to the mentioned changes in the session library, it is a nice backup in case a controller is running that doesn't have the session initialized for some reason, for example, if it was called from the CLI.

Along with this, there is a method to retrieve the status message for you.

```
public function message()
{
    $return = array(
        'message' => null,
        'type' => null
    );

    // Does session data exist?
    if (empty($message) && class_exists('CI_Session', false))
    {
        $message = $this->session->flashdata('message');

```

```
13          if ( ! empty($message))
14          {
15              // Split out our message parts
16              $temp_message = explode('::', $message);
17              $return['type'] = $temp_message[0];
18              $return['message'] = $temp_message[1];
19
20              unset($temp_message);
21          }
22      }
23
24      // If message is empty, we need to check our own storage.
25      if (empty($message))
26      {
27          if (empty($this->message['message']))
28          {
29              return '';
30          }
31
32          $return = $this->message;
33      }
34
35      // Clear our session data so we don't get extra messages on rare occasions.
36      if (class_exists('CI_Session'))
37      {
38          $this->session->set_flashdata('message', '');
39      }
40
41      return $return;
42  }
```

This simply grabs the data from wherever it might be, either the session or a class variable, and returns an array with two values, `message` and `type`.

These methods provide a very handy way of working with status messages. You can pass them to the views directly, or, even better, create a new view that will display the message with the correct HTML, and call that from your views. This allows you to have multiple display types for it by simply calling different views.

Global Profiler Control

CodeIgniter's profiler can be very handy during development, but it's not something that you want to have show up on a production site. Additionally, you don't want to have to turn it on or off within

your current code every time you want to use it. For those reasons, I find it handy to control this from a single config setting.

In my applications, I tend to keep my app-specific code in a file at `/application/config/application.php` and set it to be auto-loaded so those settings are always available. However, you could place the setting in `config.php` or any other config file that you have easy access to.

First, create the setting in the config file so we have something to reference.

```
$config['show_profiler'] = TRUE;
```

Next, we'll add the following code to the `MY_Controller` class' constructor:

```
1  if ($this->config->item('show_profiler') === TRUE)
2  {
3          $this->output->enable_profiler(TRUE);
4  }
```

By checking the config setting, this can be easily changed per environment by just copying the file to the correct folder, and changing the setting. Set it and forget it.

This is a tiny convenience method, but I find it is the addition of these multiple tiny conveniences that add up to something pleasant and efficient to work with. Isn't that what we all want?

Automatic Migrations

CodeIgniter's migrations are extremely handy, but they're not really setup to be easy to use without building a controller to hit from the command line or the browser to make it all work. Instead of going through all of that work, I find that most of the time all that I really want to do is run the migrations to the latest version so that I know I'm always up to date. This is especially helpful when you are working in teams as it keeps everyone in sync.

Whether you want to run this in production or not is very much an application-specific issue that you should discuss with your team. As a default, I would recommend turning this feature OFF in production environments to keep things safe, and have another way to trigger it, either by SSH-ing into the server, or a key-protected method you can tap through the browser. This way, you can have the convenience of automatic migrations in development, while ensuring that your production environment is only running migrations you have explicitly triggered.

Again, you need a setting to control this feature in one of your config files.

```
$config['auto_migrate'] = TRUE;
```

Back in `MY_Controller`:

```
if ($this->config->item('auto_migrate') === TRUE)
{
        $this->load->library('migration');
        $this->migration->latest();
}
```

Now, any time a page is accessed, the system will verify that the database schema is at the latest point. If not, it will run the migrations to update your database.

Cache By Default

In order to really optimize your applications, you will often need to cache some HTML fragments, or database results, etc. However, if that's not easy to use and there for you, ready to go, you are less likely to want to bake that into the application by default.

A good solution is to ensure that the cache driver is always up and running. However, to keep you from hitting the cache during development, you would want to set it up so that it was very easy to change in environments. Config settings to the rescue, once again!

```
$config['cache_type'] = 'dummy';
$config['backup_cache_type'] = 'dummy';
```

This provides a one-stop place to change the cache drivers to use. However, there might be times when you have one specific controller that needs to use a different cache engine. In that case, we also need a way to override the cache driver(s) on a per-controller basis in `MY_Controller`. Add the following properties at the top of the class:

```
/**
 * The type of caching to use. The default values are
 * set globally in the environment's config files, but
 * these will override if they are set.
 */
protected $cache_type = NULL;
protected $backup_cache = NULL;
```

Finally, in `MY_Controller` we need to check the type of cache engine to use, and load the appropriate drivers.

```
// If the controller doesn't override cache type, grab the values from
// the defaults set in the config file.
if (empty($this->cache_type))
{
        $this->cache_type = $this->config->item('cache_type');
}
if (empty($this->backup_cache))
{
        $this->backup_cache = $this->config->item('backup_cache_type');
}

// Make sure that caching is ALWAYS available throughout the app
// though it defaults to 'dummy' which won't actually cache.
$this->load->driver('cache', array('adapter' => $this->cache_type, 'backup' => $\
this->backup_cache));
```

That's all it takes. Now, whenever you think you might need to cache something in the future for performance, you know that you are always prepared and can simply use it and not worry about loading it. If performance starts to become an issue, you can change the drivers to something other than the `dummy` driver in one controller, or in all of them, and test to see if it helps you at all.

Outputting Plain Text

This is the first of several "rendering" methods that I keep handy in `MY_Controller`. These methods don't form part of a template engine. Instead, they are simply utility methods to output different content types. Some, like this, work best during AJAX calls, or in web services, where the client is expecting something other than proper HTML.

```
public function renderText($text, $typography = FALSE)
{
        // Note that we don't do any cleaning of the text
        // and leave that up to the client to take care of.

        // However, we can auto_typography the text if we're asked nicely.
        if ($typography === TRUE)
        {
                $this->load->helper('typography');
                $text = auto_typography($text);
        }

        $this->output->enable_profiler(FALSE)
                     ->set_content_type('text/plain')
```

```
15                      ->set_output($text);
16  }
```

Besides just spitting out text, this turns off the profiler to ensure no extra content is returned. It also sets the correct content type header and outputs the text through the Output library's `set_output()` function.

Additionally, it can use CodeIgniter's `auto_typography` helper to enhance the text a little and make it a little nicer, though it's off by default.

Rendering JSON Data

When building APIs you will often find yourself working with just JSON data. While a bigger API might require you to to output data in multiple formats, requiring something more robust, this simple method can provide a simple way to work with JSON data.

```
1   public function renderJSON($json)
2   {
3           // Resources are one of the few things that the json
4           // encoder will refuse to handle.
5           if (is_resource($json))
6           {
7                   throw new \RuntimeException('Unable to encode and output the JSON data.');
8           }
9           $this->output->enable_profiler(FALSE)
10                      ->set_content_type('application/json')
11                      ->set_output(json_encode($json));
12  }
```

All it accepts is the data that you want to ouput. This can be any type of item that `json_encode()` can handle, but will typically be either an array or object. The profiler is turned off, the content type set appropriately, and the output is set to the encoded JSON.

Grabbing JSON Data

Going along with the previous method, this method creates a convenient way to get the data from the `php://input` stream and format it in JSON, ready for you to work with.

```
public function get_json($format = 'object', $depth = 512)
{
        $as_array = ($format === 'array');

        return json_decode($this->input->raw_input_stream, $as_array, $depth);
}
```

The first parameter allows you to define the type of element returned, either an object or an array. The second parameter can typically be left to its default, unless you find yourself working with very large requests. See the json_decode docs[17] if you run into problems with that.

This technique could also be used to create custom methods to work with values from `PUT`, `PATCH`, and other HTTP requests.

Realtime Responses

By default, CodeIgniter will use output buffering to collect all of the output and send it to the browser all at once. Most of the time, this is fine. However, there will be times when you want to see the results of the script as they happen, not be forced to wait until the end of script execution. This could be for a very long-running data import script, or even just for a plugin update script, like WordPress, where you want to see the status of each element as it goes so that the user knows something is actually happening.

To do this, you need to empty the buffer and turn it off. I've used this simple method to handle this in the past.

```
public function render_realtime()
{
        while (ob_get_level() > 0)
    {
        ob_end_flush();
    }
    ob_implicit_flush(TRUE);
}
```

AJAX Redirects

The `redirect()` function in CodeIgniter is only intended for standard, HTTP headers-based redirects. This means that it doesn't work in the middle of an AJAX call. In order to break out, you need to send some javascript to change the window location. This method does just that for you.

[17] http://php.net/manual/en/function.json-decode.php

```
public function ajax_redirect($location = '')
{
    $location = empty($location) ? '/' : $location;

    if (strpos($location, '/') !== 0 || strpos($location, '://') !== FALSE)
    {
        if ( ! function_exists('site_url'))
        {
            $this->load->helper('url');
        }

        $location = site_url($location);
    }

        $script = "window.location='{$location}';";
```

$this->output->enable_profiler(FALSE)

```
                    ->set_content_type('application/x-javascript')
                    ->set_output($script);
}
```

While this is a sampling of common methods I find myself using on quite a few projects, my primary hope is that they will inspire you to take the time and really start to create a workflow that works for you, and helps keep development fresh.

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.

A Simple Theme System

In this section we're going to walk through creating a very simple theme system which you can use in all of your projects. It's small enough to be easily customized to your projects' needs. It will focus on a few conventions to keep your files organized and easy to find for everyone in the team. You should view this as a starting point, meant to be tweaked, modified, and expanded to fit your team and workflow needs. However, I've used variations of this simple system on several fairly large projects and can't imagine working without something like this in place.

What To Expect

The first task is to decide what features this system needs to provide for us. This engine doesn't necessarily handle working with modules well, but a couple of `str_replace()` calls can make it work just fine.

For this example, the theme system functionality will be added to our `MY_Controller` file. If you have access to PHP version 5.4 or higher, you should consider turning this into a `trait` that you can easily add to any other controller, freeing up `MY_Controller` for other, more general, uses.

Here's a list of the design decisions that went into this engine:

Organization

Keep views organized by mapping the view structure to the controller/method names. We should allow overriding of the view name, but by defaulting to a simple convention, everyone on the team can readily find the view they are looking for (and we only have to hard-code the view name for exceptions). Granted, this requires that some level of sane organization is in place at the controller level, too.

Here are some examples, with the controller/method names on the left and the view file name listed on the right, relative to the `/application/views` directory.

Users::index()	users/index.php
Users::by_role()	users/by_role.php
admin/Roles::manage()	admin/roles/manage.php

The Theme

In this example, we are only going to support a single theme for the entire site. Often, this is enough. Especially if you design as much admin interaction into the front-end of the site as possible. This could be very easily expanded to work with multiple themes with very little effort.

We are going to store the theme within the `/application/views` directory, in a sub-directory called, simply, `theme`.

We should support multiple layouts, though. This allows you to easily create page structures specific to certain areas of your site. Or to simply allow for common configurations, like a single-column layout, a two-column layout (sidebar on the left), and yet another two-column layout (sidebar on the right).

We can use views within the theme folder to hold common sections that are shared between the different layouts. No special tool needs to be created for this. We can simply use CodeIgniter's built-in `$this->load->view()` to handle it.

Assets

Stylesheets and JavaScript files should be handled in the simplest way. Many teams use LESS or SASS compilers to create a single CSS file anyway, so there's no need for us to recreate some tools that would never perform as well as a pre-processor tool. Instead, we'll simply collect an array of files and loop over them in the theme files.

Rendering A View

The first portion that we'll work on is the heart of the system, the `render()` method. This is the method you would call from any of your controller functions to display the page, wrapped up in your theme's layout. This does require a few properties to be added to the class to keep track of assets and allow us to customize which layout or view file should be used.

Class variables

```
28      protected $use_view = '';
29
30      protected $use_layout = '';
31
32      protected $external_scripts = array();
33
34      protected $stylesheets = array();
35
36      // Stores data/variables to be sent to the view.
37      protected $vars = array();
```

The `$use_view` and `$use_layout` properties are used to override the conventions we have in place, in case we need to select a different view, or use a layout file other than `index`.

The `$external_scripts` and `$stylesheets` arrays hold the names of the CSS and Javascript files that we need to attach to our page. They will be passed along with the other data to the view, where they will be looped over to create the links.

The `$vars` array allows us to collect the variables to pass to the view, instead of collecting it in a `$data` as we go along. This also allows variables to be added from other methods in your controller very simply.

The render() method

```
136     protected function render($data = array())
137     {
138         // Generate view name based on current method/controller
139         $dir = $this->router->directory;
140
141         $view = empty($this->use_view) ?
142                 $dir . $this->router->class . '/' . $this->router->method :
143             $this->use_view;
144
145         // Merge any saved vars into the data
146         $data = array_merge($data, $this->vars);
```

```
147
148         // Make sure any scripts/stylesheets are available to the view
149         $data['external_scripts'] = $this->external_scripts;
150         $data['stylesheets']      = $this->stylesheets;
151
152         // Build our notices from the theme's view file.
153         $data['notice'] = $this->load->view('theme/notice', array('notice' => $t\
154 his->message()), TRUE);
155
156         // We'll make the view content available to the template.
157         $data['view_content'] = $this->load->view($view, $data, TRUE);
158
159         // Render our layout and we're done!
160         $layout = empty($this->use_layout) ? 'index' : $this->use_layout;
161
162         $this->load->view('theme/' . $layout, $data);
163
164         // Reset our custom view attributes.
165         $this->use_view = $this->use_layout = '';
166     }
```

The first thing that we do in the render method is to determine the correct view file to render. If the controller has filled in the `$use_view` property, which is typically empty, then we use that view name. If not, then we build the view name based on the directory the controller is in, relative to the `/application/views` directory, followed by the controller name and the method name.

On line 146 we merge the `$data` passed into the method with any data that may have been set elsewhere. This provides the controller with a full set of values to send to the view to display to the user.

Next, we collect any CSS and Javascript files and make them available for the views so they can be looped over and have link tags created.

The `notice` variable is the formatted "flash message" that is set with the `set_message()` method we discussed in the previous chapter. This loads a separate view within the `views/theme` folder that allows the message to be displayed in whatever manner works best for the CSS in play. For Bootstrap or Foundation, these are known as alerts.

Then, on line 156 we actually collect the method's specific view, already loaded, with all of our data passed to it. We collect it here, instead of loading it within the view for a couple of reasons. First is that it allows you to make modifications to variables that are accessed later in the layout file. This isn't needed very often, but occasionally it's handy to be able to modify things in this manner. Second, it allows you to implement caching for the layout files if you need to later down the road. Once we collect this view, we add it to the `$data` array where it will be passed onto the layout file.

Next, we check to see if the controller is overriding the layout file. If it is, then we use the file they specified. If not, then we default to the index file.

Finally, we use the standard `$this->load->view()` process to load up the layout file.

And that is the bulk of this simple theme system. It's very simple to use and to modify to meet your needs. There are a few small utility methods that we will build out to make it more flexible before leaving you to tweak it to your needs.

Setting View and Layout at Runtime

While a controller-level override is nice, there are quite a few times when you need to override the layout or the view that should be used within a single method. We will add these next two methods to make that process a little more elegant than simply setting the properties.

The view() method

```
192     /**
193      * Specifies a custom view file to be used during the render() method.
194      * Intended to be used as a chainable 'scope' method prioer to calling
195      * the render method.
196      *
197      * Examples:
198      *      $this->view('my_view')->render();
199      *      $this->view('users/login')->render();
200      *
201      * @param  string $view The relative path/name of the view file to use.
202      * @return MY_Controller instance
203      */
204     public function view($view)
205     {
206         $this->use_view = $view;
207 
208         return $this;
209     }
```

The `view()` method allows you to set the view to use, and ignore the conventions built into the system. We are simply setting the `$use_view` property and returning the class instance so that it can be used in a chainable fashion.

The layout() method

```
213     /**
214      * Specifies a custom layout file to be used during the render() method.
215      * Intended to be used as a chainable 'scope' method prioer to calling
216      * the render method.
217      *
218      * Examples:
219      *      $this->layout('two_left')->render();
220      *
221      * @param  string $view The relative path/name of the view file to use.
222      * @return MY_Controller instance
223      */
224     public function layout($view)
225     {
226         $this->use_layout = $view;
227 
228         return $this;
229     }
```

This method works the same way as the view method, but overrides the layout that will be used instead. It also returns a class instance for chaining.

Asset Methods

To make it simple to add any stylesheets and Javascript files during runtime, we need these next two methods.

The add_script() method

```
457     /**
458      * Adds an external javascript file to the 'external_scripts' array.
459      *
460      * @param string $filename
461      * @return $this
462      */
463     public function add_script($filename)
464     {
465         if (strpos($filename, 'http') === FALSE)
466         {
467             $filename = base_url() . 'assets/js/' . $filename;
468         }
```

```
469
470         $this->external_scripts[] = $filename;
471
472         return $this;
473     }
```

The `add_script()` method allows you to add an external Javascript file to have a link created for. If the file passed in is a full URL, we don't touch it. Otherwise, we assume it's a local file and uses the `base_url()` helper method to build a full URL for the file. You might need to modify the location for your assets here if you are going to store them anywhere other than `/assets/js`.

This requires that the `url` helper has already been loaded. That is one of the few files that I will put in the autoload file so it's always available because I find that `site_url()`, `base_url()` and `url_title()` methods are invaluable and used across the site.

Notice this only handles external javascript, not inline scripts. If you need inline scripts, you can pass them as view data and echo them out as part of the layout (after the other javascript files have been loaded), or in the method's view.

```
1  if ( ! empty($external_scripts) && is_array($external_scripts))
2  {
3      foreach ($external_scripts as $script)
4      {
5          echo "<script src='{$script}.js'></script>";
6      }
7  }
```

In your view you would use something like above to create the script tags toward the end of your layout.

The add_style() method

```
477     /**
478      * Adds an external stylesheet file to the 'stylesheets' array.
479      *
480      * @param string $filename
481      * @return $this
482      */
483     public function add_style($filename)
484     {
485         if (strpos($filename, 'http') === FALSE)
486         {
487             $filename = base_url() . 'assets/css/' . $filename;
```

```
488         }
489
490         $this->stylesheets[] = $filename;
491
492         return $this;
493     }
```

This function works identically to the `add_script` method, except that it collects the CSS file links to use within the head of our layout files.

There it is! A simple, but fairly feature complete theme system that you can run with. A few things you might want to consider adding to it are listed below.

- Automatically escape any view data before passing it to the view. You can use the `xss_clean()` helper function for many occasions. Zend Framework has an excellent context-aware cleaner[2] that you might consider using. It's very thorough and full of community-proofed best practices. If you do that, though, you should consider adding it to the `set_var()` method, where you can specify the type of escaping to do.
- Collect a series of meta tags to make SEO of your pages a bit simpler.
- Use the parser to keep PHP out of your view files. We will look at integrating this in the next section.
- Allow different themes to be used and easily specified, much like we did with the `$use_view` and `$use_layout` properties.

I am sure there are many other areas that could be expanded, but that should give you some ideas to start working on.

Included with the purchase of this ebook is a complete `MY_Controller` file with this system, and the controller tweaks discussed previously, ready for you to use.

Parsing View Output

Now that you can display the content, you might find that your company needs to separate out the views for the designers to work on. Many companies prefer to use some form of template tags to keep their designers away from the PHP. CodeIgniter has the Template Parser[3] class to help you out here, though it is quite basic.

It allows you to have template tags that are substituted in your views without the need to use PHP.

[2] http://framework.zend.com/manual/current/en/modules/zend.escaper.introduction.html
[3] http://www.codeigniter.com/user_guide/libraries/parser.html

```
<html>
    <head>
            <title>{blog_title}</title>
    </head>
    <body>
            <h3>{blog_heading}</h3>

        {blog_entries}
                <h5>{title}</h5>
        <p>{body}</p>
        {/blog_entries}

    </body>
</html>
```

The items in curly braces will be replaced by view data that you send along in the `parse()` command.

```
$this->load->library('parser');

$data = array(
    'blog_title'   => 'My Blog Title',
    'blog_heading' => 'My Blog Heading',
    'blog_entries' => array(
                array('title' => 'Title 1', 'body' => 'Body 1'),
                array('title' => 'Title 2', 'body' => 'Body 2'),
                array('title' => 'Title 3', 'body' => 'Body 3'),
                array('title' => 'Title 4', 'body' => 'Body 4'),
                array('title' => 'Title 5', 'body' => 'Body 5')
    )
);

$this->parser->parse('blog_template', $data);
```

When you have opening and closing tags, like `{blog_entries}` and `{/blog_entries}`, the parser will loop through an array of items, applying the enclosed markup to each member of the array.

The `parse()` method is identical to the `$this->load->view()` method. The first parameter is the name of the view file to load. The second is an array of key/value data pairs. The third parameter, if `TRUE`, will return the resulting output, instead of displaying it. It even calls the `load->view()` method in the background and then parses the results with its `parse_string()` method.

You can easily incorporate this into the `render()` method in the simple theme system by replacing any uses of `load->view()` with `parser->parse()`.

The user guide provides lots of additional techniques for using the Parser so please refer to it if you're interested.

Integrating the Twig Template Engine

For some companies, the provided tools for creating themes is not flexible enough. They will then turn to existing template engines like Smarty or Twig. The problem becomes how to integrate it easily and elegantly into your workflow. In this section we will step through integrating the Twig[4] template library.

Installing Twig

For this tutorial, we will assume that you're not using Composer and install Twig the "traditional CodeIgniter way."

1. Download the most recent version[5] of the library.
2. Unzip the files and copy them to the `/application/third_party` directory in its own `twig` sub-directory, to keep it separate from any other libraries you download.

That's all it takes to install. However, we don't have it actually working with the application, yet. To do that, we need to create a library to wrap the Twig class for us, setting up the default configuration, and providing access to Twig's functions.

Integrating Twig

We will handle the integration with a new library. Create a file at `/application/libraries/Twig.php`. Here's the code for you. We will discuss what it does afterwards.

The Twig Library

```
1  <?php
2
3  require_once APPPATH . 'third_party/Twig/lib/Twig/Autoloader.php';
4
5  class Twig {
6
7      /**
8       * An instance of the Twig library.
9       * @var
```

[4] http://twig.sensiolabs.org/

[5] https://github.com/twigphp/Twig/tags

```
10       */
11      protected $instance;
12
13      /**
14       * CodeIgniter instance
15       * @var
16       */
17      protected $ci;
18
19      //--------------------------------------------------------------------
20
21      /**
22       * Sets up our Twig instance with configuration.
23       */
24      public function __construct()
25      {
26          $this->ci =& get_instance();
27
28          Twig_Autoloader::register();
29
30          $config = array(
31              'debug'                 => FALSE,
32              'charset'               => 'utf-8',
33              'base_template_class'   => 'Twig_Template',
34              'cache'                 => config_item('cache_path') ? config_item('\
35  cache_path') : APPPATH . 'cache',
36              'auto_reload'           => TRUE,
37              'strict_variables'      => FALSE,
38              'autoescape'            => TRUE,
39              'optimizations'         => -1
40          );
41
42          $loader = new Twig_Loader_Filesystem(APPPATH . 'views');
43
44          $this->instance = new Twig_Environment($loader, $config);
45
46          // Add our extensions.
47          $this->extend();
48      }
49
50      //--------------------------------------------------------------------
51
```

```
52      /**
53       * Attempts to call a method on the Twig instance. If it works,
54       * will append the output to any existing output.
55       *
56       * @param $method
57       * @param $args
58       */
59      public function __call($method, $args)
60      {
61          if ( ! method_exists($this->instance, $method))
62          {
63              throw new \BadMethodCallException("Undefined method '{$method}' atte\
64  mpt in the Twig class.");
65          }
66
67          $this->ci->output->append_output(call_user_func_array(array($this->insta\
68  nce, $method), $args));
69      }
70
71      //--------------------------------------------------------------------
72
73      /**
74       * Allows you to set an extensions you want to set.
75       */
76      protected function extend()
77      {
78          $this->ci->load->helper('url');
79
80          // base_url
81          $tag = new Twig_SimpleFunction('base_url', 'base_url');
82          $this->instance->addFunction($tag);
83
84          // site_url
85          $tag = new Twig_SimpleFunction('site_url', 'site_url');
86          $this->instance->addFunction($tag);
87      }
88
89      //--------------------------------------------------------------------
90  }
```

We have two properties: one to store a link to the CodeIgniter instance, and another to store our Twig instance.

__construct()

In the constructor, we load Twig's Autoloader so it can find the other Twig files. Next, we setup our configuration array. These are all of the options that can be passed to the Twig_Environment to customize how it works. Most of these are at the default values, except for the cache directory, which we set to whatever the application's cache directory is. You could create a config file to store these settings outside of the class, and simply use `config_item()` calls to get the values.

Next, we get the required instances of the loader and the environment, and save them so they can be used later.

Finally, we call our library's `extend()` method, which just acts as a convenient place to add in any custom methods. In this example, the `base_url()` and `site_url()` methods are made available to the templates.

__call()

We have implemented the `__call()` method so that any method calls against this library are simply passed on to the saved instance of `Twig_Environment`. This makes the class act as if it is the instance so you can call any Twig command directly on this library.

The output is passed on to the Output library, where it is appended to whatever other content is already there.

Using the Twig Library

Now that we have created the library, we can start using it in our controllers. I won't go over any best practices on using and organizing Twig templates here, but it is a very flexible solution with lots of users online that can provide assistance if you need it.

```
1  class SomeController extends CI_Controller {
2
3          public function __construct()
4          {
5                  parent::__construct();
6
7                  $this->load->library('twig');
8          }
9
10         public function index()
11         {
12                 $data = array(
13                         'user' => $user
14                 );
```

```
15
16                  $this->twig->render('users/profile.php', $data);
17          }
18  }
```

In your controllers, you need to make sure that you've loaded the library. In this example, this is handled in the constructor, though creating a new ThemedController or something similar might be a better solution, since you could customize your workflow around Twig.

Then, in your methods, you can simply call Twig's `render()` method, passing the name of the view as the first parameter. This must be relative to `/application/views`, and must contain the file extension. If you're using Twig, you probably want to keep any PHP out of the templates, so you don't need to use PHP files for your view files. If you want to, you can use simple .html files, instead.

The second parameter is an array of key/value pairs to make available to the views, just like you would with a `load->view()` call.

From here, you can use any of Twig's features, including layout inheritance, other template files, data filters, and more.

A Simple Widget System

Earlier in the book, I mentioned that HMVC was often used as a system for creating "widgets" that can be reused throughout your application. I mentioned that I didn't think that was the best solution. This section goes over how you can easily create your own widget implementation for use in your own projects.

Overview

What we want to accomplish with this library is to create a system that allows you to call a method in any library or fully namespaced class that can be found by your autoloader(s). If you are using Composer, this is expanded to even more classes.

We want to be able to pass in a number of named parameters so that we can customize the output, for example, if we want to change the number of blog posts returned to 5 instead of the default of 10. The easiest way to do this is a simple string with the key/value pairs separated by a space. This allows us to freely pass in the parameters in any order we want, keeping things simple in the views.

Finally, we want the system to allow the widget to be cached so that we can really crank up the performance of our application by not having to use any potentially expensive database calls to display the information.

Best of all, we can do all of this with a single method in its own library file.

The Solution

Here is the entire class. I will discuss the code below. This should be saved as `/application/libraries/Widgets.php`.

The Widgets Library

```
<?php

class Widgets {

    /**
     * Runs a callback method and returns the contents to the view, allowing
     * you to create re-usable, cacheable "widgets" for your views.
     *
     * Example:
     *     Widgets::show('blog/posts:list', 'limit=5 sort=publish_on dir=desc');
     *
     * @param string $command
     * @param string $params
     * @param int $cache_time               // Number of MINUTES to cache output
     * @param string $cache_name
     * @return mixed|void
     */
    public static function show($command, $params = NULL, $cache_time = 0, $cach\
e_name = NULL)
    {
        // Users should be allowed to customize the cache name
        // so they can account for user role, logged in status,
        // or simply be able to easily clear the cache items elsewhere.
        if (empty($cache_name))
        {
            $cache_name = 'theme_call_' . md5($command . $params);
        }

        if ( ! $output = $this->ci->cache->get($cache_name))
        {
            list($class, $method) = explode(':', $command);

            // Since $params is a string, we need to split it into
            // an array of 'key=value' segments
            $parts = explode($params);

```

```
37              $params = array();
38
39              // Prepare our parameter list to send to the callback
40              // by splitting $parts on equal signs.
41              foreach ($parts as $part)
42              {
43                  $p = explode('=', $part);
44
45                  if (empty($p[0]) || empty($p[1]))
46                  {
47                      continue;
48                  }
49
50                  $params[$p[0]] = $p[1];
51              }
52
53              // Let PHP try to autoload it through any available autoloaders
54              // (including Composer and user's custom autoloaders). If we
55              // don't find it, then assume it's a CI library that we can reach.
56              if (class_exists($class))
57              {
58                  $class = new $class();
59              }
60              else
61              {
62                  get_instance()->load->library($class);
63                  $class =& get_instance()->$class;
64              }
65
66              if ( ! method_exists($class, $method)) {
67                  throw new \RuntimeException("Unable to display the Widget at {$c\
68  lass}::{$method}");
69              }
70
71              // Call the class with our parameters
72              $output = $class->{$method}($params);
73
74              // Cache it
75              if ((int)$cache_time > 0)
76              {
77                  get_instance()->cache->save($cache_name, $output, (int)$cache_ti\
78  me * 60);
```

```
79                  }
80              }
81
82              return $output;
83          }
84
85          //--------------------------------------------------------------------
86      }
```

The `show()` method takes 4 arguments: `$command`, the name of the class/method; a string, `$params`, that contains the key/value pairs; `$cache_time`, the number of minutes to cache the results (defaults to `0`); and `$cache_name`, the name to use when referencing the cached results. If `$cache_time` is 0, then no caching will happen. If `$cache_time` is above `0` then it will cache it for that number of **minutes**. This requires that the cache library has already been loaded, otherwise it will throw an error. If no value is passed in for `$cache_name`, then we generate one.

We then try to read the info from the cache. If we find it, we simply return it, otherwise we go through the rest of the process.

On line 30, we split `$command` into its two parts, the class name and the method to call. For this class, I've used a single colon to separate them, simply to make it clear that it wasn't calling a static method, though you are free to change that if you find yourself wanting to type the double colons.

As an example, the `$command` might be passed in as `blog/posts:list_all`. The script would break that into a class name of `blog/posts` with a method of `list_all()`. In this case, this is a standard CodeIgniter library that could be found at `/application/libraries/blog/Posts.php`. If you are using Composer, or have some other autoloader running, you could use a namespaced class by passing the full class name here, like `\App\Blog\Posts:list_all`. Then, this class could be loaded from anywhere on the server, if your autoloader can find it. The code to load the class and check whether the method exists is found in lines 55-67.

Starting at line 32, we split up the string of parameters into key/value pairs. We loop through each of these, splitting them on the equals sign to find the key and value. We collect this into an array, reusing the `$params` variable since it is no longer needed once split into `$parts`.

This allows you to pass a list of parameters, named in whatever way would make the most sense to your designers as they work on the view. An array of named parameters is passed to your library method, where you can verify their existence and use them as needed. Continuing the example of the recent blog posts above, this might be `"limit=5 sort=publish_on dir=desc"`. Once handled by the script, it would be passed to your library like so:

```
array (
        'limit'        => 5,
        'sort'        => 'publish_on',
        'dir'                => 'desc'
)
```

We then try to call the class/method, passing the `$params` array to it on line 70. If this doesn't work out, PHP will tell us where we messed up so we can fix it.

Finally, we cache the information for the number of minutes that were previously specified. Again, the cache library must already be loaded and configured for this to work.

All of the output is then returned to the view.

That's it, a very simple method that provides all of the flexibility you need to implement a widgets system that is easy for your designers to use, without the potential conflicts that might exist with the HMVC controller-calling-controller system. The only restriction our library methods have is that they accept an array of parameters, and that they must return the output as a string.

Using The System

To use the system in a project, there are just a couple of steps. We will step through a sample implementation of the blog posts example from above just to make sure that everything is clear.

The Library

The first thing you need is a library to call. Create a new library at `/application/libraries/Blog_-widgets.php`. Inside that file, we need to implement the callable method. This method will take only one parameter: the array of named parameters.

```
<?php

class Blog_widgets {
        public function list_all(array $params = array())
        {
                $ci =& get_instance();

                $limit = empty($params['show']) ? 0 : (int)$params['show'];
                $offset = empty($params['offset']) ? 0 : (int)$params['offset'];
                $order_by = empty($params['sort']) ? 'publish_on' : $params['sort'];
                $order_dir = empty($params['dir']) ? 'desc' : $params['dir'];

                $ci->load->model('posts_model');
```

```
14
15                  $posts = $ci->posts_model->limit($limit, $offset)
16                                           ->where('deleted', 0)
17                                           ->order_by($order_by, $order_dir)
18                                           ->find_all();
19
20                  return $ci->load->view('blog/recent_posts', array('posts' => $posts), true);
21          }
22  }
```

This simply sets our limit and sorting values based on what the user might have selected, or some default values. Then it loads the posts from our `posts_model`, and returns the output of a view that loops over the posts to display them as a list of links.

That's all we need for our recent posts widget, so we can display it in the sidebar, or at the bottom of single posts, or wherever we may otherwise need it.

Loading the Widget Library

The Widgets class must be available whenever you are displaying your views so we need to ensure that it is loaded. You have a few ways to load the Widgets class. The first is to load it in the controller methods that need it. This can get a bit tedious and error prone, though.

```
$this->load->library('widgets');
```

If you're using a `MY_Controller` or `Themed_Controller` class to handle your theming, then a better solution is to load it as part of that class' constructor or `render()` method. If you're using `Twig` or another library, you will have to explore its documentation to ensure that it can be accessed globally, for example through template extensions.

Finally, you can add the library to the `/application/config/autoload.php` file so that it is loaded for every request. This might be tempting, but is a waste of resources on any script calls that don't require the template, like API calls, command-line calls, generating RSS feeds, AJAX calls, etc. I definitely discourage this unless every call on your site will be displaying a page.

```
$autoload['libraries'] = array('widgets');
```

Display the Widget

Now that the library has been created, and the `Widgets` library is loaded, we simply need to make a call in one of our views. For example, we will assume that we are displaying it in the sidebar.

```
<div class="sidebar">
    <?= Widgets::show('Blog_widgets:list_all', 'show=5 sort=publish_on', 60); ?>
</div>
```

That's all there is to it.

5. Working With Data

The data layer in CodeIgniter is intended to be handled through model files. These files are typically used to get your data into and out of the database. Often, these are used on a 1-to-1 basis where each model represents a single table in the database. This often works, but when your data gets more complex with many-to-many relationships, pivot tables, and more, this becomes difficult to stick to. It still makes sense to try, though.

By representing each table with a model (as often as possible) you can use and customize a `MY_Model` file to provide your own common set of methods, and your own workflow, on top of the Query Builder. We will look at this process and create our own simple `MY_Model` class later in this chapter.

First, though, let's look at how you get the system ready to actually communicate with the database.

Basic Setup

Before you can use your database, you have to configure the framework to communicate with the database by telling it what type of database you are using (MySQL, PostgreSQL, SQLite, etc.), which driver to use (mysqli, pdo, etc.), and a few other settings. CodeIgniter only provides support for relational databases, and does not provide drivers for document or key/value storage systems like Redis or MongoDB.

All configuration information is setup within the `/application/config/database.php` configuration file. In this file you can define one or more connections, along with failover connections for each. The user guide covers them pretty well, but I would like to touch on some points that I don't think are clear enough.

Drivers

If the user guide provides a list of which drivers CodeIgniter currently supports, I'm not aware of it. The following table lists the available drivers and the type of connection they need, since some require a dsn string for connection, while others can suffice with the settings. It should be noted that some drivers will attempt to create the dsn string for you from the config fields, but that behavior cannot always be relied upon.

Driver	Connection
Cubrid[1]	dsn
Firebird/Interbase[2]	config
MS SQL[3][4]	config
MySQL[5][6]	config
MySQLi[7]	config
Oracle (oci8)[8]	dsn
ODBC[9]	dsn
PDO [10]	dsn
PostgreSQL[11]	dsn
SQLite[12][13]	config
SQLite 3[14]	config
Microsoft SQL Server[15]	config

In addition, the PDO driver, which is an abstraction layer that provides a consistent way to interact with multiple databases, can be setup to use the following sub-drivers:

Driver	Sub-Driver name
4d[16]	4d
Cubrid[17]	cubrid
FreeTDS[18][19]	dblib
Firebird[20]	firebird
IBM DB2[21]	ibm
Informix[22]	informix
MySQL[23]	mysql
Oracle (oci)[24]	oci

[1] http://php.net/manual/en/intro.cubrid.php
[2] http://php.net/manual/en/intro.ibase.php
[3] http://php.net/manual/en/intro.mssql.php
[4] These drivers are deprecated and/or removed from PHP and should be avoided.
[5] http://php.net/manual/en/intro.mysql.php
[6] These drivers are deprecated and/or removed from PHP and should be avoided.
[7] http://php.net/manual/en/intro.mysqli.php
[8] http://php.net/manual/en/intro.oci8.php
[9] http://php.net/manual/en/intro.uodbc.php
[10] http://php.net/manual/en/intro.pdo.php
[11] http://php.net/manual/en/intro.pgsql.php
[12] http://php.net/manual/en/intro.sqlite.php
[13] This driver works for SQLite version 2 or lower only.
[14] http://php.net/manual/en/intro.sqlite3.php
[15] http://php.net/manual/en/intro.sqlsrv.php
[16] http://php.net/manual/en/ref.pdo-4d.php
[17] http://php.net/manual/en/ref.pdo-cubrid.php
[18] http://php.net/manual/en/ref.pdo-dblib.php
[19] This driver provides access to Microsoft SQL Server and Sybase databases through the FreeTDS library.
[20] http://php.net/manual/en/ref.pdo-firebird.php
[21] http://php.net/manual/en/ref.pdo-ibm.php
[22] http://php.net/manual/en/ref.pdo-informix.php
[23] http://php.net/manual/en/ref.pdo-mysql.php
[24] http://php.net/manual/en/ref.pdo-oci.php

Driver	Sub-Driver name
ODBC[25]	odbc
PostgreSQL[26]	pgsql
SQLite[27]	sqlite
Microsoft SQL Server[28]	sqlsrv

Connecting Through PDO

One area that the User Guide is downright missing, is much detail on how to connect to your database through PDO. At least, it wasn't clear to me when I first started looking into it. It turns out that I was looking in the wrong place, and it is all described in nice detail in the PHP manual[29], instead of the CodeIgniter User Guide. While the PDO driver provided in CodeIgniter v2.x didn't do much more than provide the connection, the PDO setup in v3 is actually full-featured enough that it's worth looking into and seeing if it meets your needs, since it has full dbforge and query builder support.

So, how do we connect? The magic is in the dsn string. You must prefix it with the name of the sub-driver you wish to use, followed by a single colon, followed by the remainder of the dsn string. For most drivers, you can provide many of the options in the `$options` array, but the host and dbname will typically need to be specified in the dsn string.

If you want to connect to MySQL through PDO, you would do something like:

```
$db['default']['dsn'] = 'mysql:host=localhost;dbname=database_name';
```

Should You Use PDO?

In plain PHP, the primary advantages of using PDO are a nicer, OOP experience using the class, prepared statements that can help boost your security, and a few other things that don't seem to get talked about as much. However, CodeIgniter provides its own abstractions on top of the PHP abstractions so many of those specific features are lost. From what I can tell, prepared statements are not currently implemented in CodeIgniter's PDO drivers, though I have heard talk that it might show up in a future version.

There's no reason *not* to use the PDO drivers. However, there's no real advantage to be gained from using the PDO drivers, either, except to prepare for the future. According to the PHP manual, there are still a few advantages[30] to using the **mysqli** driver, if that's your alternative. If you are considering using PDO over one of the other, similar drivers, your best bet is to to check the PHP manual and look for any specific limitations of that driver, if any. All things being equal, use either one. It won't matter too much, since CodeIgniter abstracts away the differences in most cases, anyway.

[25] http://php.net/manual/en/ref.pdo-odbc.php
[26] http://php.net/manual/en/ref.pdo-pgsql.php
[27] http://php.net/manual/en/ref.pdo-sqlite.php
[28] http://php.net/manual/en/ref.pdo-sqlsrv.php
[29] http://php.net/manual/en/pdo.connections.php
[30] http://php.net/manual/en/mysqlinfo.api.choosing.php

stricton

This setting turns on the Strict mode of your database. The exact behavior of this may vary from database to database, but will typically affect how invalid or missing values are handled, and sometimes even what is considered an invalid result.

Values can be considered invalid for a number of reasons, including having the wrong data type for the column, the value could be out of range, or considered missing, if no value is given for a non-null column without an explicit default value.

This setting should be set to `TRUE` during development to help ensure you catch as many errors as possible, but can be turned off for production environments if you wish, since it would potentially throw fewer errors at that point.

db_debug

When this value is set to `TRUE`, which is the default value for all environments except production, the database driver will collect the query strings and performance statistics of each query made. This information is made available to the profiler to be able to show you what queries ran and how long they took. This is especially handy when using the Query Builder to construct your queries for you. You can also use `$this->db->last_query()` to retrieve the last query that ran while debugging an application during development.

When the value is `FALSE`, it will not store these values and uses a fair amount less memory because of it. If you find yourself running very large queries and hitting memory errors, this might be your problem. Though you should probably restructure your queries to use fewer resources, as well.

This should follow the default settings: `TRUE` for development, `FALSE` for production.

Persistant Connections

One of the options in the configuration array is whether or not to use a persistent connection. By default, this is set to a `FALSE` value. This is likely the best solution for many of the sites that we do. You should know when and why to use them, though.

Like the name implies, persistent connections are connections to the database that are not closed when your script is done running. This *can* be good for performance reasons, as the time it takes to connect to the database might be relatively high. However, many times it will actually lead to worse performance, like when you are running multiple applications against a single database server, or when you have a low number of concurrent connections allowed on the server. Here are a few examples of when you may and may not want to use them:

Use When:

- You have a high cost to connect to your database server, like when it's running on a different server than the application

- You have one application on the server that accesses the database often
- There are few applications/users accessing the database at the same time.

Don't Use When:

- You need to use transactions
- You need to store sessions in the database
- You are running PHP in CGI mode (which does not include running under fastcgi for nginx)
- Your application only needs to access the database infrequently

These are simply some guidelines for its use, and how appropriate it is for your application is completely unique to your situation, how tuned your database server is, etc. You should start off with them turned OFF, then, if you start having performance issues, try using it (if it seems to meet the above "rules"), and measure the server's performance for a while to see if it was an improvement or made things worse.

Basic Queries

Once you've connected to your database, you need to be able to get data from it. There are a lot of methods in the query builder[31] class which allow you to build queries programmatically. There are too many to go over here, so I'll leave the complete descriptions to the excellent guide. There are a few points we should cover here, though, to keep your queries safe.

query()

The `query()` method provides a generic way to send raw SQL to the database and get a result set back. This is the simplest way to query your database and, at times, the most powerful.

Don't Fear Raw Queries

Using Raw Queries only has two real drawbacks: 1) it's not cross-platform, so if you move to a different database, you'll likely have to tweak your queries manually, and 2) you're responsible for escaping the data yourself to ensure it stays safe. Escaping the data is simple, though, and we'll cover that here. As for the cross-platform needs, though query builder helps make it possible to write cross-platform SQL, it doesn't prevent you from writing SQL which is not compatible with other databases, and true cross-platform SQL is not necessarily easy to write using query builder alone. For most projects, you know up front which database technology you or the client wants to use, and it's used for the life of the project. If it comes down to changing database vendors, it is probably just one part of a bigger set of issues which you are trying to solve.

[31] http://www.codeigniter.com/user_guide/database/query_builder.html

```
$this->db->query("SELECT id, username, email FROM users WHERE users.id=?", array\
($user_id));
```

This query simply pulls a few bits of information from the `users` table for a single user. There is one thing to note about this query, and that's the question mark. The question mark acts as a placeholder for a value. In the second parameter of the `query()` function you can provide an array of values that will be inserted into those placeholders. The values inserted into the placeholders are automatically escaped for you, creating safer queries. The above query will generate the following SQL:

```
SELECT id, username, email FROM users WHERE users.id=14;
```

If the query has more than one placeholder, then they will be replaced in order.

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? AND author = ?";
$this->db->query($sql, array(3, 'live', 'Rick'));

// Produces this query:
SELECT * FROM some_table WHERE id = 3 AND status = 'live' AND author = 'Rick';
```

And, finally, if one of the values is an array, it will be converted to use the `IN` syntax where possible.

```
$sql = "SELECT * FROM some_table WHERE id IN ? AND status = ? AND author = ?";
$this->db->query($sql, array(array(3, 6), 'live', 'Rick'));

// Produces this query:
SELECT * FROM some_table WHERE id IN (3,6) AND status = 'live' AND author = 'Ric\
k';
```

While you can manually escape your data with the `escape()` methods, I don't recommend it. Always use parameter binding when working with raw queries.

Query Builder

As mentioned earlier, the query builder class allows you to construct your queries programmatically using a fluent, object-oriented interface. It is database agnostic, and the method names are very easy to remember, even if you don't always remember the SQL syntax. As with parameter binding, all values are automatically escaped for safer queries. They look something like this:

```
$query = $this->db->where_in('id', array(3,6))
        ->where('status', 'live')
        ->where('author', 'Rick')
        ->get('some_table');
```

The database library provides a very large number of methods that cover almost any common SQL query you'll need to write. Please read about them in the user guide.

Should you use the query builder or stick with raw SQL? That's largely a matter for personal opinion. I will typically use the query builder, but do find myself slipping into raw queries for more complicated queries that I've had to build in my database utilities in order to work through the logic. I find that for more complex queries, especially if they involve any sub-queries, it is easier to follow the logic and debug as raw SQL instead of a series of function calls.

One thing I do love about query builder, though, is that it makes it possible to set portions of your query in different functions, or different parts of a function. I find this invaluable when I need to work with a large number of options that might come in through the URL, like pagination, multiple filters, etc. Then I can keep all of the setup of filters by itself, where it can be used by several methods, but can be modified in one spot only.

Keeping Your Queries DRY

```
1  <?php
2
3  class DRY_model extends CI_Model {
4
5      public function get_articles($filters, $where)
6      {
7          $this->set_filters($filters);
8
9          $query = $this->db->get('articles');
10
11         return $query->result_array();
12     }
13
14     //--------------------------------------------------------------------
15
16     public function get_posts($filters, $where)
17     {
18         $this->set_filters($filters);
19
20         $query = $this->db->get('posts');
21
22         return $query->result_array();
```

```
23      }
24
25      //--------------------------------------------------------------------
26      //
27      protected function set_filters($filters)
28      {
29          // Paging
30          if (array_key_exists('page', $filters))
31          {
32              $this->db->limit($filters['page'], $filters['offset']);
33          }
34
35          // Status
36          if (array_key_exists('status', $filters))
37          {
38              $this->db->where('status', $filters['status']);
39          }
40      }
41
42      //--------------------------------------------------------------------
43
44  }
```

At times, though, this can cause its own set of issues. The most common one I've encountered is when multiple queries have to happen, like when I need to grab a set of categories to use as part of my main query. If I do the `set_filters()` call *before* I perform the query to get the categories, the database engine will try to apply the filters to the categories query, often resulting in an error because columns I was filtering on don't exist in the categories table. It's worse, though, when it doesn't throw an error and you have to track down why incorrect results are coming back. The solution is simple, luckily. Move the `set_filters()` call *after* the other queries to ensure they get applied to the primary query.

Basic Query Results

Now that you've made the query, it's time to get some results from it. There are four methods that you'll use most of the time, so that's what we will cover here. All of these methods work with the result of a query.

result()

This method brings back an array of objects, with each object representing one row of results from the database.

```
$query = $this->db->get('posts');

$results = $query->result();

// Returns:
array(
        stdClass {
                ...
        },
        stdClass {
                ...
        }
);
```

This function is actually a wrapper for the `result_array()`, `result_object()`, and `custom_result_object()` functions. By default, this will create the results as objects, but you can pass in the word `'array'` as the only parameter if you want the results returned as arrays.

```
$results = $query->result('array');
```

Likewise, you can enter a class name and have the results returned as an array of objects of that class. We will explore this in more detail later.

If the query did not generate any results, then `$results` will be an empty array. While there is a function, `$query->num_rows()` that exists in the query result object, it is considered a best practice not to use that function and rely on the return values instead. This is because the `num_rows()` method is very inefficient in some of the database drivers, since they don't support that feature directly, so the entire result set must be loaded into memory and counted in those drivers to determine that number.

result_array()

This is the same thing as calling `result('array')`.

row()

Much like `result()`, this method can be used to return an object, array, or custom class. However, this only retrieves the first row from the result set. Each consecutive call will return the next row of the result set.

```
$result = $query->row();
```

If no results are found, `$result` will be `NULL`.

If you don't want the first row of results, you can specify the exact index to use by passing the desired index as the first parameter. The index is zero-based.

```
$fifth = $query->row(4);
```

row_array()

Identical to `row()` but will return the results as an array instead of an object.

Handling Large Query Results

One limitation that the database library has is the potentially high amount of memory it can use on large result sets. This is partially due to the driver retrieving all of the results and storing them in the query result object when you request them. For very large query results, you can quickly run out of memory. I have had a couple of projects where, as part of a cron job, we had to run through every row in the table and perform some calculations. Some of these tables had several million rows of data, way too much to store in memory.

One solution is to loop over all of the results, breaking the table down into 100 or 200 rows at a time. While you can go larger based on your memory, it's easier to maintain an accurate AJAX progress bar if you do it in smaller chunks.

unbuffered_row()

While breaking up the query definitely works, the database driver now provides the `unbuffered_row()` method. This method is similar to the `row()` method, but does not read the entire result set into memory. Instead, it only keeps the current row in memory. This makes it very efficient to use when processing very large data sets.

```
$query = $this->db->query("SELECT * FROM my_table");

while ($row = $query->unbuffered_row())
{
        // Process the row here
}
```

reconnect()

When you're processing large data sets like this, it is possible to lose the connection to your database due to the server's idle timeout, max request times in MySQL, etc. Often, you can use the `reconnect()` method as part of your loop to ensure that your connection remains available.

```
$count = 0;

while ($row = $query->unbuffered_row())
{
        // Process the row here

        // Reconnect every 1000 records
        if ($count % 1000 == 0)
        {
                $this->db->reconnect();
        }
        $count++;
}
```

Custom Object Results

One of the most under-utilized capabilities of the database engine, from what I've seen in forums and on the web, is the ability to use a custom class as the object returned by `result()` and `row()` methods, or by using the dedicated `custom_result_object()` and `custom_row_object()` methods. As many years as I have been using CodeIgniter, I've only recently discovered it, as the 2.x documentation only mentioned it briefly and did nothing to highlight it.

This gives you the ability to easily keep your code clean and separated. Let's walk through a small example using an imaginary blog engine.

For this blog, we have a `Post` class that represents a single blog post. By keeping it in a separate class, we have a single location where we can modify how it formats the content, dates, and anything else directly relating to a single blog post. This class only acts as a representation of the post, and should not provide any functionality for saving it to, or getting it from, the database. That will come next.

Post class

```
1  <?php
2
3  class Post {
4
5      protected $id;
6      protected $title;
7      protected $status;
8      protected $published_on;
9      protected $author_id;
10     protected $body;
11
12     //--------------------------------------------------------------------
```

```
13
14      public function published_on($format)
15      {
16          return $this->published_on->format($format);
17      }
18
19      //--------------------------------------------------------------------
20
21      public function body($word_limit = 0)
22      {
23          // Format the body
24          get_instance()->load->helper('typography');
25          get_instance()->load->helper('text');
26
27          if ($word_limit > 0)
28          {
29              $body = word_limit($body, $word_limit);
30          }
31
32          $body = auto_typography($this->body);
33
34          return $body;
35      }
36
37      //--------------------------------------------------------------------
38
39      public function __set($name, $value)
40      {
41          if ( ! property_exists($this, $name))
42          {
43              throw new InvalidArgumentException("'{$name}' is not a part of the P\
44  ost class.");
45          }
46
47          if ($name == 'published_on')
48          {
49              $this->published_on = DateTime::createFromFormat('U', $value);
50          }
51
52          $this->$name = $value;
53      }
54
```

```
55      //--------------------------------------------------------------------
56
57      public function __get($name)
58      {
59          if ( ! property_exists($this, $name))
60          {
61              throw new InvalidArgumentException("'{$name}' is not a part of the P\
62  ost class.");
63          }
64
65          return $this->name;
66      }
67
68      //--------------------------------------------------------------------
69
70  }
```

The basics of the class function through the `__set()` magic method. This allows the `result()` method to simply set the properties on the class, yet still allows us to do special formatting for any objects that need it. In this example, the `published_on` property is turned into a DateTime object when it is set.

With the `__get()` magic method, we get raw access to any of the class properties if we need it. We can create custom methods for any cases where we need special formatting. In this example, we've created a `published_on()` method that takes a format string and returns the date in the specified format. The `body()` method uses the `auto_typography()` helper function to clean up the plain text body string and convert it into HTML, and even accepts a parameter to limit the number of words returned, so it can be used as part of a post teaser on overview pages.

How do we use this class? This is for our model to handle. Here's a quick example of a `find()` method that retrieves a single row from the database as an object of class Post.

Finding A Post

```
1   <?php
2
3   class Post_model extends CI_Model {
4
5       public function __construct()
6       {
7           parent::__construct();
8
9           // Make sure the Post class exists in memory.
10          $this->load->library('Post');
```

```
    }

    //--------------------------------------------------------------------

    public function find($id)
    {
        if (empty($id) || ! is_numeric($id))
        {
            throw new InvalidArgumentException('The ID passed to the find method\
 must be an Integer.');
        }

        $query = $this->db->where('id', $id)
                          ->get('posts');

        $row = $query->row(0, 'Post');

        return $row;
    }

    //--------------------------------------------------------------------
```

The constructor ensures that the `Post` class is loaded into memory, assuming it is stored at `/application/libraries/Post.php`. In the `find()` method, we first verify that we have a potentially valid `$id` and, if not, throw an exception, telling us we did something wrong in our code so we can quickly track it down. Next, we grab the results from the database, and use the `row()` method to create an instance of the `Post` class that is loaded up with our results from the database and ready to use. If it didn't find a matching class, it would return `NULL`.

When it comes to saving the result to the database, you do have to do a little bit more work, but it's worth it.

Saving Posts

```
    public function save(Post $post)
    {
        if ( ! empty($post->id))
        {
            return $this->update_post($post);
        }

        return $this->create_post($post);
    }
```

```
10
11      //--------------------------------------------------------------------
12
13      public function create_post(Post $post)
14      {
15          $data = $this->prepare_data($post);
16
17          if ( ! $this->db->insert($data))
18          {
19              return FALSE;
20          }
21
22          return $this->db->insert_id();
23      }
24
25      //--------------------------------------------------------------------
26
27      public function update_post(Post $post)
28      {
29          $data = $this->prepare_data($post);
30
31          return $this->db->update('posts', $data, array('id' => $post->id));
32      }
33
34      //--------------------------------------------------------------------
35
36      protected function prepare_data(Post $post)
37      {
38          return array(
39              'title'         => $post->title,
40              'status'        => $post->status,
41              'published_on'  => $post->published_on('U'),
42              'author_id'     => (int)$post->author_id,
43              'body'          => $post->body
44          );
45      }
46
47      //--------------------------------------------------------------------
```

Here we have a single main entry point to either create or update a post in the `save()` method. This takes a `Post` object and determines whether to insert or update based on the existence of an ID. The logic has been separated out into the `create_post()` and `insert_post()` methods, so you can still

use them individually if necessary. They both use the `prepare_data()` method, which construct an array that can be used during both the `insert()` and `update()` methods, and ensures that all of the data is in the proper format.

Both the `insert()` and `update()` methods of the database layer will automatically escape the data before putting it in the database. This example does not include any form of data validation, but that is something that you must do in a real application and will be covered later in this chapter.

This leaves one final piece of the puzzle to deal with, and that is how we get the `Post` created in the controller to send to the `Post_model`.

```
1  public function create()
2  {
3          // The post_model's constructor will load our Post class.
4          $this->load->model('post_model');
5
6          $post = new Post();
7
8          $post->title = $this->input->post('title');
9          $post->status = $this->input->post('status');
10         $post->author_id = $this->input->post('author_id');
11         $post->body = $this->input->post('body');
12
13         $this->post_model->save($post);
14 }
```

First, we load the model, which we'll need later to save the post to the database. This also ensures that the `Post` class is loaded and ready to use, since it is loaded by the model's constructor. Next, we create an instance of the `Post` class, then populate all of the properties we can based on `$_POST` data from the submitted form. Finally, we use the `Post_model` to save the data to the database. This example is missing logic for handling errors during the save process, validation errors, etc., and is only intended to display how easy it is to work with classes in this way.

The ability to auto-populate custom objects with data from the database in the model goes a long way towards encouraging us to write cleaner, more object-oriented code.

Customizing MY_Model

Like `My_Controller`, the `MY_Model` file allows you to customize the way you interact with your data source. While models do not have to retrieve information from a database, that is one of their more common functions. You could, however, create a `MY_Model` that allowed you to use the file system for storage, or connect to Amazon Web Services for file retrieval, or almost anything else you desire. In

this section, we are going to focus on connecting to a database, since that will be the most common use.

When you extend the `CI_Model` class with a `MY_Model` class, you have the opportunity to simplify working with the database. Common uses that I've seen in different extensions often include one or more of the following capabilities:

- Provide simple CRUD functionality (Create, Read, Update and Delete)
- Allow "hooks" into the process to allow extending functionality easily
- Provide in-model validation of data so you don't have to do it in your controllers every time
- Protect certain fields from being changed
- Restrict updating to only a few pre-defined fields
- Automatically track creation and modification times
- Allowing you to easily work with "soft deletes" where only a flag is set in the record to indicate it's deleted.
- Provide simple relationship management features for other records, i.e. one-to-many, one-to-one, etc.
- Provide custom methods for database interaction with new methods like `first()`, `increment`, and more.

Creating your own base model with all of those features can be a little bit of work. Luckily, there are a number of good ones out there if you search. Many of them share common features, and some attempt to provide more solutions than others, so it's worth looking at them all for inspiration, or if you don't want to customize your own. Here are a few to get your search started:

- CodeIgniter-MY_Model[32] by Avenirer
- codeigniter-base-model[33] by Jamie Rumbelow, one of the first that I saw.
- CodeIgniter Base Model[34] by Chris Schmitz
- My own CIDbModel[35] from Sprint[36]

Using Composer?

If you're using Composer for class autoloading in your application, you don't actually have to write a `MY_Model` extension. Instead, you can simply write a new class that contains the magic `__get()` method to be able to pull from the controller instance, as that's all that `CI_Model` does, anyway.

[32] https://github.com/avenirer/CodeIgniter-MY_Model
[33] https://github.com/jamierumbelow/codeigniter-base-model
[34] https://github.com/ccschmitz/CodeIgniter-Base-Model
[35] https://github.com/ci-bonfire/Sprint/blob/develop/myth/Models/CIDbModel.php
[36] http://sprintphp.com

Custom CRUD

In this section, we are going to demonstrate how simple it can be to create a good, solid starting point for a `MY_Model` that you can then take and expand as you wish. The goal of this is to provide simple features for creating new records, updating existing records, retrieving one or more records as an array, object, or custom class, and having access to all of CodeIgniter's Query Builder functionality.

After the complete code below, I will describe how everything works so you can go off and start customizing your own version.

Custom MY_Model

```php
<?php

class MY_Model extends CI_Model {

    protected $table_name;

    protected $primary_key = 'id';

    /**
     * If TRUE, delete methods will not delete the row,
     * but will simply set the 'deleted' column to '1'.
     * @var bool
     */
    protected $soft_deletes = FALSE;

    /**
     * Specifies the way that individual rows
     * are returned to the user. Valid values
     * are: 'array', 'object', or the name of a class,
     * including namespace, if applicable.
     */
    protected $return_type = 'array';
    protected $temp_return_type = NULL;

    //--------------------------------------------------------------------

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

        $this->temp_return_type = $this->return_type;
```

```
32      }
33
34      //--------------------------------------------------------------------
35
36      /**
37       * Retrieves a single row from the database by primary key.
38       *
39       * @param mixed $id
40       * @return mixed
41       */
42      public function find($id)
43      {
44          $query = $this->where($this->primary_key, $id)
45                        ->get($this->table_name);
46
47          $row = $this->temp_return_type == 'array' ?
48                      $query->row_array() :
49                      $query->row(0, $this->temp_return_type);
50
51          $this->temp_return_type = $this->return_type;
52
53          return $row;
54      }
55
56      //--------------------------------------------------------------------
57
58      /**
59       * Returns all records. Intended to be modified with calls to db library,
60       * like:
61       *     $users = $this->user_model->where('active', 1)
62       *                               ->limit(10, 0)
63       *                               ->find_all();
64       */
65      public function find_all()
66      {
67          $query = $this->get($this->table_name);
68
69          $rows = $this->temp_return_type == 'array' ?
70                      $query->result_array() :
71                      $query->result($this->temp_return_type);
72
73          $this->temp_return_type = $this->return_type;
```

```
74
75          return $rows;
76      }
77
78      //--------------------------------------------------------------------
79
80      /**
81       * Inserts a new row into the database.
82       *
83       * @param array $data
84       * @return int
85       */
86      public function insert(array $data)
87      {
88          $result = $this->db->insert($this->table_name, $data);
89
90          return $result ? $this->db->insert_id() : FALSE;
91      }
92
93      //--------------------------------------------------------------------
94
95      /**
96       * Updates a single row based on primary key.
97       *
98       * @param mixed $id
99       * @param array $data
100      *
101      * @return bool
102      */
103     public function update($id, array $data)
104     {
105         return $this->db->where($this->primary_key, $id)
106                         ->update($this->table_name, $data);
107     }
108
109     //--------------------------------------------------------------------
110
111     /**
112      * Deletes a single row by primary key. If 'soft_deletes' are TRUE,
113      * will simply change the 'deleted' column to a value of '1'.
114      *
115      * @param mixed $id
```

```
116         * @return bool
117         */
118        public function delete($id)
119        {
120            if ($this->soft_deletes)
121            {
122                return $this->update($id, array('deleted' => 1));
123            }
124
125            return $this->db->where($this->primary_key, $id)
126                            ->delete($this->table_name);
127        }
128
129        //--------------------------------------------------------------------
130
131        /**
132         * Allows you to set the type that each object is returned as, either
133         * 'array', 'object', or a custom class name, with namespace if applicable.
134         *
135         * @param string $type
136         * @return $this
137         */
138        public function return_as($type)
139        {
140            $this->return_type = $type;
141
142            return $this;
143        }
144
145        //--------------------------------------------------------------------
146
147        /**
148         * Allow access to the database functions directly without us needing
149         * to copy every method over to this class.
150         *
151         * @param string $name
152         * @param array  $arguments
153         * @return $this
154         */
155        public function __call($name, array $arguments = NULL)
156        {
157            if (method_exists($this->db, $name))
```

```
158         {
159             $result = call_user_func_array(array($this->db, $name), $arguments);
160
161             if ($result instanceof CI_DB_mysqli_driver)
162             {
163                 return $result;
164             }
165
166             return $this;
167         }
168     }
169
170     //--------------------------------------------------------------------
171
172 }
```

Class Properties

The following class properties are needed to make things a breeze for us:

- **$table_name** This is simply the name of your database table. It is used by most of the methods to be able to pull data from the table without your child classes having to modify anything.
- **$primary_key** This is the column name of your primary key. If your primary key has multiple columns, this won't work for you, and you'll have to change the methods to work with the primary keys.
- **$soft_deletes** If `TRUE`, this requires a column named `deleted` to exist in this table, and the `delete()` method will set the value to `1` to signify that the record has been deleted. If `FALSE`, then the `delete()` method will delete the record from the table as usual.
- **$return_type** This specifies how results should be returned, whether as an array, a simple object, or a custom class. This works hand in hand with **$temp_return_type** to determine which format is returned.

Finding Items

Two methods have been provided to make grabbing items in any of the allowed formats simple.

find()

The `find()` method retrieves a single record based on its primary key. Like the rest of the methods, it has been kept pretty simple for this demonstration. First, it performs a query against the database where the primary key matches the provided `$id`. It then checks the return format and generates

the correct type of result object. Next, it resets the `$temp_return_type` to match the `$return_type` so we get the right type of result object on the next call. Finally, it returns the resulting row. If no row was found, it returns `NULL`.

```
$user = $this->user_model->find($id);
```

find_all()

This method retrieves all results in the table, unless modified beforehand. It's intended to be used with the other Query Builder methods, in place of the `get()` method. It works just like the `find()` method, except that it returns multiple results, instead of just one.

```
$users = $this->user_model->where('deleted', 0)
                          ->order_by('email', 'asc')
                          ->limit(25,0)
                          ->find_all();
```

insert()

The `insert()` method takes an array of key/value pairs and creates a new record in the database. The only thing it really does differently from the Query Builder's `insert()` method is that it automatically uses the model's `$table_name` property as the name of the table in which the data will be inserted, and returns the primary key of the new row upon success.

```
$data = array(
        'username' => 'madmax',
        'email' => 'madmax107@thunderdome.com'
);

$user_id = $this->user_model->insert($data);
```

update()

The `update()` method is here mainly as a convenient shortcut, and for completeness with the other methods, as it is identical to calling `where()` and `update()` in the Query Builder. It expects the primary key of the record to be updated as the first parameter, and the `$data` array as the second parameter.

```
$data = array(
        'username' => 'madmax',
        'email' => 'madmax107@thunderdome.com'
);

$this->user_model->update($user_id, $data);
```

delete()

This method takes the primary key of the record to delete as the only parameter. If `$soft_deletes` is set to `TRUE`, then it will simply update the record, setting the `deleted` column to `1` to indicate the record has been deleted. Otherwise, it calls the Query Builder's `delete()` method.

```
$this->user_model->delete($user_id);
```

If you need to override the soft delete check, you can pass in a boolean value to the second parameter. This is especially helpful if you have soft deletes turned on for the model, but you want to permanently delete a record.

```
// Permanently delete the record
$this->user_model->delete($user_id, FALSE);
```

return_as()

This method allows you to change the return types of the find methods on the fly. The only parameter is the type of result to return, either `'array'`, `'object'`, or a fully-qualified class name. This sets the `$temp_return_type` that we saw earlier and is only valid for one `find()` or `find_all()` call, since those methods reset the value to the class' default value.

If the model is set to always return objects, but you have one place where you could really use an array, you can easily override it with this method.

```
$this->user->model->return_as('array')
                  ->find($user_id);
```

Additionally, you can override it to return the result as an instance of a custom class. The class must already be in memory, or able to be found through one of the currently-active autoloaders, like Composer.

```
$this->user->model->return_as("\\App\\Models\\User")
                  ->find($user_id);
```

Using MY_Model

Using this custom `MY_Model` as the basis for your own models is simple. You just need your class to extend `MY_Model`, then set the properties as needed.

```
class Some_model extends MY_Model {

        protected $table_name = 'some_table';
        protected $primary_key = 'id';
        protected $return_type = "\\App\\Models\\SomeClass";

}

// Instantiate your new class as usual
$this->load->model('some_model');

---
```

This has been a quick sample of how to customize your `MY_Model` to suit your workflow. To enhance this, you could add generic custom object saving to the class, hooks, or even data validation, which we will go over in the next section. Hopefully, this section has given you inspiration to customize your own workflow.

In-Model Validation

I think there are many benefits to having your data validation within the model. The biggest one is that you have a single place to modify if you need to change some validation to meet new business requirements. This keeps your controllers very clean, since they simply have to shuffle the items off to the model and then grab the error message from the model if an error happens. It also ensures that you never have to worry about one of the team members (yourself included) forgetting to validate a certain piece of data in exactly the right way, so you will be creating a more secure app in the long run.

Integrating Data Validation

To validate data in the model, we will still use the `Form_validation` library from CodeIgniter, but we will hold all of the rules within the model itself.

Class Properties

Validation Class Properties

```
    /**
     * An array of validation rules. This needs to be the same format
     * as validation rules passed to the Form_validation library.
     * http://www.codeigniter.com/user_guide/libraries/form_validation.html#sett\
ing-rules-using-an-array
     */
    protected $validation_rules = array();

    /**
     * An array of extra rules to add to validation rules during inserts only.
     * Often used for adding 'required' rules to fields on insert, but not udpat\
es.
     *
     *   array( 'username' => 'required|strip_tags' );
     * @var array
     */
    protected $insert_validate_rules = array();

    /**
     * Optionally skip the validation. Used in conjunction with
     * skip_validation() to skip data validation for any future calls.
     */
    protected $skip_validation = FALSE;
```

We use three new class properties to make the system work. The first, `$validation_rules` is an array that holds an associative array of fields in the same format that CodeIgniter's Form Validation[37] library requires when setting rules as an array. You would use this array to setup the validation rules which will be used for both `insert()` and `update()` calls.

```
$validation_rules = array(
        array(
                'field'         => 'email',
                'label' => 'Email Address',
                'rules'         => 'trim|valid_email'
        ),
        array(
                'field'         => 'username',
                'label'         => 'Username',
```

[37] http://www.codeigniter.com/user_guide/libraries/form_validation.html#setting-rules-using-an-array

```
                'rules'        => 'trim|strip_tags|min_length[5]|max_length[20]'
        )
);
```

You may have some slightly different rules that you need applied only during `insert()` calls. This will often include ensuring the field is present and unique. You can use the `$insert_validation_rules` array to collect these rules. These will all be appended to the rules for that field already present in `$validation_rules`.

```
$insert_validation_rules = array(
        'email'        => 'required|is_unique[users.email]',
        'username' => 'required|is_unique[users.username]'
);
```

These two arrays provide all of the power you need to setup validation for your data. The only thing missing that you might consider adding is support for custom error messages. That is not covered in this book and is left as an exercise for you, if needed.

The `$skip_validation` property is a temporary flag that allows us to bypass the validation process when inserting or updating records. This should only be used with trusted data. Typically, this will be used during migrations or in Seeds (discussed later) where you have full control over the data being inserted, or for updates/inserts performed internally by the model itself, without user-supplied data. Always validate data that comes from sources you don't control, like the user, a CSV file, etc.

The Validate Methods

The `validate()` method is the heart of the system, and incorporates all of your settings and rules, runs them, and reports back the success or failure. We will revise the `insert()` and `update()` methods shortly, but first let's take a look at the method itself to understand what it's doing.

Validate Method

```
1     /**
2      * Validates the data passed into it based upon the form_validation rules
3      * setup in the $this->validate property.
4      *
5      * If $type == 'insert', any additional rules in the class var $insert_valid\
6  ate_rules
7      * for that field will be added to the rules.
8      *
9      * @param  array $data An array of validation rules
10     * @param  string $type Either 'update' or 'insert'.
11     * @return array/bool       The original data or FALSE
```

```
12      */
13     public function validate($data, $type = 'update', $skip_validation = NULL)
14     {
15         $skip_validation = is_null($skip_validation) ? $this->skip_validation : \
16 $skip_validation;
17 
18         if ($skip_validation)
19         {
20             return $data;
21         }
22 
23         // We need the database to be loaded up at this point in case
24         // we want to use callbacks that hit the database.
25         if (empty($this->db))
26         {
27             $this->load->database();
28         }
29 
30         if ( ! empty($this->validation_rules))
31         {
32             $this->form_validation->set_data($data);
33 
34             // If we have any insert-only rules, make sure
35             // they are added to the rules before we give them
36             // to the form_validation library.
37             if ($type == 'insert' && is_array($this->insert_validate_rules))
38             {
39                 foreach ($this->validation_rules as &$row)
40                 {
41                     if (isset($this->insert_validate_rules[$row['field']]))
42                     {
43                         $row['rules'] .= '|'. $this->insert_validate_rules[$row[\
44 'field']];
45                     }
46                 }
47             }
48 
49             $this->form_validation->set_rules($this->validation_rules);
50 
51             if ($this->form_validation->run() === TRUE)
52             {
53                 return $data;
```

```
54              } else {
55                  return FALSE;
56              }
57          } else {
58              return $data;
59          }
60      }
```

The first parameter is the `$data` array, which contains the key/value pairs of data from the calling method. This is the data that's being validated. The second parameter can be either `'insert'` or `'update'`, and just indicates which type of validation to perform. If it's an `'insert'`, then it will combine the `$insert_validation_rules`, otherwise, it will just use the `$validation_rules` array. The final parameter is a boolean value that is passed to the system to allow it to override the model-wide `$skip_validation` setting.

The first thing the method does is to ensure that we actually want to perform validation. If a boolean value is passed into the method's `$skip_validation` parameter, it will use that, otherwise it will use the value of the class' `$skip_validation` property. Then, if it's supposed to skip validation, it quickly exits the method so we don't cause any additional overhead. Plus, by getting out first, it limits the number of nested `if` calls in the remaining code and makes things much more readable.

Next, it ensures that the database is loaded using the default connection. If you need a special connection for this model you will need to ensure the database is already loaded with the right connection. If you have multiple connections that you need to support in your application, you will need to modify this method to know which connection it should use, likely by storing the connection as a class property and passing it in during the class' construction.

Now we're on to the heart of the application. The method ensures that the Form Validation library has the right data to work with by passing the `$data` through the `set_data()` method, new to CodeIgniter 3. If we don't pass in the data here, the library will use the data in the `$_POST` array, which isn't what we want here. While many times we could simply pass the `$_POST` array into the model, we keep things as flexible as possible to allow for any future changes in business logic, renaming of fields in the database, etc.

The method then checks if the model is performing an `'insert'` and, if so, merges the arrays so that it has all of the rules to work with, after which it passes the rules on to the Form Validation library.

Finally, it runs the `form_validation->run()` method to perform the actual validation. If it was successful, it passes the `$data` back to the calling method so it can push it into the database. If it doesn't work, it returns `FALSE`, so the calling method can tell that validation failed.

Inserts and Updates

The last piece of the puzzle is to update the `insert()` and `update()` methods to use the validation.

Modified Methods

```
    /**
     * Inserts a new row into the database.
     *
     * @param array $data
     * @return int
     */
    public function insert(array $data, $skip_validation = NULL)
    {
        $skip_validation = is_null($skip_validation) ? $this->skip_validation : \
$skip_validation;

        if ($skip_validation === FALSE)
        {
            $data = $this->validate($data, 'insert', $skip_validation);
        }

        if ($data !== FALSE)
        {
            $result = $this->db->insert($this->table_name, $data);

            return $result ? $this->db->insert_id() : FALSE;
        }

        return FALSE;
    }

    //--------------------------------------------------------------------

    /**
     * Updates a single row based on primary key.
     *
     * @param mixed $id
     * @param array $data
     *
     * @return bool
     */
    public function update($id, array $data, $skip_validation = NULL)
    {
        $skip_validation = is_null($skip_validation) ? $this->skip_validation : \
$skip_validation;

```

```
42          if ($skip_validation === FALSE)
43          {
44              $data = $this->validate($data, 'update', $skip_validation);
45          }
46
47          if ($data !== FALSE)
48          {
49              return $this->db->where($this->primary_key, $id)
50                              ->update($this->table_name, $data);
51          }
52
53          return FALSE;
54      }
```

For both of these methods, we have made almost identical changes. The first change is to add a `$skip_validation` flag as the final parameter to the method. Again, this won't be used very much, but does provide a simple way to override the class-wide settings when needed.

Inside the methods, we check our new parameter and, if `$skip_validation` is `FALSE`, we run the data through the validation method. Assuming the data comes back intact, we then insert/update the data in the database.

Errors?

One last piece of work to ensure a pleasant working environment is to ensure a simple way to get errors. The database layer provides an `error()` method which makes getting database-specific errors back simple. However, what if they are validation errors, and not database errors? In that case, you would check in the controller for the presence of validation errors.

```
if (! $this->user_model->insert($data))
{
        if (validation_errors())
        {
                $this->set_message(validation_errors(), 'warning');
        }
        else
        {
                $this->set_message($this->user_model->error(), 'warning')
        }
}
```

You could incorporate this into a custom `MY_Model` error method, but there are too many times that you might want to do things differently depending on what type of error it is for me to recommend

that method. In an API you might need to throw a different type of error. In more traditional applications, you might want to display verbose validation errors, but for database errors you may want to log the exact error and present a generic message to the user.

Using Multiple Databases

In many cases a single database will be all you need. As your site grows, however, you may start to find performance issues which need to be addressed. Often, the solution to this might be to split things across multiple databases. There are a few situations in which I have seen this needed and/or recommended. Here are a few of them.

Session Management If you need to scale your application horizontally, you will likely start using a load balancer to host your application across several servers. When this happens, you need a way to keep your sessions in one central location, or the users will constantly be logged out because the load balancer sent them to a new server. By using a separate database for storing sessions, all running servers can access the single database and users will stay logged in.

Log Management Writing to a database is quite expensive and takes much more time than reading from the database, so companies will often use a separate database to store all of their logs. This is especially true of companies that need to log every little detail to help them keep an eye on performance, stability, and more. Companies may also use log management software to store remote copies of logs from all of their major applications and systems. By using a separate database, all of the expensive writes are happening away from the database that is used for most of the application, and are not blocking any reads from the main database, keeping the application fast, while still logging many different data points.

Customer Separation If your application is used by multiple customers and stores personal data, like a CRM, you should consider using a different database for each customer. This keeps any sensitive data completely separate so that no other customers can get access to it, even if a bug in your code allows a user to accidentally access additional information. It also makes it easier to restore data when a customer accidentally deletes it. Additionally, performance can be improved because the tables stay smaller.

Separate Reads and Writes For mid-sized applications, you might not need to go to the expense and complication of multiple database servers, but an early performance boost might be gained by simply keeping all of the application *writes* in one database, while using a copy of that database for all *reads*. This has the benefit of never having writes blocking the reads and so helps improve performance. This is most often done by using a master-slave setup where the master database automatically copies its data to one (or more) child databases that are only used for reading. We will look at how to implement this in our `MY_Model` later in this section.

Using Multiple Databases

You configure multiple databases the same way that you configure a single database: in `/application/config/database.php`, or the environment-specific version of the file. The standard config file has a single configuration array named `default`. You can add as many other connections as you need, by just copying the array and providing a new name for it.

```
// Use for all Writes
$db['default'] = array(
        'dsn'        => '',
        'hostname' => '127.0.0.1',
        'username' => '',
        'password' => '',
        'database' => '',
        'dbdriver' => 'mysqli',
        'dbprefix' => '',
        'pconnect' => TRUE,
        'db_debug' => (ENVIRONMENT !== 'production'),
        'cache_on' => FALSE,
        'cachedir' => '',
        'char_set' => 'utf8',
        'dbcollat' => 'utf8_general_ci',
        'swap_pre' => '',
        'encrypt' => FALSE,
        'compress' => FALSE,
        'stricton' => FALSE,
        'failover' => array(),
        'save_queries' => TRUE
);
// Use for all Reads
$db['read_db'] = array(
        'dsn'        => '',
        'hostname' => '127.0.0.1',
        'username' => '',
        'password' => '',
        'database' => '',
        'dbdriver' => 'mysqli',
        'dbprefix' => '',
        'pconnect' => TRUE,
        'db_debug' => (ENVIRONMENT !== 'production'),
        'cache_on' => FALSE,
        'cachedir' => '',
```

```
        'char_set' => 'utf8',
        'dbcollat' => 'utf8_general_ci',
        'swap_pre' => '',
        'encrypt' => FALSE,
        'compress' => FALSE,
        'stricton' => FALSE,
        'failover' => array(),
        'save_queries' => TRUE
);
```

At the top of the config file is a setting (`$active_group`) that allows you to set the primary configuration array, which starts off set to `default`. If you change the name of the default configuration to something more descriptive (and you should when using multiple databases), then you should change this value to match your desired primary array.

```
$active_group = 'write_db';
```

Whenever you load the database with `$this->load->database()`, the configuration array that matches the `$active_group` will be the database that is connected. This will also be the database that is used by models that auto-connect to the database.

At any time, you can get a connection to any of the other configurations by passing the name of the database connection group to the `$this->load->database()` method. Passing `TRUE` as the second parameter will return the new database object, instead of initializing/overriding `$this->db`.

```
$read_db = $this->load->database('read_db', TRUE);
```

Because the uses for multiple databases can vary quite a bit, CodeIgniter doesn't provide any built-in way to manage this other than connecting and returning the connection to you. In the next section, we will look at a way to modify our `MY_Model` class to use separate read and write database connections.

Multiple Databases In MY_Model

When you need to use multiple databases, the easiest thing to do is to incorporate it into `MY_Model` so you always have what you need in place when working on custom methods for you models.

MY_Model With Multiple Database Support

```php
<?php

class MY_model extends CI_Model {

    protected $table_name;

    protected $primary_key = 'id';

    /**
     * If TRUE, delete methods will not delete the row,
     * but will simply set the 'deleted' column to '1'.
     * @var bool
     */
    protected $soft_deletes = FALSE;

    /**
     * Specifies the way that individual rows
     * are returned to the user. Valid values
     * are: 'array', 'object', or the name of a class,
     * including namespace, if applicable.
     */
    protected $return_type = 'array';
    protected $temp_return_type = NULL;

    /**
     * Stores the database connections for
     * separate read and write databases.
     */
    protected $write_db;
    protected $read_db;

    //--------------------------------------------------------------------

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

        $this->temp_return_type = $this->return_type;
    }

    //--------------------------------------------------------------------
```

```
42
43      /**
44       * Retrieves a single row from the database by primary key.
45       *
46       * @param mixed $id
47       * @return mixed
48       */
49      public function find($id)
50      {
51          $this->ensure_db('read');
52
53          $query = $this->read_db->where($this->primary_key, $id)
54                                 ->get($this->table_name);
55
56          $row = $this->temp_return_type == 'array' ?
57                      $query->row_array() :
58                      $query->row(0, $this->temp_return_type);
59
60          $this->temp_return_type = $this->return_type;
61
62          return $row;
63      }
64
65      //--------------------------------------------------------------------
66
67      /**
68       * Returns all records. Intended to be modified with calls to db library,
69       * like:
70       *     $users = $this->user_model->where('active', 1)
71       *                               ->limit(10, 0)
72       *                               ->find_all();
73       */
74      public function find_all()
75      {
76          $this->ensure_db('read');
77
78          $query = $this->read_db->get($this->table_name);
79
80          $rows = $this->temp_return_type == 'array' ?
81                      $query->result_array() :
82                      $query->result($this->temp_return_type);
83
```

```
84          $this->temp_return_type = $this->return_type;
85
86          return $rows;
87      }
88
89      //--------------------------------------------------------------------
90
91      /**
92       * Inserts a new row into the database.
93       *
94       * @param array $data
95       * @return int
96       */
97      public function insert(array $data)
98      {
99          $this->ensure_db('write');
100
101         $this->write_db->insert($this->table_name, $data);
102
103         return $this->db->insert_id();
104     }
105
106     //--------------------------------------------------------------------
107
108     /**
109      * Updates a single row based on primary key.
110      *
111      * @param mixed $id
112      * @param array $data
113      *
114      * @return bool
115      */
116     public function update($id, array $data)
117     {
118         $this->ensure_db('write');
119
120         return $this->write_db->where($this->primary_key, $id)
121                               ->update($this->table_name, $data);
122     }
123
124     //--------------------------------------------------------------------
125
```

```
126     /**
127      * Deletes a single row by primary key. If 'soft_deletes' are TRUE,
128      * will simply change the 'deleted' column to a value of '1'.
129      *
130      * @param mixed $id
131      * @return bool
132      */
133     public function delete($id)
134     {
135         $this->ensure_db('write');
136 
137         if ($this->soft_deletes)
138         {
139             return $this->update($id, array('deleted' => 1));
140         }
141 
142         return $this->write_db->where($this->primary_key, $id)
143                               ->delete($this->table_name);
144     }
145 
146     //--------------------------------------------------------------------
147 
148     /**
149      * Allows you to set the type that each object is returned as, either
150      * 'array', 'object', or a custom class name, with namespace if applicable.
151      *
152      * @param string $type
153      * @return $this
154      */
155     public function return_as($type)
156     {
157         $this->return_type = $type;
158 
159         return $this;
160     }
161 
162     //--------------------------------------------------------------------
163 
164     /**
165      * Ensures that the proper database is loaded.
166      *
167      * @param string $type  Either 'read' or 'write'
```

```
168      */
169     public function ensure_db($type)
170     {
171         if ($type == 'read' && empty($this->read_db))
172         {
173             $this->read_db = $this->load->database('read_db', TRUE);
174         }
175         elseif ($type == 'write' && empty($this->write_db))
176         {
177             $this->write_db = $this->load->database('write_db', TRUE);
178         }
179     }
180 
181     //--------------------------------------------------------------------
182 
183     /**
184      * Allow access to the database functions directly without us needing
185      * to copy every method over to this class.
186      *
187      * @param string $name
188      * @param array  $arguments
189      * @return $this
190      */
191     public function __call($name, array $arguments = null)
192     {
193         if (method_exists($this->db, $name))
194         {
195             call_user_func_array(array($this->db, $name), $arguments);
196 
197             return $this;
198         }
199     }
200 
201     //--------------------------------------------------------------------
202 
203 }
```

We have two new class properties, `$read_db` and `$write_db`. These will store the database connections for your read and write databases, respectively. To keep performance high, we only connect to each database when it is needed.

This happens in the `ensure_db()` method. It only takes a single parameter, which is set to either `read` or `write` to specify what type of connection is desired. Inside the method, it loads up the appropriate

connection and stores it in the correct class property so it's available for other calls.

Finally, in all of the CRUD methods, we call `$this->ensure_db()` to make sure our database is available, and that's really all there is to make it work. If you implement this in the real world, you would also want to create an `error()` method that would be able to grab the error from the appropriate database.

Migrations

Migrations provide a way to version your database schema across all installations. Just like you can keep versions of your application in a system like Git, migrations[38] allow you to easily keep a record of changes to your database schema and move between versions or roll back changes as needed. Since they are all stored in standard PHP files, you can tuck them away in Git just like all of your other files. Then, all installations, whether they are on multiple developers' personal computers, a QA server, or even the production server, can have the instructions required to keep the database schema up to date.

Migrations are simply PHP files containing `up()` and `down()` methods to apply the database changes, or roll them back, respectively.

It should be noted that migrations only perform actions which you explicitly define in their `up()` and `down()` methods. They do not behave like a version control system for your data. If you make changes to the structure of existing tables, you can lose data, so you need to keep that in mind and use migrations carefully, but they are a very powerful tool.

Creating Migrations

Migrations are simple to create, but first you have to decide on a numbering scheme, since CodeIgniter now supports two types of migration file numbering: sequential or timestamps.

Sequential migrations have a 3-digit number and an underscore as the prefix of their file names. An example of a file name in this format would be `001_Add_blog_table.php`. In order to run migrations, you must ensure that the numbers continue uninterrupted, otherwise the migration will stop at the gap between numbers and the remaining migrations will not run.

Timestamp migrations have a timestamp instead of the 3-digit number in the file name prefix. The timestamp is in the format `yyyymmddhhiiss`, or all of the date and time elements from year on the left to seconds on the right. This is the preferred method, since it is easier for multiple developers working on the same project to create migrations without coordinating sequential migration numbers. However, developers still need to ensure that they are not creating migrations against the same table(s) without coordinating their efforts. This is also the default value.

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

To create a new migration you would create a file in `/application/migrations/`. If we go with the example of creating a blog table, it might be named, `/application/migrations/20150902110217_-Create_blog_table.php`. Inside that file we create a new class which extends `CI_Migration`. The name of the class must start with `Migration`, followed by the portion of the filename after the timestamp/sequential prefix. For our example, we would need a class named `Migration_Create_-blog_table`.

The class must contain two methods: `up()` and `down()`, so our file would look something like:

```
1  class Migration_Create_blog_table extends CI_Migration {
2
3          public function up()
4          {
5                  $this->dbforge->add_field(array(
6                          'blog_id' => array(
7                                  'type'           => 'INT',
8                                  'constraint'     => 5,
9                                  'unsigned'       => TRUE,
10                                 'auto_increment' => TRUE
11                         ),
12                         'blog_title' => array(
13                                 'type'       => 'VARCHAR',
14                                 'constraint' => '100',
15                         ),
16                         'blog_description' => array(
17                                 'type' => 'TEXT',
18                                 'null' => TRUE,
19                         ),
20                 ));
21                 $this->dbforge->add_key('blog_id', TRUE);
22                 $this->dbforge->create_table('blog');
23         }
24
25         public function down()
26         {
27                 $this->dbforge->drop_table('blog');
28         }
29 }
```

As you can see in the example, the Database Forge[39] is already loaded for you to use, as is the usual database library (`$this->db`).

[39] http://www.codeigniter.com/user_guide/database/forge.html

Using Migrations

There are three primary tasks within migrations:

- Create new tables
- Modify existing tables
- Modify existing data

The first two are fairly obvious, but the third option is just as important since these changes will eventually be made on the production server with live data. You might need to modify existing data if your schema has changed and you're moving data from one table to another. You also might have had some business rules change that would affect existing data, and migrations are the perfect way to ensure those changes are applied to that data. You might need to change the format of data in some way. All of these, and more, are perfectly good reasons to modify existing data with migrations.

The only big problem with migrations in CodeIgniter is that there is no built-in method to run the migrations via the CLI, so you can't have them run automatically without setting up your own system. Luckily, we built that feature into the `MY_Controller` class earlier in the book. Before we look at that, we should head over to the `/application/config/migrations.php` file to make sure our settings are correct.

The primary setting you will need to change is to set `migration_enabled` to `TRUE`. If this value is `FALSE`, no amount of magic in our controller will make it happen.

```
$config['migration_enabled'] = TRUE;
```

An interesting setting is the `migration_auto_latest` flag. When set to `TRUE`, the migrations will automatically run to the latest version for you. While this sounds like what we want, it has the drawback of always running to the latest version. This can be an issue if we need to rollback some changes and stay at a previous version for a little while as we fix the migrations, fix our code, etc. Instead of staying at that version, the library would automatically push us to the latest version the next time it is loaded.

The `migration_version` setting sounds like it could be used to set the version and not have it migrate past that version when automatically migrating to the latest version. Unfortunately, that's not the case. So, the built-in auto-latest features do not work for me.

Migrations and the Controller

If you take a look back at the `MY_Controller` we worked on earlier, you'll see we added a couple of pretty handy features.

MY_Controller Migration Support

```
        // Try to auto-migrate any files stored in APPPATH ./migrations
        if ($this->auto_migrate === TRUE)
        {
            $this->load->library('migration');

            // We can specify a version to migrate to by appending
            //  ?migrate_to=X
            // in the URL.

            if ($mig_version = $this->input->get('migrate_to'))
            {
                $this->migration->version($mig_version);
            }
            else
            {
                $this->migration->latest();
            }
        }
```

This checks the `$auto_migrate` property to see if it is `TRUE`. If so, we load up the migration library and migrate to the latest version. Since this is in the base controller, which will be extended by almost all of our other controllers, this will happen on every page load. While this is handy in development, you probably want to ensure this is set to `FALSE` on a production server.

This also allows you to specify a specific version that you want to migrate to simply by appending `?migrate_to=20150814101112` to the current URL in your browser, which will cause it to migrate backwards or forwards to the specified version.

Migrations are extremely handy and should be used by everyone. The benefits far outweigh the time it takes to write them.

Data Seeders

Database seeding is the initial population of a database with data. Quite often this is used to fill a database with dummy data. This allows you to develop your application and be able to see what it would look like with live data, even though you don't have any real data to use in it. Other times, seeding can be used to populate required information, like initial admin accounts. This is popular in other frameworks like Laravel or even Microsoft's EntityFramework for ASP.net.

While CodeIgniter doesn't ship with a way to do database seeding, it's pretty simple to add support for it. This only requires a single library that forms the base class your seeders will extend, and a controller to allow you to tell it to actually seed the database.

First, we will look at the Seeder library and go over how it works. After that, we will look at one way to implement a calling controller.

The Seeder Library

Seeder Library

```php
<?php

/**
 * Provides a base class for your own Seeder libraries,
 * to populate the database with data.
 */
class Seeder {

    protected $ci;

    protected $db;

    protected $dbforge;

    //--------------------------------------------------------------------

    public function __construct()
    {
        $this->ci =& get_instance();

        // Ensure our database is loaded and ready
        $this->ci->load->database();
        $this->ci->load->dbforge();

        // Setup some convenience variables
        $this->db      =& $this->ci->db;
        $this->dbforge =& $this->ci->dbforge;
    }

    //--------------------------------------------------------------------

    /**
```

```php
     * Runs the database seeds. This is where the magic happens.
     * This method MUST be overridden by the child classes.
     */
    public function run() {}

    //--------------------------------------------------------------------

    public function call($class)
    {
        if (empty($class))
        {
            show_error('No Seeder was specified.');
        }

        $path = APPPATH .'seeds/'. str_ireplace('.php', '', $class) .'.php';

        if ( ! is_file($path))
        {
            show_error("Unable to find the Seed class: {$class}");
        }

        try {
            require $path;

            $seeder = new $class();

            $seeder->run();

            unset($seeder);
        }
        catch (\Exception $e)
        {
            show_error($e->getMessage(), $e->getCode());
        }

        return TRUE;
    }

    //--------------------------------------------------------------------

}
```

The library starts off by ensuring that our seed classes will have ready access to the CodeIgniter instance, as well as the database and dbforge. In the constructor, we save references to these three as properties. This isn't strictly necessary, but makes working with them a bit nicer since you only need to type `$this->db` instead of `$this->ci->db`. We also ensure the database is loaded, so we have a connection ready to use.

The `run()` method is defined as an empty method. It's not needed in this class, but it is required in all child classes since that is the method that is called to do the seeding. When you create your child classes, this is the only method that you need to define. You would add any data generation and storing functions inside this method.

The `call()` method is what actually looks for the seed class, loads it, and calls the `run()` method. It's the engine that makes it all work. It takes a single parameter: the name of the class to load and seed. The class name must match the file name in order for this to work. This class will look in `/application/seeds` for a file with a name that matches the class name. It loads the class, creates an instance of it, and calls the `run()` method on the instance.

Because this method is part of the class you will extend, this also allows you to call other seed classes from a seed class, allowing you to organize the seeds however you see fit. You might create a `TestSeeder` class that populates the database with dummy data, and other seeders that seed one specific data type. This helps keep the classes smaller, more focused, and easier to maintain.

A Seeding Controller

Now that we have a working class we can use, we need to create a way to call it. In this example, we will create a seeding controller which must be called from the command line. This prevents strangers from destroying our live database by calling a seed method from the browser.

Let's create a new controller at `/application/controllers/Database.php`. You can combine other utilities in here, as well. For example, in similar controllers I have combined seeding with migration tools to install a certain version, or even completely refresh the database.

```
1  class Database {
2
3          public function seed($class)
4          {
5                  if ( ! $this->input->is_cli_request())
6                  {
7                          show_error('Seeding can only happen from the command line.');
8                  }
9
10                 $this->load->library('seeder');
11
12                 $this->seeder->call($name);
```

```
13          }
14  }
```

That's all that is needed to make this system work. It first checks to verify that we are using the command line and, if not, it will stop execution and display an error. Next, it loads up the Seeder library and calls the `call()` method to run everything.

You can run the seeder from the command line with the following command:

```
$ php index.php database seed MySeeder
```

Creating Seeds

Creating new seeds is simple. You create a new file in the `seeds` directory with a class containing a `run()` method. Inside that method, you get creative and start adding data to the database. It can be as simple as the following:

```
1   class AdminSeeder extends Seeder {
2
3           public function run()
4           {
5                   $user = array(
6                           'username' => 'kingdarth',
7                           'email' => 'darth@theempire.com',
8                   );
9                   $this->db->insert('users', $user);
10          }
11  }
```

Instead of adding data directly, you could also load up one of your models, using the methods in there to insert the data. This ensures that any processes needed in the model are run on the data. In this example, it might be ensuring the password is hashed and the user is forced to create a new password the next time they log in.

```
public function run()
{
        $this->ci->load->model('user_model');

        $user = array(
                'username' => 'kingdarth',
                'email' => 'darth@theempire.com',
                'password' => 'password'
        );

        $user_id = $this->ci->user_model->insert($user)
        $this->ci->user_model->forcePasswordReset($user_id);
}
```

There are also great tools out there for generating dummy data, the best of which is probably Faker[40], which is capable of creating nearly any type of dummy data you need. It can be loaded via Composer and used to generate your data. The details of using this library require much more room than can be dedicated to it here, but a good starting point is the library's website, and the article Simplifying Test Data Generation with Faker[41], at SitePoint.

[40] https://github.com/fzaninotto/Faker

[41] http://www.sitepoint.com/simplifying-test-data-generation-with-faker/

6. AJAX Is Easy

AJAX is a concept that is pretty simple at its core. The complexity comes with the many different variations and the different JavaScript frameworks that people use to do this. To make this more easily digestible, we are going to focus on what is probably the most widely-used, and one of the easiest methods to get started with: jQuery. After all, this is not a book about JavaScript, it's a book about CodeIgniter. AJAX is here to stay, and we need to be able to work with it in a manner that is simple to understand and implement, and organized so that anyone who enters the project can quickly come to terms with what we're doing.

What Is AJAX, Really?

AJAX, as used today, is really a bit of a misnomer. It is a name for a collection of techniques and tools combining server-side programming with client-side JavaScript, CSS, HTML, and the DOM. Originally, it stood for "Asynchronous JavaScript and XML". The term was coined in a 2005 article by Jesse James Garret, titled Ajax: A New Approach to Web Applications[1], which was based on approaches developed at Google.

Due to its verbosity, XML has fallen out of favor, with most people using JSON[2] in its place. JSON is used to pass plain JavaScript objects back and forth between the client and the server. This is technically referred to as AJAJ, but it's generally accepted that AJAX doesn't require XML, and JSON is frequently used in its place.

By combining the technologies mentioned earlier, AJAX allows you to dynamically update a part of a web-page without needing to refresh the entire page. It is at the core of what allows you to create elegant web applications that function more like a traditional software application. It can be used to update data based on choices made in the UI, update status messages, or even create an entire application in a single HTML page where everything is loaded dynamically, much like traditional GUI software programming.

AHAH! AJAX's Cousin

While JSON is a very elegant solution for passing data back and forth, it does often require that the UI is generated on the client side. With the power of today's devices, this is quite often just fine. However, there are other options. The most common of these is **AHAH**, or *Asynchronous HTTP and HTML*, which retrieves HTML fragments from the server and inserts them directly into the DOM, with no need to process them on the client.

---

[1] http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

[2] http://json.org/

Perhaps my favorite solution is a combination of AJAX and AHAH, where the server returns a JSON object that contains a list of DOM entity ids, and the HTML that should replace what's currently there. This allows multiple pieces of the UI to be updated in a single HTTP request.

Server-Side vs. Client-Side Rendering

Since we brought up the debate between whether it's better to render the data on the client or the server, which should you opt for? In this book, I'll be covering the combination I just mentioned, but you should know the details of the choices before you decide for your own application. You might make a different choice for different applications, depending on the application's requirements. We will do this by exploring Twitter's decisions over the years.

When Twitter was first created, they used a single-page architecture, where all UI rendering and logic was pushed to the client. The browser then talked directly with Twitter's API to get the data, and knew how to render it. While this approach allowed them to open up new, dynamic possibilities that hadn't been used much prior to that, it had a few drawbacks. Since all of the logic and templates were stored in JavaScript, the initial page load was pretty heavy and could take a couple of seconds. Since different browsers handled JavaScript with different levels of performance, their site performed differently in each browser.

In May, 2012, Twitter announced[3] they were switching from their all-JavaScript front end to move more of the processing to the server, for performance reasons. They wanted to improve the page load time and the time to first tweet. Their solution used JavaScript modules that could be optimized for delivery, including minimizing how much was needed, and when more modules should be loaded.

This conversion did a few other things on the performance side. First, they no longer had to download the HTML file, the JavaScript, then parse the hashtag, then download the proper page. Instead, they could display what the user wanted to see immediately. No extra steps. Second, it allowed them to take advantage of server-side performance tools. While their post doesn't specifically mention it, I imagine the ability to use server-side caching of content was a big win, too.

Here are some other articles that talk about server-side rendering vs. client-side rendering if you want to learn more:

- Client-side vs Server-side Rendering[4], by Karl Seguin
- Client vs serverside rendering- the big battle?[5] at m-way solutions.
- How Basecamp Next got to be so damn fast without using much client-side UI[6] by David Heinemeier Hansson.
- Server-generated JavaScript Responses[7] by David Heinemeier Hansson

[3] https://blog.twitter.com/2012/improving-performance-on-twittercom

[4] http://openmymind.net/2012/5/30/Client-Side-vs-Server-Side-Rendering/

[5] http://blog.mwaysolutions.com/2013/11/08/client-vs-serverside-rendering-the-big-battle-2/

[6] https://signalvnoise.com/posts/3112-how-basecamp-next-got-to-be-so-damn-fast-without-using-much-client-side-u

[7] https://signalvnoise.com/posts/3697-server-generated-javascript-responses

jQuery Setup

jQuery is used for the examples in this book simply because its use is so widespread, and it makes it very simple to work with AJAX. While I'll explain what each of the functions in this example does, I won't cover every little detail. The functions are flexible, powerful, and also quite well documented[8].

I find it handy to keep a simple `ajax.js` file around that I use in most of my projects. It gets tweaked in each project to include project-specific tasks, but this is an extremely handy set of utilities. With a little bit of time, you could turn it into something first-rate.

Showing & Hiding the Loading Status

Whenever an AJAX call is made to the server, there is no indication to the user. If that call takes a long time, or gets stuck because of server-side errors, the page could appear completely unresponsive. This will cause your users to run away fast, so it's up to you to make sure that you show the user that something is happening in the background. There are many ways to do this. For this example, we will use an approach similar to Gmail's method - a small div at the top center of the page simply says "Loading...".

Since we don't want to specifically tell the Loading div to show and hide manually every time, we need to make that happen automatically. Thoughtfully, jQuery has provided the `ajaxStart` and `ajaxStop` functions for just that purpose.

Ajax.js

```
1  /**
2   * ajaxStart()
3   *
4   * Simply makes the loader img visible when the first
5   * AJAX method starts.
6   */
7  $(document).ajaxStart(function(){
8      $('#ajax-loader').css('display', 'block').css('top', posTop() + 15);
9  });
10
11 //---------------------------------------------------------------------
12
13 /**
14  * ajaxStop()
15  *
16  * Simply hides the loader img when the last running
17  * AJAX method is done.
```

[8] http://api.jquery.com/

```
 */
$(document).ajaxStop(function(){
    $('#ajax-loader').css('display', 'none');
});

//---------------------------------------------------------------------

/**
 * Browser Position
 * copyright Stephen Chapman, 3rd Jan 2005, 8th Dec 2005
 */
function posTop()
{
    return typeof window.pageYOffset != 'undefined' ?  window.pageYOffset : docu\
ment.documentElement && document.documentElement.scrollTop ? document.documentEl\
ement.scrollTop : document.body.scrollTop ? document.body.scrollTop : 0;
}
```

We tap into the `ajaxStart()` event to display the Loading message whenever an AJAX call is fired. If the server hangs and never responds, then this will be all that is shown. This method makes use of the `posTop()` function that I have used for so many years that I can't remember where I found it. It returns the current position of the top of the visible browser window. This allows us to always set the top of the loading div relative to the top of viewable window. It always shows up this way, instead of scrolling the page back to the top.

For the loading message itself, you will need a div with an id of "ajax-loader" somewhere on every page. I always put it at the bottom of my page template so that I never need to think about it, since it's always just there.

```
<div id="ajax-loader" style="display: none">
        Loading...
</div>
```

Next, we tap into the `ajaxStop` event to hide the div once the AJAX call is finished.

Loading Page Fragments

In this file, I've also included a couple of utility functions which I find make working with a dynamic page very pleasurable. We have the ability to return several separate bits of HTML from a single server call. Each fragment is identified by an id or class. You can use this when creating a new resource. The fragments returned might include a copy of the new resource, a new count of the total of all of these objects, and a status message.

Ajax.js

```
/*
    Responsible for loading a page fragement into a specific element.
 */
function loadFragment(_url, _target)
{
    $.post(_url, function(data)
    {
      // Any HTML content?
      if (data.html != undefined)
      {
          $(_target).html(data.html);
      }

        // If we have a 'fragments' element, then
        // assign those to the fragments they represent.
        if (data.fragments != undefined)
        {
            for (var k in data.fragments)
            {
                if (typeof data.fragments[k] !== 'function')
                {
                    $(k).html(data.fragments[k]);
                }
            }
        }
    },
    'json');
}

//---------------------------------------------------------------------
```

The first function, `loadFragment()`, is the function at the core of this methodology. It takes a URL and an identifier used to determine where the resulting HTML should be placed. The URL should be the full URL to which a POST request will be sent. The target parameter is usually an ID since you usually only want the results placed into a single element on the page.

```
<div id="myTitle">Some Title Here</div>

<script>
        // Get a random title from the server
        loadFragment(
                'http://example.com/ajax/random_title',
                '#myTitle'
        );
</script>
```

The server should return a single element called "html". The content of this element will be inserted into the `#myTitle` div. This uses the `renderJSON()` method from the chapter on Controllers to set the content type to JSON and return the encoded data.

```
public function random_title()
{
        $title = rand($this->titles[rand(0,20)]);

        $this->renderJSON(['html' => $title]);
}
```

What if we want to update a random number at the same time? We could return a `fragments` array as part of the JSON. The `loadFragment()` function looks for an element with a matching name for each key and inserts the value into that HTML element.

```
public function random_title()
{
        $return = [
                'html' => rand($this->titles[rand(0,20)]);
                'fragments' => [
                        '#myNumber' => rand(1, 100);
                ]
        ];

        $this->renderJSON($return);
}
```

Magical Anchors

While this works well, it's tedious and error-prone to manually call that method every time. What if we could add a couple of attributes to the anchor elements that trigger the effect and have the script take care of the rest? Well, we can!

Ajax.js

```
/**
 * Grabs data from the link and loads a page fragment into elements
 * specified by the link's data-url and data-target attributes.
 */
$('body').on('click', 'a.ajax-fragment', function (e)
{
    var url    = $(this).attr('data-url');
    var target = $(this).attr('data-target');

    if (url == undefined || target == undefined)
    {
        alert('AJAX-Fragment link missing required data attributes!');
        return false;
    }

    loadFragment(url, target);
});

//---------------------------------------------------------------------
```

This code binds a function which is triggered whenever an anchor with a class of "ajax-fragment" is clicked. We use the `$('body').on(...);` syntax because this allows it to work even with elements that have been loaded through AJAX, and allows us to handle click events on all of the potential anchors with a single event handler in the DOM.

Whenever a link with the "ajax-fragment" class is clicked, the function looks for two attributes on the link itself: `data-url` and `data-target`. These will be familiar because they are the same attributes which get passed to the `loadFragment()` function. Once it has the attributes, it simply calls the `loadFragment()` function to do the heavy lifting.

```
<a href="#" data-url="http://example.com/ajax/random_title" data-target="#myTitl\
e">
        Randomize Title
</a>
```

Code Organization

I mentioned this in an earlier chapter, but it should be repeated here for ease of reference. What is a good way to organize your code in the application while working with AJAX? First, let's look at one way I do *not* recommend, and then a couple of different strategies that have worked well for me in the past.

AJAX Controller

If a project is a bit small, it might seem like a good idea to include all AJAX methods in one controller, say `/application/controllers/Ajax.php`. I don't recommend it. While you always know which controller to check for AJAX methods, it quickly gets out of hand. Having multiple methods from varied different topics or domains in one file makes it hard to navigate and find what you need. It works, but it isn't pretty, and it's not the most fun to maintain.

Mixing AJAX & Non-AJAX Methods

You could keep your AJAX methods inside the normal controller for that resource. For example, you have a controller for a photo gallery. This controller has all of the user-facing methods for showing galleries, listing the galleries, and showing individual photos. You might have a couple of methods that are used only by AJAX to handle some small gallery status items, like favoriting a gallery, etc. If there are only a couple of methods, it might be fine to keep them in the main Galleries controller. Make sure to group them together, though, so they're all easy to find within the file. If there are more than two or three methods, though, I would shy away from this style. The larger a file gets, the harder it is to maintain.

AJAX Directory

The best way that I've found is to create a new directory under the controllers directory dedicated to your AJAX classes, say `/application/controllers/ajax/`. New controllers are created for each resource type, like Galleries, Photos, Articles, etc. This keeps the location of the functionality you're looking for crystal clear in your mind, and makes it simple for any other developers coming into the project to locate the code they need.

```
/application
        /controllers
                /ajax
                        Galleries.php
                        Photos.php
                        Articles.php
```

7. Working With Files

Working with files in PHP is fairly straightforward, but the commands needed can be a bit obtuse, and the processes verbose, since they have a tendency to be fairly low-level functions. CodeIgniter provides a number of helpers to make working with files a much more pleasant experience. The methods it provides can be much higher-level, meaning they hide many of the details. They also provide common workarounds for potential cross-platform issues. I highly recommend becoming familiar with the provided file helpers.

CodeIgniter's File Helpers

The provided functions are spread across several helper files and a couple of libraries. I'll list the functions here with brief descriptions, but will leave the in-depth details to the User Guide.

security_helper

This helper includes one function that we're interested in here: `sanitize_filename`. This takes only a single parameter: the path to the file itself. This function primarily protects against directory traversal attacks on the system, but does some additional security cleanup as well. It should be used whenever you're reading or writing a filename that was provided by anyone other than the application itself.

```
$this->load->helper('security');
$filepath = $path . $filename . $ext;
$filepath = sanitize_filename($filepath);
```

directory_helper

This file contains a single function, `directory_map`, which allows you to very easily build arrays of files and directories. You can control the depth of directory traversal as well as whether hidden files are shown.

file_helper

This file contains the majority of the helper functions for working with files, as you would expect.

write_file()

Generates a lock on a file and writes data to that file. You can supply a file mode as the third parameter to change many aspects of the function's handling of the file. By default, the function will write the passed data to the file. If the file exists, any data will be overwritten; if the file does not exist, it will be created.

delete_files()

Deletes all files within the specified directory. By default, this function will not recursively delete sub-directories, but passing `TRUE` as the second parameter will do just that.

get_filenames()

Gets an array of filenames contained within a directory. If the optional second parameter is `TRUE`, the path to the file will be prepended.

get_dir_file_info()

Scans a directory and returns an array of files and meta-data about those files, including the filename, file size, date, and permissions.

get_file_info()

Grabs information about a single file, including the name, size, date, and file permissions.

get_mime_by_extension()

Looks at a file's extension and attempts to determine the MIME type using `/applications/config/mimes.php`. This command is NOT secure, and is intended for convenience only.

symbolic_permissions()

Takes a numeric file permission, like 0755, and returns a string representation of the file permissions, as you might see using the Linux command `ls -al` (e.g. `-rw-r--r--`).

octal_permissions()

Takes a numeric file permission in the format returned by `fileperms()` and returns the 3-digit octal representation, (e.g. 644).

path_helper

This file contains a single function, `set_realpath()`, which can be used whenever you need to convert a relative path into a full server path. While you could simply use `realpath()` to do much the same thing, this function provides some additional security checks against accidentally including remote URLs, and can verify that the file exists.

FTP and You

CodeIgniter has a very powerful FTP library[1] that provides all of the tools you would need to work with files on an FTP server, including the following methods:

- `upload()` and `download()` for basic file transfers.
- `mirror()` to copy an entire local directory to a remote server, overwriting any existing files in the process.
- `rename()`, `move()`, `delete_file()`, `delete_dir()`, and `chmod()` for basic remote file management.
- `list_files()` to actually see what's there.
- `mkdir()`, and `changedir()` for navigating the remote server.

Potential Uses

My need for the FTP library has been pretty minimal over the years, to be honest, but the following are some examples of how it might be used.

Server Separation In one application that I worked on, they had many dealers that were allowed to upload data to an anonymous FTP directory on one of their servers. While this server contained parts of the application (it had 3 separate, large parts to it), and these transfers were typically handled through an automated listing service, the team decided that the more sensitive portions of the application should be run on a different server. The FTP tools were used to login to that server and copy the files to the main server every night during a cronjob. Once downloaded, another cronjob would run imports on all of the vehicles so they could go through the rest of the process on that site.

Local Site Deployment For some teams, a simple deployment system can be created through FTP that is easier to manage than setting up a Capistrano install. You could even create a local dashboard that allows you to push copies of any site live with a single click. If you go this route, you will want to ensure that your site continues without interruption by maintaining two directories on the remote server, uploading to the non-live one, then updating a symbolic link where the web site is actually served from.

[1] http://www.codeigniter.com/user_guide/libraries/ftp.html

Application Updates If you have an application that has been given or sold to many people, such as a CMS or gallery script, you could have updates automatically downloaded on a nightly basis and waiting for the next time users logged in and wanted to update.

Backups This is one of the most common uses I have found for the library. It can be used for handling automatic backups of user-uploaded files, or raw SQL dumps of the current database, then copying the files to a remote server for safe-keeping. This is such a common task that we are going to step through an example that does just that.

Simple Database Backup

For our example, we are going to back up our database, zip it up, and ftp it to a remote server for safe keeping.

Simple Backup Class

```
28  <?php
29
30  class DBBackup {
31
32      /**
33       * Creates a backup of the current database, zips it up,
34       * and FTP's the results to a backup server. This script assumes
35       * that the FTP configuration is setup within a config file at
36       * /application/config/ftp.php.
37       *
38       * NOTE: The db backup used here only works on MySQL databases, and
39       * cannot process huge databases without hitting memory and time
40       * limits imposed by most PHP settings. It should be considered
41       * for example, or for smaller sites, only.
42       */
43      public function do_backup()
44      {
45          $ci =& get_instance();
46
47          $ci->load->dbutil();
48
49          $prefs = [
50              'ignore' => ['ci_sessions'],    // We don't want to keep our sessions
51              'format' => 'txt',
52              'add_drop'   => true,           // DROP tables during imports
53              'add_insert' => true,           // Include INSERT statements for dat\
54  a,
```

```
55              'foreign_key_checks' => false   // disable foreign key checks for ea\
56 sier imports
57         ];
58 
59         // Create a SQL dump and store it in the $backup variable
60         $backup = $ci->dbutil->backup($prefs);
61 
62         // Compress and write it to disk.
63         $ci->load->library('zip');
64 
65         $date = date('Y-m-d-H-i-s');
66         $local_file = APPPATH ."cache/db_backup_{$date}.zip";
67         $remote_file = "/backups/db_backup_{$date}.zip";
68 
69         $ci->zip->add_data("db_backup_{$date}.sql", $backup);
70         $ci->zip->archive($local_file);
71 
72         // FTP it to the destination
73         $ci->load->library('ftp');
74 
75         $ci->ftp->connect();
76 
77         $result = $ci->ftp->upload($local_file, $remote_file);
78 
79         $ci->ftp->close();
80 
81         return $result;
82     }
83 
84     //--------------------------------------------------------------------
85 
86 }
```

The first step is backing up the database. To do this, we load up the `dbutil` library and setup our preferences:

- we want to backup all tables except the `ci_sessions` table
- we want the backup in text format
- it should include DROP TABLE statements to completely replace tables during imports
- it should include INSERT statements so the data is actually saved, not just the structure
- foreign key checks should be disabled during the import, which is necessary when we are telling it to drop tables.

Then we create the SQL dump and store it in the `$backup` variable.

At this point, the backup doesn't exist as a file, just in memory, so we need to do something about that. We load up the zip library and add this data to a new archive on the disk in `APPPATH/cache/`, since we know that that location should always be writable. You might want to use your server's temp path, instead.

Finally, we load up the FTP library and connect to the server. In this example, we assume that all configuration is stored in a configuration file. This is the recommended way to do it, because you can then take advantage of environments and have different credentials locally than you do on a production server. This configuration is loaded automatically whenever the library is loaded.

We tell the FTP library to upload the file and store the result (a boolean value) so that we can close the connection. We return the boolean result so we can generate a simple success/fail message.

As with most examples, before using this in a production setting, you would want to put some error checking and fail-safes in place.

Multiple File Uploads

One question that I have seen crop up over and over again on the forums is how to get multiple file uploads working. This section will explore how HTTP works for most multiple-file upload solutions, what the data should look like when it gets to our server, and how we can work with it through CodeIgniter's Upload library.

Typically, multiple file uploads are handled using the array syntax for the names of the file inputs. In my experience, Javascript solutions often do it this way. A simple HTML form would look something like this:

```
<form action="" method="post" enctype="multipart/form-data">
        <fieldset>
                <legend>Send these files:</legend>
                <input name="userfile[]" type="file" />
                <input name="userfile[]" type="file" />
        </fieldset>
        <input type="submit" value="Send files" />
</form>
```

Once the form has been submitted, the data is available in the `$_FILES` variable. I believe this is where the confusion usually sets in, because the data for both files is stored in the same `userfile` entry. What PHP provides in the `$_FILES` array looks like this:

```
Array
(
    [userfile] => Array
            (
            [name] => Array
                    (
                        [0] => 0001.jpg
                        [1] => 23.jpg
                    )

            [type] => Array
                    (
                        [0] => image/jpeg
                        [1] => image/jpeg
                    )

            [tmp_name] => Array
                    (
                        [0] => /usr/tmp/php/phpeQ7r6L
                        [1] => /usr/tmp/php/phpeWZnOW
                    )

            [error] => Array
                    (
                        [0] => 0
                        [1] => 0
                    )

            [size] => Array
                    (
                        [0] => 65730
                        [1] => 20959
                    )

            )

)
```

The problem is that the Upload library is not designed to handle multiple files and will crash when trying to verify that `$_file['tmp_name']` is an uploaded file, since it's expecting that to be a string instead of an array. So, what's the solution?

There are two possible solutions. The first would be to extend the Upload library and modify it to

handle multiple files better. While that might be a better long term solution, we don't always have the need for something that complex when a very simple solution is available. If you do create that solution, though, you should consider making the changes to the original library and submitting a pull request to the CodeIgniter team.

The simpler solution is to reconfigure the $_FILES array directly so the Upload library can handle the files separately in the form it expects in the first place. The following function will do that for you:

```
public function restructureFilesArray($files)
{
        $new_files = [];

        foreach ($files as $name => $file)
        {
                $file_count = count($file['name']);
                $file_keys = array_keys($file);

                $temp = [];

                for ($i = 0; $i < $file_count; $i++)
                {
                        foreach ($file_keys as $key)
                        {
                                $temp[$key] = $file[$key][$i];
                        }

                        $new_files[$name.$i] = $temp;
                }
        }

        return $new_files;
}
```

In your controller, before using the Upload library, you simply call this function, and it will create output that is structured like this:

```
Array
(
        [userfile0] => Array
    (
        [name] => 0001.jpg
        [type] => image/jpeg
        [tmp_name] => /usr/tmp/php/phpco72xp
        [error] => 0
        [size] => 65730
    )

        [userfile1] => Array
    (
        [name] => 23.jpg
        [type] => image/jpeg
        [tmp_name] => /usr/tmp/php/phpdQWJMU
        [error] => 0
        [size] => 20959
    )
)
```

Now that each of elements of the array has its own unique name and data, you can loop over them, calling the upload library on each element.

```
if (! empty($_FILES))
{
        $config['upload_path']   = APPPATH.'cache';
        $config['allowed_types'] = 'gif|jpg|png';
        $config['max_size']      = 100;
        $config['max_width']     = 1024;
        $config['max_height']    = 768;

        $this->load->library('upload', $config);

        $_FILES = $this->restructureFilesArray($_FILES);

        foreach ($_FILES as $file => $arr)
        {
                if ( ! $this->upload->do_upload($file))
                {
                        echo "Unable to upload {$file}.<br/>";
                }
```

```
                echo "{$file} uploaded.<br/>";
        }
}
```

8. Multiple Applications

CodeIgniter allows you to create multiple applications that all use the same `/system` directory. This has multiple uses, each with its own pros and cons. Later in this chapter we will look at how to setup a variety of different use cases and what configuration would be necessary to make the parts work together. Before we do, though, we need to talk about how to change the configuration of your application to allow any of these configurations to work.

Configuring Applications

The application's main `index.php` file contains most of the configuration options that need to be changed to make your applications work with a shared `/system` directory. This file contains the variables which specify where to find the `/application` and `/system` directories:

```
$system_path = 'system';
$application_folder = 'application';
```

The paths defined here are typically relative to the `index.php` file. In the default setup, all that is needed is the name of the directory, since the file and directory are in the same location. If you want to move the directories, though, you will need to either use full system paths, or provide other directory traversal information.

```
// Up one directory
$system_path = '../system';
$application_folder = '../application';

// Full Paths
$system_path = '/home/apps/system';
$application_folder = '/home/apps/application';
```

The location in which CodeIgniter attempts to locate your views may also be configured by setting a variable in the `index.php` file. The `$view_folder` variable is typically left blank, in which case the system will use the default location (usually `/application/views`), but you can set it to a location anywhere on your server. This can be used to share views between applications, or to isolate the views from the rest of the application (keeping the designers away from the developers' playground).

```
$view_folder = '/home/apps/views';
```

Using Packages

The remaining puzzle piece is the ability to use packages to setup a common directory to share code between applications. Packages were discussed in Chapter 2, so we won't go into much detail here on how to use them, but we will go over package configuration as needed in the examples which follow.

One reason to use a common location to share packages between applications is when a single web application is itself made up of separate parts. For example, the front-end and back-end of your application may be made up of separate CodeIgniter applications. A package could be used to store models or libraries which need to be shared between the two applications, ensuring that the application's business logic is maintained in a single location.

Since packages also provide access to views, you could store common themes in a single location, providing consistent branding between applications.

```
$autoload['packages'] = array('/home/common/themes');
```

Example Use Cases

In this section, we will look at some possible uses for multiple applications and how you would set everything up to work correctly. These are not the only possible uses, just some of the most common. If you have some creative uses that you've needed on projects in the past, shoot me an email, because I would love to hear them.

Multiple unrelated applications

Multiple unrelated applications could be run on the same server, sharing a single CodeIgniter installation. This results in less code to maintain on the server. However, it also means that every application must be tested with any updates to the framework before the new version can be installed. This would be structured something like this:

```
/application1
        /assets
        index.php
/application2
        /assets
        index.php
/system
```

While you could rename the `index.php` file for each application and keep them out of the `/application*` directories, it creates additional complications when trying to get visitors to the correct application. Structuring each application with the index file within the application directory is simpler, because no matter which web server you are using, you can simply point the domain to the application directory.

You would also store any assets for you site, like the CSS files, javascript files, images, etc, within each `/application` directory to keep them separated and contained in the application that they are a part of. Your index variables would be set like:

```
$system_path = '../system';
$application_folder = '.';
```

When doing it this way, you must ensure that you have the `.htaccess` file, or equivalent, within each `/application` directory to restrict access to files other than any assets or the `index.php` file.

This example was setup to have the server point to the `application` directory as its web root. If you would rather have each application as a directory under your domain, you would need to modify `/application/config/config.php` and set the `$base_url` to include your directory:

```
$config['base_url'] = 'http://mysite.com/application1';
```

Multiple CodeIgniter versions

Multiple CodeIgniter versions can be kept on the server to allow applications to be moved to the latest versions one at a time, only after they have been tested. If you find a flaw while testing your application with the new version, it is painless to set it back to the previous version. This option is best when switching to new major releases (2.x to 3.x) instead of point releases (2.2) as point releases typically do not have backward compatibility breaks. This would be setup something like:

```
/application1
        /assets
        index.php
/application2
        /assets
        index.php
/codeigniter2
/codeigniter3
```

This method is nearly identical to the previous setup, except that the `$system_path` variable must point to the the correct version.

```
$system_path = '../codeigniter2';
```

One additional thing to note here is that the `codeigniter2` and `codeigniter3` directories are simply the `/system` directory, renamed to reflect the version.

Separate Application Areas

Separate application areas can be maintained as separate applications to allow different teams to take control of each application without disturbing the other, and to isolate the applications. This has the added advantage that, if you need to move one application onto a separate server for performance reasons, you can very easily do so.

This is an ideal setup if you have sections of the website with completely separate concerns. One automotive site which I worked on had a section for the end users, a section for the dealers, and a section for the website administrators. By separating this into multiple applications, it was very easy to keep the business concerns separate.

This is also a perfect time to share some common code. Most likely, there will be common objects that are shared between the areas, though they might be presented differently to each area's users. You would want libraries and models that maintain the business logic in one place, but accessible by all of your applications. This is when packages come in handy.

Using the automotive site for an example, the directory structure might look like this:

```
/dealers
        /assets
        index.php
/users
        /assets
        index.php
/admin
        /assets
        index.php
/common
        /config
        /helpers
        /libraries
        /models
/system
```

Each application would need to have the path variables set as we've done previously:

```
$system_path = '../system';
$application_folder = '.';
```

In `/application/config/autoload.php`, we would need to ensure the `/common` directory was recognized as a package so that the shared config files, helpers, libraries, and models could be found.

```
$autoload['packages'] = array('../common');
```

Multiple Index files

A slight variation on the above patterns, is the ability to use more than one index file for a one or more applications. This can be especially handy when you are testing out your application on a newer version of the framework, but can be used to further customize the applications for different needs.

```
/application
/ci2
/ci3
index.php                // $system_path = 'ci2'
index2.php        // $system_path = 'ci3'
```

In situations like this, you might consider giving each index file it's own ENVIRONMENT setting to help you maintain separation between the configuration settings.

While the configuration options seem simplistic at first, they provide plenty of configuration for almost any need that you might have. You should always remember that these are not the only options available to you. Nor must they be used exclusively.

For example, if you have a shared system directory and find that one (or more) of the applications doesn't behave well on the latest version of CodeIgniter, you would most likely move the well-behaved application(s) to the newest version, while keeping the application(s) with issues on an older version. So, you might have started with a configuration which follows the first use case, but a new release gave you a reason to move to a configuration which matches the second use case. At the same time, you might have multiple applications sharing the same version of CodeIgniter, so those applications would still match the first use case, even though their system directory may be named "codeigniter3" instead of "system".

9. Security Considerations

Building secure applications is hard. Let's get that out of the way up front. It requires you to be constantly vigilant throughout the entire development process. Unfortunately, there is no simple, one-click answer to make your applications secure. However, there are some simple things we can do that will go a long way towards securing our applications.

In this chapter, we'll cover the basic things you should be doing. We will discuss good ways to use the security tools built into CodeIgniter, as well as some built-in PHP functions.

Security is complex, and constantly changing, so here are a few good resources to learn more about a variety of security-related topics:

- OWASP Guides[1] discuss many topics and provide details on how exploits work, best practices for protecting against them, and even how to test your application to ensure you're covered.
- PaulSec's Security Talks list[2] is a list of videos from several security-related conferences since 2014. A lot of topics are covered, and the list is updated as new conferences post their talks online.
- Building Secure PHP Apps[3] by Ben Edmunds gives good coverage of a number of security techniques from a non-framework-specific angle.

Validate Input - Filter Output

The most basic concept that you should be familiar with is that you should **validate input, and filter output**. This is the essence of developing secure applications. It's not everything, but it is the number one thing to think about while creating your applications. What does it mean? Let's look at each portion of this separately, along with some specific ways to protect your application.

Validate Input

Any input that comes into your application, that is not generated by the application itself, should be considered **untrusted**. Untrusted sources commonly include any input from the browser, including headers, cookies, GET and POST data, as well as any data read from files, received from the database, etc.

[1] https://www.owasp.org/index.php/Guide_Table_of_Contents
[2] https://github.com/PaulSec/awesome-sec-talks
[3] https://leanpub.com/buildingsecurephpapps

Ideally, the data should be validated everywhere it's about to be used. For example, if the user has submitted some data in a form, then that data should be validated in the controller, if possible. Then, if the data is sent to a model or library, the model or library should also perform validation. This helps protect the app in case the attacker found a way to get past the first round of checks. It also permits each portion of the code to perform specific validation according to its use of the data and its area of responsibility.

When performing validation, you should ensure that the data is of the expected type and the content meets expectations. Validate against a white-list, not a black-list. In other words, ensure that only known-valid data is passed, instead of checking for the existence of some possible bad characters. Let's look at a couple of examples.

Numbers should be one of the easiest things to validate, but we often get it wrong by ensuring it looks like a number through something like `is_numeric()`. This is a good first step, but you should also validate that the number is within a valid range, since you should always be able to determine the minimum and maximum acceptable values. If you don't accept negative values, you cannot simply assume that no negative sign means it's not a negative number. Some number systems will convert a number into a negative if the number is large enough to overflow the allowed maximum value for that type.

Strings can be a little trickier, but the essence is the same: determine what's valid and limit the permitted input accordingly. If the string has an expected pattern, use regular expressions to check that the supplied string matches the pattern.

Remember that data read from a file or database should be considered input just as potentially dangerous as that from the user/browser. Validate input from your database just as you would data from a form, ensuring that it meets your expectations and is safe to use in the manner you will be using it.

Filter Output

As we just discussed, the data will generally be untouched when you need to use it. It is possible that it will have `<script>` tags, or encoded text that might hold dangerous elements when presented in specific contexts, like outputting it to HTML. The phrase "Validate Input, Filter Output" is a little deceiving, but only because we typically think of it as outputting to the browser. There's more to it than that, though. You should filter the data prior to any use of it. A "use" might include:

- Saving it to the database
- Displaying it in the browser
- Using it within a CSS style, or a javascript script
- Inserting it into an email
- Outputting into a file
- etc.

In each of these cases, the way you need to escape the data is slightly different because there are different characters which cause issues in different situations. For example, you can save something with a dangerous `<script>` block in the database without worrying about it doing anything bad. However, if you display the same data within the browser, it could do any number of things. In many cases, the `xss_clean()` function should be fine when filtering output for the browser, though a context-specific library like Zend Framework's Escaper[4] is much better. At least for displaying output in HTML emails, the browser, etc. To the best of my knowledge, most text is safe within a file, but should be filtered when pulled back out of the file, especially if it is displayed to the user in a web page or HTML email. When saving to databases, you would use database-specific escaping tools, which will be discussed a little later.

Since the topic of security is such a broad one, and this book is specifically geared towards CodeIgniter, this section will limit itself to how to use the `xss_clean()` function when building your web pages. You are encouraged, though, to start learning more about security on your own every chance you can get.

CodeIgniter provides a pretty good tool, `xss_clean()` that is intended to protect against common attacks against cookies, Cross Site Scripting (XSS), and others. While it does a pretty good job, it is not perfect. Nothing ever is, as new attacks are being created or discovered daily and the team can never keep up with all of them. That said, it's still a pretty good tool, and, if all you do is use that function on your output, the security level on your site will have dramatically increased. Just don't fool yourself, your client, or your employer into thinking they are 100% secure. No one can promise that.

The question then becomes, **when** should you filter the information? If this is used as your primary protection, then it should be done in the controller, after you've pulled the information from the database, validated it from a user-submitted form, etc. but before being sent to the view. This provides a single place to filter the data, making it simpler to ensure the data actually gets filtered. It's too easy to forget a location in the views, especially if the data is used in several locations.

```
// An abbreviated example using POST data
$foo = $this->input->post('foo');
. . .

$this->load->helper('security');

$data = [
        'foo' => xss_clean($foo)
];
$this->load->view('some_view', $data);
```

One thing to note in this example, is that you must load the **Security Helper** in order for the `xss_clean()` function to be available.

[4] http://framework.zend.com/manual/current/en/modules/zend.escaper.introduction.html

```
// An abbreviated example using the Database
$rows = $this->some_model->find_all();

foreach ($rows as &$row)
{
        $row->foo = xss_clean($row->foo);
}

$this->load->view('some_view', ['rows' => $rows]);
```

It should be noted that the `xss_clean()` function takes a bit of processing power, so you should consider using the Cache library[5] in production to keep you from having to do that on every page view.

Validation Tools

CodeIgniter provides a great Form Validation library[6] that can be used in your controllers or models. It has a number of built in validation routines, and you can create custom callback routines to handle any other special needs that you might have. This is a must-have tool for your validation. By default, it will use the `$_POST` data, but you can set any data to be validated using the library's `set_data()` method.

In addition to that library, PHP has the excellent `filter_var()` function, and its friends. While `filter_input()` seems like an obvious thing to use, if you don't know the exact server environment that your app will be running in, I recommend sticking to `filter_var()`, since there are some issues with `filter_input()` when running under FastCGI.

The `filter*` methods are able to validate a number of different types of data, depending on the filter type[7]. The validation filters take care of a lot of the tiny details that you might otherwise forget, and they're fast, which is always good. Among the things they can validate are:

- boolean values
- email addresses
- floating point numbers
- integers
- IP addresses
- MAC addresses
- URLs
- custom regular expressions

[5] http://www.codeigniter.com/user_guide/libraries/caching.html
[6] http://www.codeigniter.com/user_guide/libraries/form_validation.html
[7] http://php.net/manual/en/filter.filters.validate.php

You should take the time to become very familiar with these filters, as they are some of the best, and easiest to use, validation tools you have. They can also be used with the Form Validation library by calling them within your own validation functions/callbacks.

When using the Form Validation library, it's tempting to clean up – sanitize – the string at that point. Best practices say that you should only do that prior to actually using the data. This could mean by properly escaping prior to saving in the database, or cleaning up as it's being output into an HTML page. The reason for this is that each use of the data has different needs and it is difficult to accurately predict all of the uses of the data. Some characters may be perfectly valid when simply used as a data source - and might even be a crucial part of that data - but can cause problems when output as HMTL. At some point in the future, the application might be modified to provide outout in JSON or XML formats, where the original data might be needed intact. If you've cleaned it prior to saving it to the database, then that original data source is no longer available to you.

There are times, however, where you know for a fact the original data is not desirable. In these cases, it is perfectly fine to clean the data up using the Form Validation library's rules. A common example of this is accepting user input through textareas or input fields. Most of the time, you know that you don't want HTML to be available ever. In times like this, using `strip_tags` is perfectly valid, and can reduce your storage requirements, and, when done in the model just prior to saving, can help plug some potential security holes in your application. Think carefully about the data prior to committing to changing the data at the validation phase.

For this reason, I recommend not using `xss_clean` as a validation rule any more, though I must admit that I've been guilty myself of attaching that rule to all text input fields in the past.

CSRF Protection

Alongside XSS attacks, which we just covered, CSRF attacks are among the most common that you will need to protect against. Lucky you, CodeIgniter has some great protection for those built right in.

What are CSRF Attacks?

Cross-Site Request Forgery, or CSRF, attacks attempt to trick the user into doing something they didn't intend to do. This is often done through fake emails or websites that make the user think they're using the main site, but are actually doing what the attackers want them to do. This could be as simple as changing their password in the background to something only the attacker knows, stealing the password during a password change action, forcing them to transfer money without their knowledge, and many, many other possibilities.

How to protect against them?

The most effective method to protect against the attacks is to ensure that the request comes from your site. This is done by creating a temporary code that is only valid for a short amount of time

that must be submitted with your forms. This code is checked on the server to ensure that it matches and is a valid token for that user and that form.

Turning on this protection is simple to do in CodeIgniter. Simply turn it on globally in `/application/config/config.php` and it will be checked for every non-GET request. Which means it also helps to protect any RESTful APIs that you build.

```
$config['csrf_protection'] = TRUE;
```

Now every non-GET request is checked to ensure that the token exists and matches. Which leaves one question. How does it know the current token? The simplest method is to use the `form_helper` to open and close your forms. This will automatically insert the CSRF code in the form for you, which will be submitted as a hidden input with the form itself.

```
// In the controller
$this->load->helper('form');

// In the view
<?= form_open('/path/to/form/controller') ?>

// Creates a hidden input:
<input type="hidden" name="csrf_token" value="{csrf_token goes here}">
```

CSRF and AJAX

What if you cannot, or don't want to, use the form_helper? No problem. You can use the `get_csrf_token_name()` and `get_csrf_hash()` methods to build your own input, or create javascript variables that can be passed back in your AJAX calls.

The problem that you can run into when using AJAX calls, though, is that you could hit a moment where the user has been on that screen long enough that the current CSRF hash has been marked as invalid. This can happen from time spent on the site (or they had to walk away from the computer for a few moments). Additionally, the default setting is for the hash to be regenerated on every request. In this case, your first AJAX request would be fine, but any additional requests from that same page would have an invalid hash, since it's been regenerated. This can be turned off in `/application/config/config.php`.

```
$config['csrf_regenerate'] = FALSE;
```

This will keep the hash good for as long as the cookie is valid for. Check your cookie settings in that same config file to see the lifespan of a cookie.

Expected External Forms

What if your application has a valid form that will be submitted that comes from an external site? Obviously you don't want to turn off CSRF completely. Instead, you can whitelist URLs to exclude from CSRF protection. Back to the config file again for this setting.

```
$config['csrf_exclude_uris'] = array('api/person/add');
```

You can also use regular expressions to build up a more flexible list. Just be sure that your list of endpoints is restricted as much as you can so that you don't accidentally open up security holes.

```
$config['csrf_exclude_uris'] = array(
    'api/record/[0-9]+',
    'api/title/[a-z]+'
);
```

In short - always enable CSRF protection for the good of your users. When you get to a point that you're seeing issues, one of the last couple of settings should be able to fix it. Just remember - whitelisting is a last resort, and should only be used after careful consideration.

Database Security

The biggest concern you have when saving data to the database is to watch out for **SQL injection attacks**. CodeIgniter makes this pretty simple.

What Is an SQL Injection Attack?

Hackers use SQL attacks to attempt to get or create information they shouldn't have access to from a database. They could use an un-sanitized forgotten password page to map out bits of your database schema, or potentially get information about other users on the system, guess common elements, like ID, usernames, or passwords to find accounts they can access, etc. The attack is successful because of a single quote, literally.

For example, an attacker could abuse the simple query used in a forgotten password page to retrieve the user and make sure they're valid. The original query might be:

```
SELECT id, email FROM users WHERE email = '{$email}';
```

Since the single quote represents the end of a variable, they could include a single quote of their own and append their own checks, starting with something simple like:

```
// Attacker submitted: anything' or 'x'='x
SELECT id, email FROM users WHERE email = 'anything' or 'x'='x';
```

In this case, the query will always be true because of the *x=x* portion and would return all members in the system. Obviously not what we want to have happen. For a fascinating example of abusing this exact situation to find more and more information, take a few minutes to read over SQL Injection Attacks by Example[8].

How to Protect Your App?

The number one most effective strategy is to ensure that everything that goes into the database that came from somewhere outside of that class gets escaped in a database-specific way. If you're using mysql, then PHP's built-in *_mysql_real_escape_string* is your best friend. Having to do it become tedious, though, and easy to make mistakes, so it's best if it's done automatically. The good news is that CodeIgniter makes this dead simple.

If you use the Query Builder for your queries, you're in the clear, since all of those functions will automatically escape the data for you. Job done.

If you need to build queries manually, you have a couple of different tools at your disposal. The first one is actually two different functions built into the database driver that both escape a value for you.

```
$foo = $this->db->escape($foo);

OR: If you're using it in a LIKE query
$foo = $this->db->escape_like_str($foo);
```

The only difference between the two is that the latter one works with the special characters of a LIKE query (%) and won't mangle the query for you.

The second method, and that one that I suggest using any time that you build your query manually, is to use the parameter bind features of the database driver. This works much like prepared statements that are available in a number of databases, and makes them work across all of the databases that CodeIgniter supports.

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? AND author = ?";
$this->db->query($sql, [3, 'live', 'Rick']);
```

This works by replacing all of the question marks in the raw SQL with the elements of the array sent in the second parameter of the `$query` method.

As long as you use these methods for ALL of your database work, then your site will be protected from probably 99% of all database-related attacks that are likely to happen on your site. Nice, and painless, just the way I like it.

[8] http://www.unixwiz.net/techtips/sql-injection.html

Logging As Security

It might feel strange to talk about logging in a chapter on security, but it makes sense when you think about it. Logging has many uses, from ensuring things are running correctly, to watch for slow queries, or - the part we're most interested in right now - tracking your users throughout the application.

An essential part of security is being able to determine who did what at any place where something sensitive might happen. Common things that you should look at logging is:

- when objects are created, updated, or deleted (especially users, but any sensitive info)
- when people log in, log out, change password, fill out the forgot password form, etc.

By tracking those bits of information, you should be able to determine what users could have done something suspicious. There are two ways to log this information: with the objects in the database, or through the standard logging functions.

Database Objects

A simple way to track who modified information in the database is to provide `created_by`, `updated_by`, and `deleted_by` fields in the object's database tables, that each hold the last user to do that action on that object. You could modify your MY_Model class to automatically add this information in for inserts, updates, and deletes. That way it's always tracked and you never have to worry about remembering to do it for every call. And having things happen automatically is the best way to make any type of security happen.

The drawback to this method is that it only tracks the last person to modify it. For many types of objects, this is probably fine. If you need more security, you would need to create another table to keep track of history of changes. You could still have your MY_Model do it automatically for you, though the initial coding of MY_Model options might be a little more complex. It is time well spent, though.

For a very detailed talk about what all you should be logging in your application and why, take 41 minutes to watch this video by Stelian Mocanita at the BG PHP Conference in 2015, called Tame your application with logging and monitoring[9]

Manual Logging

CodeIgniter provides the `log_message()` function that allows you to log anything you want to its log files. It has several different levels of severity that can be used when you log information, but there's only one level that's practical to use for this: `debug`. While the `info` level is tempting to use, the framework itself will put a lot of log entries here for every class that is loaded. That quickly fills up a log file and becomes a lot of noise to wade through when you need to find something.

[9] https://www.youtube.com/watch?v=7IJAWmzl_4k

```
log_message('info', "User {$user_id} ({$username}) logged in.");
```

To ensure that your information is getting logged, you need to edit the `/application/config/config.php` file to ensure that logging is turned on.

```
$config['log_threshold'] = 2;  // Only debug and error messages
```

10. Performance Tips

It would be wrong of me to start this chapter with anything but a reminder that you should never optimize your application prematurely. In other words, get the app running first. Get some customers coming in the door. When you start to see problems, that is the time to optimize. Often times, you'll find bigger gains to be found by scaling and optimizing your hardware than your application.

That said, there are a few simple things you can do within your application that help improve the speed of site your site dramatically. Before we look at the CodeIgniter-specific things you can do, let's look at some of the issues you should be aware of outside of CodeIgniter.

The Three Quickest Performance Gains

While optimizing your application outside of CodeIgntier is not what this book is about, and it's too big of an area, I need to briefly discuss the easiest things you can do to see quick performance gains. I won't go into great detail here about how to work with these, but there are ton of excellent resources out there that can help with the details.

A great resource is the book, "Scaling PHP Apps[1]", by Steve Corona. This details everything that he and his team had to do to scale TwitPic up to serving in excess of 22 billion HTTP requests per month. It's chock full of detailed information about how to tweak your servers, how to organize them, running proxies, and pretty much everything else that you need to know to get more *oomph* out of your application. I highly recommend this book to anyone that wants to get serious about scaling.

Opcode Caching

The first thing that you should be doing for any site, is to turn on OpCode caching for PHP. This creates huge performance gains since it removes the compilation phase of reading your scripts into memory and saves that pre-compiled opcodes for later use. In PHP 5.3 or lower, the recommendation was always to use APC or XCache to handle this. Since PHP 5.5 the Zend OpCache has been bundled into PHP, and and extension is available for php 5.2, 5.3, and 5.4.

OpCache can be turned on within the `php.ini` file with the following setting, as long as the extension has been loaded:

[1] https://leanpub.com/scalingphp

```
opcache.enable=1
```

Refer to the manual for all of the possible settings[2].

Application-specific Caching

Application caching can take several forms, but the goals are always the same: save the results of expensive operations so they can be reused later without. You might cache entire pages so you don't even have to hit the majority of your application, or you might cache fragments of results after they've been converted into HTML. Depending on your cache naming system, you might even cache elements based on roles, like if you need to display some admin-specific tools on pages.

This one is actually discussed in more detail below, since CodeIgniter has built-in caching capabilities. Three different types, actually!

Proper Indexing of the Database

With that out of the way, we need to look at properly indexing your database. By far, the biggest performance bottleneck in your application is going to be your database, and poorly designed and indexed queries are show-stoppers. Unfortunately, there's no way that I can teach database design here, and I'm probably not qualified, anyway. But there is one tip I can give you regarding indexing: **index your joins**.

The first thing to realize is that indexing must be properly handled. Simply adding indexes to every column is going to be worse for performance, and create a much larger database.

Second, ensure that your tables have primary keys. Primary keys are the one column that will always be unique in that table. Creating an index on the primary key means that looking up a piece of data by it's ID, for example, can happen almost instantly, since the database doesn't have to scan through every row looking for the right data.

Next comes the big helper: indexing joins. This is where the biggest performance gains can often be seen once a table grows past tiny. Indexing creates another invisible table, if you will, that the database can very quickly scan to see relationships between data. Whenever you join a table, the database has to somehow locate which data matches up to the data you're joining it with. This is often done through the use of primary keys in the other tables, but not always. Without an index of some sort, it must scan the tables for every row it is joining. The goal is to minimize that as much as possible.

You should try to make sure that the columns you are joining on have proper indexes. Primary keys count as an index. If your schema has foreign keys, they should probably be indexed.

Finally, though, remember not to optimize prematurely. It is often best to wait until you see the application working to see which queries are slowing it down, then optimize the slow queries that are are used the most often first to avoid unnecessary bloat of your database.

[2] http://php.net/manual/en/opcache.configuration.php

Database optimization is a complex subject, and very heavily dependent on the database system you are using, and how it is used in your application. This has just scratched the surface and you're encouraged to explore the subject in more detail.

Autoloading Strategies

CodeIgniter provides the handy ability to autoload helpers, libraries, models, and more through `/application/config/autoload.php` file. This is extremely handy for the developer, but can be highly detrimental to your application's performance.

Any items that are automatically loaded on every request, regardless of whether or not they were used. If you have many items in here, you are simply wasting server resources and slowing your application down. Instead, you should only put items in here that are crucial to the operation of the application as a whole. And really think hard about each item.

For me, there's only 4 items that might get autoloaded in my typical project:

- **url helper** - because I always want `site_url()`, `base_url()`, and the other functions available to ensure any links I create function correctly no matter what the server setup is.
- **language helper** - needed if there's any possibility that I'll use language features to set UI text.
- **application config** - most of my projects have a new configuration file that contains application-specific items. If used, I will always autoload this so there's never any question whether it's available or not, and the types of settings it stores is used very frequently.
- **application language file** - if language functions are used, I will have a single, global, `application_lang` file that includes common strings used throughout the application.

And that's about it. It's tempting to have things like the database or the session libraries autoloaded, but they're not used on every page. Items like that can happen as needed in the controllers.

MY_Controllers

Every project I work on will make use a MY_Controller class extension, like we've discussed previously. This is the perfect time, though, to tackle some of those items that you think might need to be global. If you use a heirarchy or base controllers (like MY_Controller, ThemedController, AdminController, etc) then you can more easily fine-tune what gets loaded when.

In the list of controllers I just mentioned, MY_Controller would handle nearly site-wide items, like caching, etc. This allows you to have flags setup as class properties that can decided whether that resource is needed on a per-controller basis. During the constructor, it can check the flags and load what is needed.

ThemedController would extend MY_Controller, loading up the theme engine, setting default layouts and theme, etc. Because, truthfully, even in a web application, not every request needs to display a themed page. Some might be responsible for handling cron jobs, others might be generating RSS feeds, or handling API requests, etc. Again, flags can be used, along with custom methods, to allow you to change the default view or layout used, add/remove view data, etc.

AdminController would extends from ThemedController, and load up the authentication system, ensure that the current user has the appropriate permissions to be there, and sets the admin theme.

In each new controller, we're loading only what is needed for those situations. Theming libraries are only loaded when you intend to display web pages. Authentication libraries are only loaded only when you know you'll need to authenticate someone. And so on. Even though every single method in that controller might not need those libraries, if the majority of methods do, then this is an excellent trade-off between performance and ease of use.

Controller Constructors

Often, you will have files that are needed throughout most of the methods in a controller, but not anywhere else in the site. This is most often seen with resource pages, where an Article controller would need the ArticleModel and the article_lang files everywhere in that controller. This is the perfect time to load the files up in the controller's constructor. While you could add simple auto-loading functionality to your MY_Controller, it is often simpler, and easier to understand down the road, to just load them up in the constructor.

```
class Articles extends ThemeController {
        public function __construct()
        {
                parent::__construct();

                $this->load->model('article_model');
                $this->language->load('article');
        }
}
```

In Method

Finally, don't be afraid to load any resources as you need them in your methods. It's the best way to handle it in CodeIgniter, since it doesn't have any true class autoload capabilities.

```
public function show($id)
{
        if (! is_numeric($id))
        {
                throw new InvalidArgumentException('Bad ID. No cookies.');
        }

        $this->load->model('article_model');
        . . .
}
```

If every method ensures that the resources it needs is loaded, you might see multiple load calls to the same resource, if not carefully watched. However, the performance impact when you do this is truly minimal and not something to become too concerned about.

Caching

The goal of any performance tweaks are to make sure you can access the cached data faster than you can retrieve it normally. There's very few hard and fast rules of when you should cache, and how you should cache, since your application and the server's that it is running on will determine much of that. For example, if you have a slow hard drive that is overused currently, you might find it's actually slower to cache a large number of items. Instead, you might look at caching in something like Redis, since it's in-memory, but if your Redis server is a different server and you have a high latency between servers, you might lose all performance benefits.

With those disclaimers out of the way, though, let's take a look at the types of caching that CodeIgniter provides, since it can drastically speed up your site, if used well.

Full-Page Cache

The fastest cache that CodeIgniter provides is the full-page cache[3]. This will cache the an entire response and return it very early in the bootstrap process, so it never loads up most of the framework, instead, displaying the rendered and saved response, typically the entire HTML page. While it's not as fast as pure HTML, it's pretty close and allows you see some pretty dramatic speed ups.

If you're building a site in a PHP framework, it's safe to say that the site is going to have a number of dynamically generated parts to it. These could be recent posts, leaderboards, articles, or any number of other elements in your page that change on a frequent basis. So, you might think that using a full-page cache is not possible on your site. But, you just might be wrong, especially if your site is under a heavy load. Let's say you're getting 1 hit per second on your site on average. Even if you were to have your full-page cache set to a minute, that means that only 1 in every 60 requests has to

[3] http://www.codeigniter.com/user_guide/general/caching.html

load up the entire framework and hit the database. For a site under heavy load, that can be a huge savings in server resources. I can't think of many sites that need to have the data be more accurate than within a minute. Most sites could have the full-page cache longer, anywhere from 15 minutes to an entire day, depending on how fresh the site needs to be. If you have a blog, you could cache it for weeks or even months, and then ensure the cache was deleted whenever a new blog post was published.

Using the full-page cache is simple. Anywhere during that page's processing, you call the Output class' `cache()` method:

```
public function index()
{
        // Cache for 1 day
        $this->output->cache(1440);

        // Do stuff here
        $this->load->view('my_view', $data);
}
```

Outside of how fresh your data needs to be, the other thing to consider is whether or not you have any functionality present on the static pages that varies based on if a user is logged in, etc. Sometimes you can code the app to get around this, but many times you cannot, and there's too much data to cache the entire page. In these cases, you would need to look into using the Caching Drivers[4].

Cache Drivers

The Cache Drivers allow you to save any information you might need to. This might be HTML fragments, data that was expensive to pull from the database, or to process, or even connections information retrieved from third-party websites that may take a long time.

I recently used it on a third-party API to cache the results from their 2 required calls that we needed to get the basic information about their service, including customized URLs for other endpoints, media types supported, etc. On the production server, caching this single item took the web page speed from 1.8 seconds down to 0.07 seconds. Because of the type of information it was, we felt it was safe to cache for a day at a time. That's a huge savings on server usage, but also much better for the end-user who is frequently un-willing to wait around for 2 seconds.

Again, the type of caching, and even if caching will help, is something that must be measured and considered for each site on its own merits. Once you decide that caching is needed, though, you have several cache engines, or drivers, that you can use.

- Alternative PHP Cache (APC)

[4] http://www.codeigniter.com/user_guide/libraries/caching.html

- File System
- Memchached
- Redis
- WinCache

Your choice of which driver to use will depend on the nature of your application's server setup. Some engines, like Memcached and Redis, work best when they live on a separate server than the application since they will use all available memory, which can impact your site loading, making these more appropriate for medium-to-large scale sites. Others, like the file-system, can be used to good advantage on any site, though slow or over-taxed hard drives can limit their uses at times.

Because caching is such a tremendous help, I almost always have the driver setup in the MY_-Controller class, with configuration settings in an `/application/config/application.php` config. This allows me to ensure that the cache drivers are always loaded and available for use. On my local development server I keep the cache driver set to `dummy` so that it always misses. This way, development can happen like normal, with no confusion when the cache hits. The, using Environments, production and staging servers can have their own caches setup. Staging servers might use a file-based cache for simplicity, but still giving us the effect, while the production server might have a Memcached driver loaded.

The configuration settings look something like this:

```
$config['cache_type']           = 'dummy';
$config['backup_cache_type']    = 'dummy';
```

Then, in MY_Controller, we ensure the proper driver is loaded and ready for use.

```
protected function setupCache()
{
        // If the controller doesn't override cache type, grab the values from
        // the defaults set in the start file.
        if ( empty( $this->cache_type ) )
        {
                $this->cache_type = $this->config->item( 'cache_type' );
        }
        if ( empty( $this->backup_cache ) )
        {
                $this->backup_cache = $this->config->item( 'backup_cache_type' );
        }

        // Make sure that caching is ALWAYS available throughout the app
        // though it defaults to 'dummy' which won't actually cache.
```

```
        $this->load->driver( 'cache', array( 'adapter' => $this->cache_type, 'backup' =\
> $this->backup_cache ) );
}
```

In this example, MY_Controller also has class properties allowing each controller or method to override the type of caching used, if necessary.

During development, there are also times when there's no expectation of having a cache engine running. However, you'll often get the feeling that a few methods are going to be potentially slow. In those cases, you can still check the cache for a hit, even if the cache is only ever set to the `dummy` driver. Then, down the road when your client calls you up and complains about the speed of the application, and you've ensured your database is indexed properly, you can simply change the cache driver to something else and look like a hero in your client's eyes. Always a good thing, right?

Russian-Doll Caching

One of the more extreme, though thorough, methods I've seen used for caching was by 37 Signal's Basecamp team, doing what they called Russian Doll Caching[5]. (More details here[6]). During a major redesign of their site, they decided to cache *everything*, and they do it in multiple layers. It might first check the the whole page. If that cache is expired, it will try to load large segments of the page from the cache engine. In each of those, they might be smaller sections, and even smaller cached sections within those. And so on. Just like the nesting Russian dolls[7].

Cache Naming

What you call your cache can make huge difference in how effective your cache system can work. The biggest thing I want to bring up here is a use for it that I hadn't considered until I read an article a few years ago. The trick was prefixing the current user's role name onto the cache name. This is especially helpful for times where there are portions of the page that are only available to a single role, like tools for admins. By prefixing `admin_`, or `mod_`, or even `member_`, to the name of the cache, you can still have a dynamic site but take advantage of the cache engine also.

Database Caching

The final cache type that you have access to is to cache the results of a query. This is especially useful for any slow queries that don't change much. This only works for SELECT type queries, not for writes, which only makes sense. Instead of hitting the database, the results of the query are stored as text files on disk, so you need to be aware of the performance of your hard disks under load before determining for sure if this technique is useful.

[5] https://signalvnoise.com/posts/3112-how-basecamp-next-got-to-be-so-damn-fast-without-using-much-client-side-ui

[6] https://signalvnoise.com/posts/3113-how-key-based-cache-expiration-works

[7] https://en.wikipedia.org/wiki/Matryoshka_doll

We'll run through a quick example to show how it works. For this example, we are dealing with comments on a blog post. Since the comments very rarely change, it makes sense to take a little load off of the database here. The first thing to do is to wrap our existing query in `cache_on()`, `cache_off()` method calls to determine what gets cached.

```
public function post($post_id)
{

        $this->db->cache_on();
        $query = $this->db->where('post_id', $post_id)->get('comments');
        $this->db->cache_off();

        return $this->load->view('blog/post', ['comments' => $query->result()], true);
}
```

Ignoring the fact that this method is incomplete, the results of `$query` are stored in a file on disk the first time that it's called. Any remaining times that it's called the results are read from the file, and the database is never touched. Where is it stored, though?

Files are stored within the directory specified in your database configuration's, `cachedir` value. Within that, they are organized further by the first two URI segments for the current request. This will generally be the name of the controller and method called. If this page was accessed at `example.com/blog/posts/5`, the directory structure would look like this:

```
{cachedir}
        blog/
                posts/
                        // cache files go here
```

Each query is saved under its own file. The name is determined by an md5 hash of the query itself. This means that a new file would be saved for every post's comments, since the query hash would change as the post_id value changes. This is something to watch out for since a query that varies by user could create hundreds or thousands or cache files very quickly.

These cached files will exist forever, as there's no system in place to automatically expire the cache. This means that it's up to you to delete the files as necessary. In our comments example, you would delete the cache whenever a new comment on that post is made.

```
public function add_comment($post_id, $comment)
{
        if (! $this->comment_model->insert(['post_id' => $post_id, 'comment' => $commen\
t]) )
        {
                return false;
        }
        // Clear the cache
        $this->db->cache_delete('blog', 'posts');
}
```

This would delete all of the cached files for that controller and method.

By taking advantage of these three layers of caching, you can easily create high-speed output for your users. You do need to be sure to test that the caching is helping and not hindering your speed, though, before committing.

Database Tweaks

While much information can be cached, there are times when you cannot cache the results because it's simple data processing that needs to be done. In these cases, there are a few CodeIgniter-specific database tricks that can help speed up your data processing.

Stop Saving Queries

By default, the database engine will save all of the generated queries for a log. This is then used by the profiler and allows you to get the last query or even pull a list of all queries. In a production environment, these features are often not needed. This can free up a lot of memory during intensive data processing tasks and stop you from running out of memory.

You can tell the database engine not to save the queries by setting the `save_queries` setting in your database configuration to `false`.

Don't Return Insert ID

Often, we'll return an insert id automatically in methods we create in our custom MY_Model classes. This is extremely convenient, and completely fine in most cases. When you're processing millions, or even thousands, of rows, it can be a performance hit you don't need, since it executes an additional query against the database for each insert, doubling the number of queries executed. When do that kind of intense processing, ensuring that your script can function without getting the insert id can be a huge gain.

Persistant Connections?

Allowing persistent connections are often seen as a way to speed up your database performance, but that's not always the case. They are really only good in times when it takes a long time to make the connection to your database. With modern computers, this is almost never the case, anymore. When it is the case, it is usually something different that's causing the connection lag that should be discovered.

Whenever you use persistent connections, it has the potential to negatively impact performance because the connection is held on to while other users are waiting on an available connection to finish their request.

My recommendation is to always keep persistent connections off unless you can truly make a case for it.

11. Fun at the Terminal

CodeIgniter has never been known for it's command-line support. Unlike many frameworks today that make extensive use of Composer for the core portions of the framework, CodeIgniter's CLI tools are pretty weak. While we will likely never have the equivalent of Laravel's `artisan` CLI tool, we can still take advantage of the command line for our own scripts.

Why would you want to work with scripts on the command line? There's a wide variety of uses, including:

- Creating cronjob scripts that work with all of your application's models, libraries, etc.
- Create cli-only tasks for running backups, changing the site status, pruning cache folders, etc
- Create scripts that can be called from other languages, allow Python, Ruby, or even C programs to make use of your existing logic and models by simply calling your script.

Running CLI Scripts

If you're not familiar with running CodeIgniter scripts from the command line, you'll be delighted to know that it is very similar to using it from a browser. Controller methods can be called in almost the exact same way that you'd make a controller/method to run from the browser. There are three primary differences in the way that you access the script:

There are no domains On the command line, there are no domain names. There's no localhost. You're simply running a PHP script directly from the terminal.

Always use index.php While most of the URL's to our web applications are designed to remove `index.php` from the path to make it look nicer, on the command line you will always need to use it. Without creating a different cli-only script, you can't get around needing the index.php file in your path.

Segments use Spaces Where URI segments in your browser are separated by forward slashes, on the command line they are treated as separate words, or arguments.

If you would access a portion of your application in the browser at `http://example.com/index.php/users/show`, you would access the same script at the terminal like this:

```
> php index.php users show
```

So this simple command simply says to run the `index.php` script, using PHP, and pass the `users` and `show` values into the script as arguments. From there, CodeIgniter takes over, treating those arguments as URL segments. From there's it's business as usual. Almost all of the skill you have creating code for browsers in CodeIgniter will still apply. There are only a couple of differences to be aware of.

CLI Differences

Sessions

When writing methods for the command-line, the biggest difference is that the Session library is not available. When it is loaded it will simply log a message that it cannot run under the command line, and then exit. If you try to access the session at any point you'll get and error that stops execution.

Different Errors

If you customized your error messages previously to provide nicer looking errors for your users, you don't need to worry about those customizations breaking your error messages on the CLI. They use different error templates. For CLI requests, the error templates can be found under `/application/views/errors/cli`. You might never need to modify those. I've found it handy, though, to remove the backtrace at times for smaller error messages, since I'm often using the terminal from a small window within my IDE.

Restricting Access

One thing to be aware of is that any methods that you create for CLI use is still accessible from the browser by default. Usually, you don't want these scripts to be accessible from the browser. To restrict access to only from the CLI you can use the `is_cli()` global function to check if it's being called from the CLI and then throw an error.

```
class SomeController extends Controller
{
        public function __construct()
        {
                if (is_cli())
                {
                        show_error('Unauthorized Access.');
                }
        }
}
```

Exit Codes

CodeIgniter has several status codes that it uses to provide information to other scripts that are watching. This is especially handy if your script has been called from another script or language. The status codes are provided as constants that are always available, allowing you to use them in your scripts, also.

```
EXIT_SUCCESS                    0               // No errors
EXIT_ERROR                        1               // Generic error
EXIT_CONFIG                     3               // Configuration error
EXIT_UNKNOWN_FILE           4               // File Not Found
EXIT_UNKNOWN_CLASS       5              // Cannot find the class
EXIT_UNKNOWN_METHOD   6              // Cannot find the class method
EXIT_USER_INPUT                7              // Invalid user input
EXIT_DATABASE                   8               // Database error
```

While CodeIgniter will use these codes itself, you are encouraged to use them in your own scripts, also.

```
if (! file_exists($path))
{
        log_message('Cannot find file: '. $path);
        exit(EXIT_UNKNOWN_FILE);
}
```

Simple cronJob Runner

One of the frequently needed CLI-type features is the ability to easily run tasks at a scheduled time. This might be processing an email queue every 5 minutes, or maybe ensuring that old records are purged from the database once a day. There are many uses for this type of script, but it's something that CodeIgniter doesn't ship with. So, let's build our own that you can easily re-use in any program.

Overview

Before we get started, we need to look at the solutions we need to come up.

First, it needs to be able to run tasks on a regularly defined time periods. For the sake of this script, we'll break the tasks down into 4 categories: frequent (every 5 minutes), hourly, and daily. This takes care of most needs, though you might occasionally need a task to run every minute. There are a number of ways that this could be solved, including a simple list of librarys and the methods to run in order, but for the sake of this example, we are going to use Hooks. We haven't used them elsewhere in the book, and they provide a flexible way to link multiple modules into it.

Next, we need to be able to keep track of the history. We will keep things simple and just track the last run times for each of these, but you could easily modify the script to track run-time duration, output and more.

The Database

You will need a simple table in your database to hold the last run times of our different cron jobs. This table will only hold a single record in this example. You could easily modify it for your needs, though. The SQL to generate the table is as follows:

```
CREATE TABLE `cron` (
  `id` int(1) NOT NULL DEFAULT '1',
  `frequent` datetime DEFAULT NULL,
  `hourly` datetime DEFAULT NULL,
  `daily` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The Controller

Like any request, CodeIgniter needs a controller built to handle the logic, so that's where we'll start. It's primary job is to fire the correct hook at the right time, and not run them more often than we need to since, depending on the task, that could cause very real problems, like running subscription charges too frequently. First, we'll post the entire class here, and then we'll go over how it works.

CRON Controller

```php
<?php

class Cron extends CI_Controller
{
        /**
         * Holds last run times for
         * @var array
         */
        protected $runs;

        //--------------------------------------------------------------------

        /**
         * Script entry point. Ensures the correct
         * hooks are fired at the correct times.
         */
        public function run()
        {
                $this->runFrequentTasks();
                $this->runHourlyTasks();
```

```
21                  $this->runDailyTasks();
22          }
23
24          //--------------------------------------------------------------------
25
26          protected function runFrequentTasks()
27          {
28              if (! $this->shouldRun('frequent', 5)) return;
29
30                  $this->hooks->call_hook('cron_frequent');
31
32                  $this->updateRuns('frequent');
33          }
34
35          //--------------------------------------------------------------------
36
37          protected function runHourlyTasks()
38          {
39                  if (! $this->shouldRun('hourly', 60)) return;
40
41                  $this->hooks->call_hook('cron_hourly');
42
43                  $this->updateRuns('hourly');
44          }
45
46          //--------------------------------------------------------------------
47
48          protected function runDailyTasks()
49          {
50                  if (! $this->shouldRun('daily', 1440)) return;
51
52                  $this->hooks->call_hook('cron_daily');
53
54                  $this->updateRuns('daily');
55          }
56
57          //--------------------------------------------------------------------
58
59          /**
60           * Checks the time since the last run of $name, as stored in
61           * the database.
62           *
```

```
63          * @param $name
64          * @param $interval        Number of minutes after last run that we should run.
65          *
66          * @return boolean
67          */
68         protected function shouldRun($name, $interval)
69         {
70                 $this->loadRunData();
71
72                 // If it's never run before, than go ahead and run it now.
73                 if (! isset($this->runs[$name]) || is_null($this->runs[$name]))
74                 {
75                         return true;
76                 }
77
78                 // If the last run time plus our interval (in minutes)
79                 // is less than or equal to now, the job should run.
80                 if (strtotime($this->runs[$name]) + ($interval * 60) <= time())
81                 {
82                         var_dump($name);
83                         return true;
84                 }
85
86                 return false;
87         }
88
89         //--------------------------------------------------------------------
90
91         /**
92          * Loads the last runs from the database.
93          */
94         protected function loadRunData()
95         {
96                 if (is_array($this->runs)) return;
97
98                 if (empty($this->db)) $this->load->database();
99
100            $query = $this->db->limit(1)->get('cron');
101
102                $row = $query->row_array();
103
104                if (! count($row))
```

```
105                 {
106                         $row = [
107                                 'frequent' => null,
108                                 'hourly'   => null,
109                                 'daily'    => null
110                         ];
111                 }
112
113                 $this->runs = $row;
114         }
115
116         //---------------------------------------------------------------------
117
118         /**
119          * Updates the last run times in the database.
120          * @param $name
121          */
122         protected function updateRuns($name)
123         {
124                 $data = $this->runs;
125
126                 $data[$name] = date('Y-m-d H:i:s');
127
128                 $this->db->replace('cron', $data);
129         }
130
131         //---------------------------------------------------------------------
132
133 }
```

Since we store all of the latest run times in the database, we need a class property to store the data after we pull it from the database so we don't have to hit the database more times than neccessary. The `$runs` class property will store this.

The main entry point into the script is the `run()` method, which simply calls out to other methods to run our different frequency tasks. It would have been possible to clean up the class a little more and refactor the three `run*` methods into one, but that would make things a little more difficult to understand, and the code is simple enough here I kept them separate.

Within each of the `run*` methods, `runFrequentTasks()`, `runHourlyTasks()`, and `runDailyTasks()`, it first checks if the tasks should be run or not. If not, we return and move on to the next run method. If it has been long enough since the last run of this cronjob, then we call the hook. Finally, it updates the database with the current time this one was ran.

The `call_hook()` isn't really documented, since hooks were originally intended to be just used to extend the system, but they make a decent generic event system that you can use throughout your own applications. The only thing you have to do to use them is to call `$this->hooks->call_hook('hook_name')` , replacing 'hook_name' with anything that describes your hook action or placement.

In the `shouldRun()` method, we first load our last run times from the database so that we have something to compare to. If the frequency matching `$name` is null, then we know it's never been ran, so we go ahead and tell the calling function to do its first run. If a last run exists, we check to see if the last run time + the number of minutes specified in `$interval` is less than or equal to the current time. If it is, we know we need to run the tasks, otherwise it's not yet time.

And that's pretty much all there is to the controller. It's a very simple setup, but one that works quite well. The next step is to define the tasks to run.

Defining the Hooks

In order to define the tasks that need to be ran at each interval, we need to define the hook. This is described in depth in the CodeIgniter user guide[1], but I'll provide an example here for a refresher.

Open up **/application/config/hooks.php**. By default, there are not any hooks defined so we need to define the first one. Create a new entry that looks something like this:

```
$hook['cron_frequent'][] = array(
    'class'    => 'Welcome',
    'function' => 'index',
    'filename' => 'Welcome.php',
    'filepath' => 'controllers',
    'params'   => array('beer', 'wine', 'snacks')
);
```

It's fairly obvious that this hook wouldn't actually do anything we wanted, but it shows the two main things you need to be aware of.

First - the name of the hook ('cron_frequent') must match the name you used in the `call_hook()` method back in your controller, otherwise they will never actually be called.

Second - you need to make sure to add the details as a new array element so that you can have multiple actions for each hook that is called.

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

And that's all it takes to make a simple, but effective, cron runner for your applications. Simply create a new hook entry for every task that you need ran on a repeating schedule and you're good.

The only thing this didn't detail is setting up the cron script to point to this script and call it. This can vary from system to system, but on Linux you'd use **crontab** files, while you'd use the Task Scheduler on Windows systems. In both cases, they should point to this file, whether through the command line or through a URL, and call it whatever your most frequent task is ran. In this example, you'd want the script called every 5 minutes so that you can run the frequent tasks as often as they're supposed to be called.

12. Composing Your Applications

Unless you've been hiding, or are brand new to development, you've probably heard of Composer[1]. It is a package-management tool that allows you to easily pull in third-party libraries and all of their dependencies, and easily keep them updated. Several popular PHP frameworks even use it at the core of their architecture, building on top of other, battle-tested libraries. I think it's fair to say that this library has single-handedly changed the PHP ecosystem for the better. Instead of everyone re-creating the same packages over and over, we can now share in the hard work that others have done, in a way that simpler, and more effective, than ever before.

At its heart, Composer has a very flexible autoloader that is PSR-0 and PSR-4 compatible, making lazy-loading of packages simple. By adopting this common autoloader method, it makes it possible for all any package that wants to follow it also be able to be found by any application that adopts these standards also. With Composer, that seems to be just about everyone.

Though there are ways to create your own repositories of packages for Composer, most of the packages that you'll ever use can be found over at Packagist[2]. When I last checked, there were over 100,000 individual packages available.

In this chapter, we'll first look at how simple it is to use third-party packages within our applications. In the last half of the chapter, we'll use Composer to structure our applications in a way that allows us to separate our application from the framework a little, helping us ride out framework changes, and even major application changes.

Using Third-Party libraries

The first step to use Composer in your applications is to ensure it's installed. You have two options here, either globally or per-project. I recommend globally, personally. Full details can be found at Composer's silte[3].

Tell CodeIgniter to Use It

Now we need to let CodeIgniter know that we want to use Composer. It has support built in. Open up `/application/config/config.php` and scroll down to find the `composer_autoload` setting. You have two options about how to turn it on. The first option is simply to set the value to `true`. This way is potentially more secure than the other method, but does expect the `vendor` folder that Composer

[1] https://getcomposer.org/

[2] https://packagist.org/

[3] https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx

creates to be within the application folder. The second option is to provide a string that is the path to the `autoload.php` script itself. We'll demonstrate how to make the first version work for you while still keeping it convenient to use.

```
$config['composer_autoload'] = TRUE;
```

Composer.json

Next, we have to create a configuration file that tells Composer what packages should include. This is a json file that must be named `composer.json`. Place this in the root of your application, right next to the application and system folders. For this example, we're going to load the `ramsey/uuid` package. It allows you to create uniqued IDs for the items in your project. Our bare-bones file would look something like this:

```
{
        "require": {
                "ramsey/uuid": "3.4.*"
        },
        "config": {
                "vendor-dir": "application/vendor"
        }
}
```

The "require" block contains all of the packages that you want to use, along with the version number to use. Specifying a version number is important. If you simply include "dev-master" or something like that, you never know when they are going to change something that breaks your application. In this example, we've specified that we want any minor updates to the 3.4 branch of ramsey/uuid.

The "config" block has a number of options, none of which you'll need to mess with until your setup gets very complex or different, with the exception of the "vendor-dir" setting. If you don't specify this, Composer will create a "vendor" directory in the same location as your composer.json file. For many projects this is fine. For a standard CodeIgniter file structure, though, it's not ideal, because we don't want to make it easy for potential attackers to look into what resources we're using, looking for attack vectors. So, we place it the application folder which you've hopefully locked away from prying eyes through .htaccess rules, or something similar.

Now, at the command line, run the following command and you'll be set:

```
composer install
// or
php composer.phar install
```

This creates the `/application/vendor` folder and downloads the packages you've specified.

To use the packages that you've downloaded is as simple as creating a new instance of the main class, in many instances. The package will have specific directions, of course, but you don't need to worry about loading the file first. Composer takes care of autoloading it when you instantiate it. For this library, it would be used something like this in your controller:

```
use Ramsey\Uuid\Uuid;

public function index()
{
        $uuid = Uuid::uuid4();
    echo $uuid->toString() . "\n"; // i.e. 25769c6c-d34d-4bfe-ba98-e0ee856f3e7a
}
```

Restructuring Your App

How do you normally build your applications? If, like most of us, you make use of all of the features of the framework, separating things cleanly between the controllers, models, and libraries folders, you might not realize it, but you're tying yourself very closely to the framework. That might seem obvious when it's said, but is it a problem? It might be in the long term, so let's take a second and look at what the implications are, and why you might want to restructure your app. Don't worry, this is very relevant to Composer.

Is Framework Dependence Bad?

To some extent, your application is always going to be tied to the framework that you use. There's no getting around that. With any luck, your applications are going to live for many years on the web. What happens when a new major version of the framework comes out that changes the structure or even basics about how the framework works?

Don't shrug it off. I've heard a number of stories from frustrated development teams that had apps built on Symfony 1 or Zend 1. Sometimes they were lucky enough to stay with the whole project, other times they took over a legacy project at their company. Both of those frameworks had pretty big compatibility-breaking changes between version 1 and 2. The developers were often stuck looking for advice on how to fix an issue with a framework that had been dead for a couple of years. It was hard to find answers to their problems, since all of the help out there was now about the newer versions. Third-party libraries had been modified to work with the new versions, dropping support for previous versions, so there were no bug-fixes or security patches. Laravel users also know the pain, because there are quite often BC-breaking changes, sometimes not even a year since the previous major version came out.

CodeIgniter 4 is coming down the road, and it's a complete re-write of the application that is definitely changing things up.

You can never fully protect yourself, but hopefully you can see that the wise move is to separate your business logic from any application-specific code. This keeps the mission-critical parts of your application sheltered from any changes should you need to upgrade the framework, or - worse yet - switch to another framework. Of course, certain portions of your application will always be tightly coupled with the underlying framework, like the Controllers and the models. You can take steps to minimize their affect, though.

Take Command of the Application Folder

The first step toward independence is to realize that the application folder is just a suggested structure. With Composer, you can easily add a namespace to the application folder, creating new folders to take advantage of any different design patterns or organization that you might want to use. Open up `composer.json` again, and assign an `App` namespace to the application folder:

```
{
        "autoload": {
                "psr-4": {
                        "App\\": "application/"
                }
        }
}
```

What this says is that any class in the `/application` folder or its child folders can be found in a namespace. So, controllers could live in `App\controllers` namespace, models under `App\models`, and so on. Any new folders that you add will automatically be under the App namespace. The catch here is that the case of the namespace must exactly match the case of the directories. This does look a little strange, since namespaces are typically all UpperCamelCase. You can fix this by adding additional, more specific entries to composer.json:

```
{
        "autoload": {
                "psr-4": {
                        "App\\Controllers\\": "application/controllers",
                        "App\\Models\\": "application/models",
                        "App\\Libraries\\": "application/libraries",
                        "App\\": "application/"
                }
        }
}
```

This has an added benefit if you have a large amount of classes: it could actually improve your app's performance. When Composer tries to locate the files it has to look through all of the directories

and scan all of the files to find it. By splitting the namespaces up into different directories, there is less scanning of the file system and better performance. For smaller applications with less files, you might see a slight hit on performance, but it shouldn't be too much. I remember reading a post-mortem where the company did this one change, and manage to increase performance by 25%-33%. Not too shabby.

Version Your Application

Many applications can benefit from a little future-proofing by versioning the app itself. This is especially true of API's, but is not specific to them. Most applications change over the course of their life and being able to work on one version while keeping the other for the main users. Due to limitations in CodeIgniter, you cannot version the controllers or the views since the framework would never be able to find the files, but we can freely move the models and libraries around. Since this where the bulk of our business logic should live, this works pretty nicely.

To start, you might change the application folder to something like this:

```
/application
        /controllers
        /views
        /v1
                /models
                /libraries
```

Now, we change the namespace location of the models and libraries in composer.json:

```
{
        "autoload": {
                "psr-4": {
                        "App\\Controllers\\": "application/controllers",
                        "App\\Models\\": "application/v1/models",
                        "App\\Libraries\\": "application/v1/libraries",
                        "App\\": "application/"
                }
        }
}
```

Now, whenever your create a new instance of `App\Models\UserModel` it will look for it in `/application/v1/models`. If you ever need to switch it over, you just change the path in composer.json and you're set.

This is just one idea of how you could use it. Play with it. Experiment. Find what works for you.

Start Namespacing

At this point, you can still use CodeIgniter exactly as you're used to, but I wouldn't stop there. You should actually namespace your models, libraries, and others. Doing this breaks the pattern of calling `$this->load->model('foo')` or `$this->load->library('bar')`, so why bother? This forces you to remove this dependence on CodeIgniter's super-object. This makes it much simpler to switch out down the road, because you're now using standard PHP functions to load the files, which work in any framework.

```
class User {
        public function index()
        {
                $model = new App\Models\UserModel();

                $this->load->view('users/index', [
                        'users' => $model->getAll()
                ]);
        }
}
```

Of course, the `load->view()` portion is tied to CodeIgniter. But what would it look like if you switched to Laravel? Pretty close:

```
class User {
        public function index()
        {
                $model = new App\Models\UserModel();

                return view('users.index', [
                        'users' => $model->getAll()
                ]);
        }
}
```

Obviously, it would still take a fair amount of time to move frameworks, but that helps make your controllers, at least, much easier to convert.

Use Libraries

A common phrase you'll hear among the developer community is, "Thin Controllers, Fat Models". I agree with this to some extent, but I'm of the opinion that both the controller and model should be

thin. Then where does all of the necessary code go? In Libraries. In this model of working, the model just provides ways to transfer the data to and from the database. The libraries would contain all of the business logic related to one entity type, like a blog post, an article, a financial transaction, etc. The library would be responsible for interacting with the model at this point, leaving the controller to call on the libraries. Following the previous advice, the library would namespaced, so that you don't need to grab the superobject. At this stage in our journey, your code might look something like:

```
// The controller
class Users extends CI_Controller
{
        public function index()
        {
                $user_lib = new App\Libraries\UserLibrary();
                $users = $user_lib->getActiveUsers();
        }
}

// The library
namespace App\Libraries;

class UserLibrary
{
        protected $model;

        public function __construct()
        {
                        $this->model = new App\Models\UserModel();
        }

        public function getActiveUser()
        {
                return $this->model->where('active', 1)->get('users');
        }
}
```

There are more pieces here, but each piece is insulated from the framework a little bit more. If you need to change out how you store the users, perhaps moving it to a third-party service, all you need to do is create a new model with the same methods as your current model, and load that one instead. The library will only need to load up the correct model, and the rest of your application can stay the same.

Even better, base your models off of an Interface[4] that you've designed to ensure they always stay the same. Since you can create new folders in your application folder now, and have the classes easily found through the App namespace, you might consider creating a new Interfaces folder:

```
/application
        /controllers
        /interfaces        <-- Store all interfaces here, under App\Interfaces namespace
        /libraries
        /models
        /views
```

Inject Dependencies

The previous example was a definite improvement, but we can do better. The problem we're left with is that the library in that example had a hard dependency of the specific model class, since it was instantiated in the library's constructor. To make things even simpler, we'll use Dependency Injection, something you're surely heard of, but might have gotten confused in all of the talk about difrerent containers and how you'd integrate them into the framework, etc. Don't let it scare you off, though, becuase it's a very simple concept, and there's no DI container needed.

All Dependency Injection means is that you pass in any dependencies to the class, creating them outside of the class first. This allows to you specify exactly what class you're using and, as long as they're both implementing the same interface, the library doesn't need to know what the exact class is. It will know that it can depend on any class it gets handed to have the same methods it can use.

Here's how we'd convert the previous example to use this methodology:

```
// The Interface
namespace App\interfaces;

interface ModelInterface
{
        public function find_where($key, $val);
}

// The model
namespace App\Models;

use App\Interfaces\ModelInterface;

class UserModel extends CI_Model implements ModelInterface
{
```

[4] http://php.net/manual/en/language.oop5.interfaces.php

```
        public function find_where($key, $val)
        {
                return $this->db->where($key, $val)->get('users');
        }
}

// The Library
namespace App\Libraries;

use App\Interfaces\ModelInterface;

class UserLib
{
        protected $model;

        public function __construct(ModelInterface $model)
        {
                $this->model = $model;
        }

        public function getActiveUsers()
        {
                return $this->model->find_where('active', 1);
        }
}

// The Controller
use App\Libraries\UserLib;
use App\Models\UserModel;

class Users extends CI_Controller
{
        public function index()
        {
                $user_lib = new UserLib( new UserModel() );
                $users = $user_lib->getActiveUsers();

                $this->load->view('users/index', ['users' => $users]);
        }
}
```

There you go. Now your application is an application that *users* the CodeIgniter framework, instead of being a CodeIgniter application that is completely reliant the framework for everything. Now, if

CodeIgniter is every not maintained for a few years and your boss is worried about security patches, etc, you can make the move to a different framework much less work becuase things are buffered from the framework. All of the business logic is contained in the Libraries which don't have any reliance on the framework at all. You could swap out CodeIgniter's database layer for Eloquent, Doctrine, or PHPActiveRecord and just change how the models work with the database, and your entire application keeps on working.

So, not only does it bring your development style up to more modern practices, but it creates code with much less technical debt that's easier to maintain, easy to test, less fragile, and will fare better for longer without any additional effort on your part. Sounds like a win for the small amount of up-front planning you need to do.

Explore Design Patterns

There's one last thing that using Composer within your applications allows you to do: explore different design patterns. In the years that CodeIgniter has been around, many new design patterns have come over from different languages to start being accepted as good practices to follow. Of course, not every design pattern is needed, but having the flexibility to arrange your application however you want no longer restricts you to the simplistic view that CI uses, however nice and elegant a solution it seems to be. Of course, no design pattern is a magic bullet, either, but you now have the flexibility to explore.

For example, look at the Repository Pattern[5]. This pattern basically says that you have a Repository that is a collection of items. Where and how those objects are stored are an implementation detail. The Repository's sole job is to fetch objects for us, and to persist them. Sounds kind of like a Model, right? You could use a model, but it's better, as we saw above, to leave the persistance layer separate from the Respository. So, the Repository is a simple class that manages the collection, and it uses a Model to persist the data somewhere. That somewhere could be to a file on disk, into the database, in a Redis store, or a third-party web service.

Along with the Repository, you have the Entity class. This is the Library that we used previously to represent our object. The only difference being that an Entity only knows about itself, and the parts that make itself up. Whereas the Library example above had the Library also being resposible for retrieving the objects from the database. In the Repository pattern we flip that around so the Repository retrieves the LIbrary instances.

This is now simple to implement. As a matter of fact, we already implemented a simplfied version of it in Chapter 5: Working With Data, in the example showing how to use Custom Result Objects. The only difference from a "pure" Repository pattern and what we implemented was that we merged the persistance and repository layers together into one layer: the model. So, I'll leave it as an excercise for you to implement the changed version. Between that chapter and this one, you have all of the information you need.

With Composer to help you find classes, you can even be extrememly clear in your application structure:

[5] http://shawnmc.cool/the-repository-pattern

```
/application
        /controllers
        /entities
        /models
        /repositories
        /views
```

13. Whew! We made it.

We are at the end of our journey here. Thank you so much for purchasing a copy. I only hope that it gave you lots of bits of inspiration and drive to change up the way that you use CodeIgniter, pushing your code better every day.

Just because we're at the end of the book, though, doesn't mean your journey is over. Oh, no. Indeed, it's just starting. This is the point where you take what you've learned and implement it in your current project. Or your next one if you're too far along in the current one to change things up. With every project you'll refine how you use the tools at your disposal. You'll find new ways to twist those tools to your bidding. New ways to write clearer, more maintainable code. New ways to make things easier for you and those that might work on the project in the future.

So, go forward and create. And enjoy doing it!

Again, thanks so much for joining me on this ride and trusting me with your time. I truly appreciate it.

Lonnie