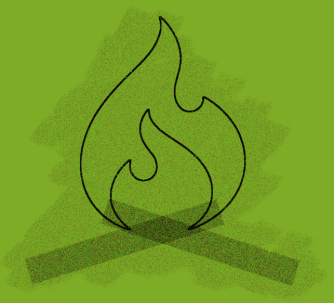
jamie rumbelow's
the codeigniter handbook



volume one

who needs ruby?

efendi
publishing

Jamie Rumbelow

THE CODEIGNITER HANDBOOK

Volume 1 - Who Needs Ruby?

Who Needs Ruby?

Copyright ©2012 Jamie Rumbelow

All rights reserved. No part of this book may be reproduced without the prior written permission of the publisher, except for personal use and the case of brief quotations embedded in articles or reviews.

Every effort has been made to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, Efendi Publishing nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Efendi Publishing has endeavored to provide trademark information about all companies and products mentioned in this book, however we cannot guarantee that this information is 100% accurate.

CodeIgniter® is a registered trademark of EllisLab, Inc. CodeIgniter® logo copyright 2007 - 2012 EllisLab, Inc., used with permission.

First Published: 2012

This Edition Published: 2012

Published by **Efendi Publishing**

ISBN: 978-0-9571791-1-0

For Mum, Dad and the Alexes. I couldn't ask for better people in my life.

Table Of Contents

Acknowledgments i

An Introduction to The CodeIgniter Handbook ii

An Introduction to Volume One iii

Who should read this book? iv

Part 1 - Models

- Fat Model, Skinny Controller 3
- Standard Model Conventions 4
- Observers 16
- Scoping 20
- Validation 26
- MY_Model 32

Part 2 - Views

- Presenters 34
- Partials 42
- Fragment Caching 46

Part 3 - Controllers

- Autoloading Views 54
- Layouts 59
- Autoloading Models 65
- Filters 68

Part 4 - REST

- HTTP: The Forgotten Protocol 73
- CodeIgniter-friendly semi-RESTful Routing 77

Summary 81

Acknowledgments

A huge thank you to my fantastic editors, Hope Doty and Dean Howe. Without your tireless efforts the book would probably be full to the brim with overly-pretentious, badly spelled locutions and horrendously broken code samples. Many thanks go to the love of my life–the ever-gorgeous and wonderfully talented Laura Parrott–for her fantastic illustrations that provide a much needed visual break between the reams of otherwise plateauing textual boredom. Thanks to Charlie + Matt over at Print GDS for being so understanding and helping me destroy the planet, one dead tree at a time. Cheers to the EllisLab team for CodeIgniter and allowing us to use their logo. Thanks to the CodeIgniter Crew: Phil Sturgeon, Elliot Haughin, Tom Myer, Jeremy Gimbel, Erskine, Colly, Five Simple Steps, A Book Apart, Simon and Angela Campbell and the EE lot.

Thanks to me Mum and Dad for keeping me sufficiently sober and adequately drunk, respectively. Lots of love to my Granny who passed away before this book could be published. Love to my friends: Alex, Alex, Izzy, Claudia and everyone else, and Chris, Bella, Pat, the Lambs + Riot, Andy and Jo in Bath for looking after me.

I would also like to thank you, dear reader, for providing me with enough disposable income to buy an Aston Martin. Naturally, I'm joking. Nobody became rich from writing a tech book. Just ask Tom Myer.

But a guy can dream, right?

An Introduction To The CodeIgniter Handbook

I've been programming with CodeIgniter for six years. When I first downloaded the source code and dug into what would soon become the heart and soul of my professional career, I had an epiphany. Suddenly, I had found a framework that made sense from the word 'go.' It took me all of fifteen seconds to download the source and extract it into my *Sites* directory. From that point onward, my life as a developer had dramatically changed.

What really made CodeIgniter special was the sense of excitement that it invoked. If you've ever had the misfortune of meeting me, you know that the excitement is still there. I'm so passionate about CodeIgniter; it's visible on my face and in my body language. Just as it is visible in the faces of every other developer I've ever met who has experienced the same epiphany.

I assume you're reading The CodeIgniter Handbook because you're passionate about CodeIgniter. You're excited about the future of the framework. But more important, you're excited about the present. About the sheer promise that CodeIgniter can bring to your applications. Despite all the hype around other frameworks and languages, CodeIgniter still remains the greatest PHP framework for any developer developing real-world applications. CodeIgniter provides pragmatism without over-abstraction, speed without simplification, power without bulk. It's the perfect mix of programming happiness and scalability. It's eclectic.

An Introduction To Volume One

Ruby on Rails–and its programming language Ruby–have taken the web development world by storm. In just eight years they've managed to accumulate a huge community, powering massive sites all across the web including Twitter, Shopify and Basecamp. They are both widely praised for promoting elegant syntax, rapid development speed and the controversial Convention Over Configuration pattern.

What's the secret to their success? Why can't us CodeIgniter developers have a slice of the Rails pie?

In this short, concise, succinct book I'll demonstrate that, not only can we can eat the pie, but it's tasty too. We'll take a look at the basic concepts that make Rails developers happy and how you can implement them in CodeIgniter in a CodeIgniter-friendly way. We'll explore the concept of Convention Over Configuration and what simple steps can be taken to make writing reliable applications in CodeIgniter *really, really easy*. We'll look at tidying up your code by studying design principles like Don't Repeat Yourself (DRY), and using RESTful style controllers to provide a consistent URL pattern across the application.

It might be opinionated, but that doesn't mean the techniques we'll learn can't be flexible. I sincerely hope that you'll find the concepts and ideas in this book make programming in CodeIgniter fun. Really, really fun.

Who should read this book?

If you're a novice to CodeIgniter, this book is for you. If you are an experienced CodeIgniter developer, this book is for you. As long as you understand the core concepts of Model-View-Controller and can code up a simple app, you're bound to get something valuable from this text. I'll assume that you understand the basic differences between libraries, helpers, models and views and how to use them within CodeIgniter.

This book was written with CodeIgniter version 2.1.0. Any code examples here are tested to work with this CodeIgniter version, not any prior or future versions. With the rapid pace of development, I can make no guarantee that code will work on a future version. If you'd like to follow along, please use 2.1.0.

models
PART 1

...in which we take a look at the M in MVC, learn what's best to go in the model and how powerful the model layer can, and should, be. We'll examine an effective way to improve your models across the board, as well as look at some Rails-inspired patterns to make coding your models more efficient. We'll also discuss some commonly made errors in the CodeIgniter world and look at how best to fix them and implement better solutions.



The Model-View-Controller (MVC) design pattern teaches us a series of rules for building more robustly structured applications. The most important of these is that models store all the code related to processing data. This tells our models that they are in control of changing data or state. CodeIgniter's implementation of MVC is rather loose, insofar as it allows developers to bypass the model layer and directly interact with the database in the

controller layer. Its implementation (and the CodeIgniter standard library) also promote some bad practices, such as placing validation logic in the controller. In reality, the 'M' bit in MVC means that any code that stores data to the database, validates data, sends out emails, connects to APIs, or calculates statistics *needs* to be put in the model layer.

This whole idea revolves around two concepts from the Rails world: fat model, skinny controller and convention over configuration.

Fat Model, Skinny Controller

This idea tells us that we need to write our applications *around* the logic in our models. Here comes the first sweeping statement of, I'm sure, many to come: *any code that you can't justify putting anywhere else should go in the model*.

Our models should be fat. They should be the juicy chicken of the traditional British Sunday lunch that is our application; stuffed to the brim with the application-specific stuffing, tasty, meaty and fat. Our controller is the gravy. It's lightweight, it's thin, it's succinct. Don't get me wrong, it's totally necessary; the gravy binds the whole meal together and provides common ground for the meat and the trimmings to communicate on the plate. However, it shouldn't be the part of the meal you focus on. It's not the centerpiece.

Carrying on this analogy, the view is the trimmings. The veg, condiments and the pigs in blankets; the things that fill up the plate, look wonderful, and make us drool at the sight of it.

I'll stop now, for I'm getting hungry, but I hope the principle is clear. The model should be what drives the application, not the controller. It should be the thing that makes everything run. We depend on the controller to bring it together, and we depend on the view to present that data to the user in an

intuitive and friendly way, but it's the model that really does the work. It's because of the chicken that we sit down to eat at all.

It's also important to remember that the data might not be coming from a database. It might be coming from an API, or a file on the server, or the user's session. It might even be going into a different database than it comes from. Decoupling this means that the same interface is used across the application to interact with the data, whatever the data source. And this decoupled nature becomes immensely useful as your application grows.

Standard Model Conventions

Arguably the most driving design decision behind Ruby on Rails is Convention over Configuration–the idea that establishing a set of conventions across all projects is better than requiring lots of configuration and changes. It means that developers need to make less decisions; developers only specify unconventional aspects of the application.

In practice, when looking at models, we make a bunch of assumptions about our database-backed models & their corresponding database tables. This means we can write less code, our application is consistent across the board, and we can abstract a bunch of functionality away.

Without further ado, here is *Jamie Rumbelow's Spectacular List of Model Conventions*.

- Each database-backed model maps to a single database table
- The model name is a singular, the table name is plural
- Each table will contain an `id` column
- Each table will contain both a `created_at` and an `updated_at` column

These are all reasonable assumptions to make, and by making them, we can improve the quality and conciseness of code in our model layer. Let's go

through them, one by one, and take a look at some examples of how to implement them.

Each database-backed model maps to a single database table

As with any rule, there are exceptions (which we'll look at in a moment). For the majority of cases, however, each database-backed model will primarily interact with one database table.

Chances are we have a bunch of methods that interact with our database table in various ways. They're also probably going to include our standard roster of Create, Read, Update and Destroy (CRUD) methods. Because of this, we're going to be specifying the database table across the class.

```
public function get($where)
{
    return $this->db->where($where)
                    ->get('users')
                    ->row();
}

public function get_all($where)
{
    return $this->db->where($where)
                    ->get('users')
                    ->result();
}

public function insert($user)
{
    return $this->db->insert('users', $user);
}
```

```php
public function update($where, $user)
{
    return $this->db->where($where)
                    ->update('users', $user);
}

public function delete($where)
{
    return $this->db->where($where)
                    ->delete('users');
}
```

Simply put, we don't have to specify this across the class. Instead, we can use an instance variable to specify the table once. That then makes it easy to change the table, if we so wish, without having to re-code the whole model.

```php
class User_model extends CI_Model
{
    protected $_table = 'users';

// ...

$this->db->get($this->_table);
$this->db->insert($this->_table, $user);
$this->db->update($this->_table, $user);
$this->db->delete($this->_table);
```

I'm calling it `$this->_table` so we don't conflict with CodeIgniter's table library.

As well as specifying the table across most–if not all–of our models, chances are we're going to have this aforementioned standard roster of methods in them too. In lieu of replicating these, we can extend CodeIgniter's `CI_Model` and create a `MY_Model` class. This ensures we're abiding by another important Rails principle, *Don't Repeat Yourself* (DRY).

All of our models can extend this `MY_Model`. By extending `MY_Model` we'll get a bunch of these basic CRUD functions baked in for 'free'. Create a new file in your **application/core** directory called **MY_Model.php**.

```
class MY_Model extends CI_Model
{
    public function get($where)
    {
        return $this->db->where($where)
                        ->get($this->_table)
                        ->row();
    }

    public function get_all($where)
    {
        return $this->db->where($where)
                        ->get($this->_table)
                        ->result();
    }

    public function insert($data)
    {
        return $this->db->insert($this->_table, $data);
```

```
    }

    public function update($where, $data)
    {
        return $this->db->where($where)
                        ->update($this->_table, $data);
    }

    public function delete($where)
    {
        return $this->db->where($where)
                        ->delete($this->_table);
    }
}
```

CodeIgniter will load this for us. We can then extend our models from `MY_Model`, remembering to specify a table:

```
class User_model extends MY_Model
{
    protected $_table = 'users';
}

class Post_model extends MY_Model
{
    protected $_table = 'posts';
}

class Category_model extends MY_Model
{
```

```
    protected $_table = 'categories';
}
```

...and now we can access the three database tables in a consistent, elegant fashion, without having to duplicate code.

The model name is a singular, the table name is plural

Did you spot a pattern across those models we just wrote?

Each model had a singular name (appended with *_model*), and each table was the plural of that singular name. What a great convention to have! It's simple and it means that we can auto-guess the table name based on pluralizing the singular model's name.

Let's write some code to guess that. We can put it in our `MY_Model`'s class constructor, which will ensure that it gets run whenever we load the model.

```
public function __construct()
{
```

Make sure we call `CI_Model`'s constructor too–we need to access CodeIgniter!

```
parent::__construct();
```

We can now load CodeIgniter's inflector helper, which contains a bunch of useful functions for dealing with English language strings.

```
$this->load->helper('inflector');
```

Finally, we can use `get_class()` to fetch the class name, remembering to get rid of the _model, and pluralize it. We should only try to guess the table name if we have not set it already. That way, if we have to specify an unconventional name, we can.

```
    if ( ! $this->_table)
    {
        $this->_table =
strtolower(plural(str_replace('_model', '',
get_class($this))));
    }
}
```

The `plural()` function is pretty sophisticated. It'll successfully pluralize most words. If it's struggling to pluralize a word, we can easily override it by specifying it directly in our model.

Now, we can get rid of the explicit table definition from our models:

```
class User_model extends MY_Model { }
class Post_model extends MY_Model { }
class Category_model extends MY_Model { }
```

Delicious.

Each table will contain an id column

Another assumption we can safely make is that we'll be mainly using the `id` column to uniquely refer to an individual row in our database table. With this assumption in mind, we can slightly rework the previous CRUD methods we wrote in the last section. Let's change our `get()` method a little:

```
public function get()
{
    $args = func_get_args();

    if (count($args) > 1 || is_array($args[0]))
    {
        $this->db->where($args);
    }
    else
    {
        $this->db->where('id', $args[0]);
    }

    return $this->db->get($this->_table)->row();
}
```

This allows us to pass through multiple `WHERE` conditions if we choose, as well as just passing through our ID column.

We'll duplicate this method and call it `get_all()`, changing the last line to return the `result()` rather than the `row()`:

```
return $this->db->get($this->_table)->result();
```

We can also adjust the `insert()` method to return our new ID:

```
public function insert($data)
{
    $success = $this->db->insert($this->_table, $data);

    if ($success)
    {
        return $this->db->insert_id();
    }
    else
    {
        return FALSE;
    }
}
```

The call to `$this->db->insert_id()` will mean we'll get the newly inserted ID as a return value from the `insert()` method.

Finally, let's adjust our `update()` and `delete()` methods in similar ways:

```
public function update()
{
    $args = func_get_args();

    if (is_array($args[0]))
    {
        $this->db->where($args);
    }
    else
```

```
        {
            $this->db->where('id', $args[0]);
        }

        return $this->db->update($this->_table, $args[1]);
    }

    public function delete()
    {
        $args = func_get_args();

        if (count($args) > 1 || is_array($args[0]))
        {
            $this->db->where($args);
        }
        else
        {
            $this->db->where('id', $args[0]);
        }

        return $this->db->delete($this->_table);
    }
```

...and just like that, the ID is now set as the default means of unique row identification. Let's take a look at a bit of controller code that will interact with our models:

```
    public function test()
    {
        $id = $this->user->insert(array( 'username' =>
    'jamierumbelow' ));
```

```
    $user = $this->user->get($id);

    $this->user->update($user->id, array( 'username' =>
'jamierumbelow' ));
    $this->user->delete($user->id);
}
```

It's a short, concise syntax based on common conventions. It goes to show that by using basic techniques and following basic principles, you can seriously improve the quality of your code.

Each table will contain both a created_at and an updated_at column

This is an interesting one. While you might not always *need* to know when a row in your database table was created or updated, it can still be useful information. If something goes wrong internally, it can be really useful to know when a certain resource was created or changed. Additionally, it can be helpful to know these dates and times when you're exporting data into other systems, trying to maintain a cache or fetching data from remote places to populate your database.

While these factors may not be the case the vast majority of the time, when you need them, you'll be glad you put them in. Plus, with the other conventions in place, keeping and maintaining this information is simple.

Let's add a couple of lines to our `insert()` and `update()` methods:

```
public function insert($data)
{
    $data['created_at'] = $data['updated_at'] = date('Y-m-d
```

```
H:i:s');

    $success = $this->db->insert($this->_table, $data);

    if ($success)
    {
        return $this->db->insert_id();
    }
    else
    {
        return FALSE;
    }
}
```

`date('Y-m-d H:i:s')`, as I'm sure you'll know, is the standard `DATETIME` format for MySQL. Adjust accordingly for your database server.

```
public function update()
{
    $args = func_get_args();
    $args[1]['updated_at'] = date('Y-m-d H:i:s');

    if (is_array($args[0]))
    {
        $this->db->where($args);
    }
    else
    {
        $this->db->where('id', $args[0]);
    }
```

```
        return $this->db->update($this->_table, $args[1]);
    }
```

And now, every time you insert something into your database via your model, or update a row, the *created_at* and *updated_at* columns will be updated appropriately.

Observers

There are tons of occasions where you need to alter the data going in and out of your models. These are usually things like assigning the current user's ID to a row, adding timestamps, and hashing passwords. One way of achieving an MVC implementation of this would be to overload the base methods or add custom methods to the model. While this *would work*, it's not particularly nice. It's not consistent, and you'd end up violating the core DRY principle.

A better way of achieving this is to use a technique called **observing**. Observers are callback methods that sit in your model. They are called at certain points (or with certain state changes). You'll probably be familiar with the observer pattern already; it's used across the programming world in a variety of different circumstances.

There are several points (state changes; moments) that we may need to be notified about. They include before and after:

- A row has been created
- A row has been updated
- A row has been retrieved
- A row has been deleted
- Validation

I'm only going to focus on the first. There are many moments in a program's flow where you might wish to notify methods of state changes. However, the pattern is incredibly similar across the board, so if you'd like to add the pattern in anywhere else, it'll be easy.

Just like our validation rules, we're going to define the observers at the top of our model:

```
class User_model extends MY_Model
{
    public $before_create = array( 'hash_password' );
}
```

We can now modify our `MY_Model`'s `insert()` method so that our `hash_password()` method is called before a row is inserted into the database:

```
foreach ($this->before_create as $method)
{
    $data = call_user_func_array(array($this, $method),
array($data));
}

$success = $this->db->insert($this->_table, $data);
```

...and let's add the before_create array to our `MY_Model` so we don't get any errors if the user doesn't define any methods:

```
public $before_create = array();
```

Fantastic. Let's do the same with `after_create()`:

```
public $after_create = array();

// ...

$success = $this->db->insert($this->_table, $data);

if ($success)
{
    foreach ($this->after_create as $method)
    {
        call_user_func_array(array($this, $method),
array($data));
    }

// ...
```

But, discerning readers will have noticed that we are now duplicating code, and this violates our DRY philosophy. Let's abstract the observer mechanism into an `observe()` function.

```
public function observe($event, $data)
{
    if (isset($this->$event) && is_array($this->$event))
    {
        foreach ($this->$event as $method)
        {
            $data = call_user_func_array(array($this,
$method), array($data));
```

```
            }
        }

        return $data;
    }
```

We can then use this function, which gives us a decent observation mechanism, anywhere we want, rather than duplicating code:

```
$data = $this->observe('before_create', $data);

$success = $this->db->insert($this->_table, $data);

if ($success)
{
    $this->observe('after_create', $data);
```

Let's now implement our `hash_password()` function:

```
public function hash_password($user)
{
    $user['password'] = sha1($user['password']);

    return $user;
}
```

While this is a simple mechanism of hashing a user password, it demonstrates how simple it can be to define observers and clean up your

codebase. It's important to remember that every observer callback you define *needs to return the $data variable* that's passed through.

With this abstracted observer pattern in place, we can add any observers we like in our models, even in our custom methods. Observers are yet another way of simplifying and enhancing existing and new code.

Scoping

Model scoping allows you to easily (and beautifully) manipulate your queries, particularly making database finds elegant and convenient. They open up a whole new world of “literate” programming. The essence of model scoping is that by chaining named methods together, you can change the parameters of a query, while retaining readability, keeping this logic in the model, and ensuring you don't repeat yourself. In a similar way that CodeIgniter's ActiveRecord allows you to chain methods together to build queries, you can use model scoping to incrementally add onto your query.

The trick to writing scopes is `return $this`. By returning `$this`, you're returning the current instance of the model class. This allows PHP to chain the methods onto one another. Take, for example, a regular model function:

```
public function get_all_confirmed()
{
    return $this->db->where('confirmed', 1)
                    ->order_by('date')
                    ->get($this->_table)
                    ->result();
}
```

This might not appear too messy, but what happens if we're trying to get many confirmed rows by a country:

```
public function get_all_confirmed_by_country($country)
{
    return $this->db->where('confirmed', 1)
                    ->where('country', $country)
                    ->order_by('date')
                    ->get($this->_table)
                    ->result();
}
```

It's getting a little messier. We also want to get many confirmed rows by first name, so instead of defining another `get_all_confirmed_by_blah()` method, we consolidate these lookups:

```
public function get_all_confirmed_by($key, $value)
{
    return $this->db->where('confirmed', 1)
                    ->where($key, $value)
                    ->order_by('date')
                    ->get($this->_table)
                    ->result();
}
```

...and then what happens if we decide we need to get one? Do we duplicate the method? Or do we add some logic and add a third parameter to return one or many?

You can see how quickly this whole lookup process can get convoluted. Let's rethink this. What if, instead, we had one clean method that added the confirmed query, one method that got all rows, one method that got all rows by and one method that we got a single row by?

```
public function confirmed()
{
    $this->db->where('confirmed', TRUE);
    return $this;
}

public function by($key, $value)
{
    $this->db->where($key, $value);
    return $this;
}

public function get_all()
{
    return $this->db->get($this->_table)
                    ->result();
}

public function get()
{
    return $this->db->get($this->_table)
                    ->row();
}
```

That's looking much, much better already. If we then combine it with our previously constructed `MY_Model` we have an even neater model:

```
public function confirmed()
{
    $this->db->where('confirmed', TRUE);
    return $this;
}
```

How do we interact with our model?

```
$users = $this->user->confirmed()->get_all();
$users_uk = $this->user->confirmed()->by('country', 'United Kingdom')->get_all();
```

This is already an almost flawless system. With just a tiny bit of code, we've managed to considerably clean up our model and have given ourselves a reusable interface to add certain parameters to our query. We can add or remove as many scopes as we like, and they can be as comprehensive or as simple as we like. It doesn't even have to be limited to database lookups. We're sharing an instance of the class, so we can add things to an instance variable which we can then print, or we can perform other data-related processes all scoped out by our method scopes. There's just one problem.

What if we want to make a separate query INSIDE our chain mechanism? Let's say we need to pull in some data from a separate table or database. We might be able to achieve this with a `JOIN`, but it might already be a complex query where throwing another `JOIN` in the fray would only complicate matters. Take, for example, this scope:

```
public function favorited()
{
```

```
    $fav =
$this->db->select('user_id')->get('favorites')->result();
    $ids = array();

    foreach ($fav as $row)
    {
        $ids[] = $row->user_id;
    }

    $this->db->where_in('id', $ids);

    return $this;
}
```

This will only work if it's called first in the chain. You can easily get conflicts in SQL queries when you're chaining and making separate queries inside the scopes. You'll be looking for columns that don't exist, and executing the previously called scopes. The call to `$this->db->get()` resets any other Active Record parameters, so, for example, if we called the scope thus:

```
$this->user->confirmed()
           ->favorited()
           ->get_all();
```

We'd be searching for all rows in the **favorites** table where **confirmed = 1**. The **favorites** table might not have a **confirmed** column, and then, instantly, we've got an erroring query. We don't want to have to worry about putting these things in the correct order. Thankfully, the solution is reasonably simple.

The trick is to *isolate the query*. Previously, I'd take the current values from CodeIgniter's ActiveRecord class, cache them, clear them, make the query, and then reset the variables to their original state. However, these variables start with an underscore. While they're still public–internally they're defined with `var`–I have a funny feeling that sooner or later they'll be defined as `protected` or `private`, which will break this method.

My solution is to re-initialize the database class and return that, allowing methods to run queries that start with a blank slate. All this takes is a call to the core `DB()` function:

```
public function favorited()
{
    $db = DB('default');

    $fav =
$db->select('user_id')->get('favorites')->result();
    $ids = array();

    foreach ($fav as $row)
    {
        $ids[] = $row->user_id;
    }

    $db->close();

    $this->db->where_in('id', $ids);

    return $this;
}
```

Now we can use the `favorited()` scope without fear!

```
$this->user->confirmed()
           ->sorted()
           ->favorited()
           ->get_all();
```

As you can see, scopes are a powerful and flexible way of creating semantic, clean, and concise code.

Validation

One of the biggest mistakes nearly all CodeIgniter developers make is to place the validation in the controller layer. CodeIgniter's built-in library promotes this bad practice, but, as we learned earlier, validation is processing data, so it *needs to be in the model layer*.

That said, CodeIgniter's form validation library is pretty great. It provides a bunch of validation methods that make validating your POST data incredibly easy.

But what if you're not validating form data? What if you're writing an API, and you're validating the input data from the user's request? What if you're bringing in data through some form of import mechanism: perhaps a CSV file, a CMS, or another data-driven system?

These are all cases where you'll be wanting to validate the data, but you cannot do it through CodeIgniter's default validation mechanism.

Continuing the idea of fat model, skinny controller, let's begin by defining the validation rules in our model directly. We only need to do this once, so we can define it at the top of the class:

```
class User_model extends MY_Model
{
    public $validate = array(
        array( 'field' => 'username', 'label' => 'Username',
'rules' => 'required|max_length[20]|alpha_dash' ),
        array( 'field' => 'password', 'label' => 'Password',
'rules' => 'required|min_length[8]' ),
        array( 'field' => 'email', 'label' => 'Email',
'rules' => 'valid_email' )
    );
}
```

As you can see, we are using the same syntax of array as if we were passing this through to the `$this->form_validation->set_rules()` method. We can then define a function to actually process the validation.

Since CodeIgniter's form validation function directly accesses the `$_POST` array, and we want to validate any arbitrary array of data, we have to do something fairly dirty in order to enable this functionality. We have to directly alter the `$_POST` array before passing the rules through to the form validation library, thus fooling the library into thinking that the data has come from the user's `$_POST` data.

We'll create a `validate()` function, which will check to see that we have validation rules, load the form validation library, populate the `$_POST` data, and run the validations.

```
public function validate($data)
{
    if (!empty($this->validate))
    {
        foreach ($data as $key => $value)
        {
            $_POST[$key] = $value;
        }

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

        $this->form_validation->set_rules($this->validate);
        return $this->form_validation->run();
    }
    else
    {
        return TRUE;
    }
}
```

Now, in our controller, instead of setting the rules and processing the validation, we can simply call `validate()` to ensure that the data is valid before we insert it into the database. If it's not valid, we can access the validation errors just like normal, through the `validation_errors()` helper method. The associated controller code could look something like this:

```
public function create()
{
    $user = $this->input->post('user');

    if ($this->user->validate($user))
```

```
    {
        $user_id = $this->user->insert($user);
        redirect('/users/show/' . $user_id);
    }
    else
    {
        $this->session->set_flashdata('error',
validation_errors());
        redirect('/users/add');
    }
}
```

This is clean, and certainly clearer than it could be, but we can still improve on this. Remember the `MY_Model` that we created earlier? We can move the `validate()` method into there, which will provide consistent validation across the board.

We can also modify our `insert()` and `update()` methods to support data validation.

```
public function insert($data, $skip_validation = FALSE)
{
    $data['created_at'] = $data['updated_at'] = date('Y-m-d
H:i:s');

    if (!$skip_validation && !$this->validate($data))
    {
        $success = FALSE;
    }
    else
    {
```

```
            $success = $this->db->insert($this->_table, $data);
        }

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

We're adding a second parameter, the `$skip_validation` variable, which will allow us to skip the validation if we so choose. Let's do something similar with `update()`.

```
    public function update()
    {
        $args = func_get_args();
        $args[1]['updated_at'] = date('Y-m-d H:i:s');

        $validate = (isset($args[2])) ? !$args[2] : TRUE;

        if ($validate && $this->validate($args[1]))
        {
            if (is_array($args[0]))
            {
                $this->db->where($args);
            }
            else
            {
                $this->db->where('id', $args[0]);
            }

            return $this->db->update($this->_table, $args[1]);
        }
        else
```

```
    {
        return FALSE;
    }
}
```

Again, we add an optional third parameter which allows us to skip the validation. Otherwise, we validate the data before we process it.

With these changes implemented, the previous controller code can now become this:

```
public function create()
{
    $user = $this->input->post('user');

    if ($user_id = $this->user->insert($user))
    {
        redirect('/users/show/' . $user_id);
    }
    else
    {
        $this->session->set_flashdata('error',
validation_errors());
        redirect('/users/add');
    }
}
```

That's a little neater and more concise. More important, it goes to show how easily we can ensure the adherence of the MVC pattern within CodeIgniter.

MY_Model

Over this chapter we've built up a pretty smart and efficient `MY_Model`. While it's incredibly important to understand the method behind the magic, it's also important to try to minimise your work and ensure that you make the most out of the resources made available by the community.

Most of the ideas, concepts and code in this chapter have been lifted and adapted from the code in my freely-available, open-source `MY_Model`. It's under active development, has a fair amount of community contribution and is used in lots of popular applications, including the fantastic PyroCMS.

You can head over to the GitHub repository[1] and download my MY_Model for free. Copy it into your **application/core** directory and away you go!

1.https://github.com/jamierumbelow/codeigniter-base-model

views
PART 2

...in which we tidy up our presentation logic and discuss concepts present in the Rails world that makes it easier to write cleaner and better-formatted views. We'll look at separating the presentational logic out into an abstracted class, allowing us to DRY up our views and keep them clean. We'll examine partials, letting us repeat sections of content across the application, and take a look at caching fragments of views for sites with heavier loads.



Presenters

Over the course of an application's development, it can be very common for views to become messy. Views can easily become obfuscated and mixed up, which makes it much easier to violate DRY and MVC. After all, there shouldn't be any business logic in the view.

Presenters are a rather new technique that I discovered in the Rails world that can help by adding another layer of abstraction, by providing a class representation of the state of view. I find that presenters can be a really sleek way of hiding presentational logic.

Let's look at a simple view that displays some information about a user's bank account:

```
<div id="account">
    <h1><?= $this->bank->get($account->bank_id)->name ?> -
<?= $account->title ?></h1>

    <p class="information">
        <strong>Name:</strong> <?php if ($account->name):
?><?= $account->name ?><?php else: ?>N/A<?php endif; ?><br />
        <strong>Number:</strong> <?php if
($account->number): ?><?= $account->number ?><?php else:
?>N/A<?php endif; ?><br />
        <strong>Sort Code:</strong> <?php if
($account->sort_code): ?><?= substr($account->sort_code, 0,
2) . "-" . substr($account->sort_code, 2, 2) . "-" .
substr($account->sort_code, 4, 2) ?><?php else: ?>N/A<?php
endif; ?>
    </p>

    <p class="balances">
        <strong>Total Balance:</strong> <?php if
($account->total_balance): ?><?= "&pound;" .
number_format($account->total_balance) ?><?php else:
?>N/A<?php endif; ?>
        <strong>Available Balance:</strong> <?php if
($account->available_balance): ?><?= "&pound;" .
```

```
number_format($account->available_balance) ?><?php else:
?>N/A<?php endif; ?>
    </p>

    <p class="statements">
        <?php if ($this->statements->count_by('account_id',
$account->id)): ?>
            <?= anchor('/statements/' . $account->id, 'View
Statements') ?>
        <?php else: ?>
            Statements Not Currently Available
        <?php endif; ?>
    </p>
</div>
```

This is a rather typical view; it's displaying bits of content from an object, checking for a value's existence, and pulling in bits from other database tables. It's fine, but it's all a bit messy, and we're duplicating a fair amount of code. Ideally, we want our view to look something like this:

```
<div id="account">
    <h1><?= $account->title() ?></h1>

    <p class="information">
        <strong>Name:</strong> <?= $account->name() ?><br />
        <strong>Number:</strong> <?= $account->number()
?><br />
        <strong>Sort Code:</strong> <?=
$account->sort_code() ?>
    </p>
```

```
    <p class="balances">
        <strong>Total Balance:</strong> <?=
$account->total_balance() ?>
        <strong>Available Balance:</strong> <?=
$account->available_balance() ?>
    </p>

    <p class="statements"><?= $account->statements_link()
?></p>
</div>
```

This clears up our view considerably and removes a bunch of the duplication. It also strips out as much logic as possible from the views, and can make for some very succinct code. This means that we can output this information again in other places, if we so choose. **(DRY, DRY, DRY!)**

We're going to create an **application/presenters** directory, inside of which is the **presenter.php** file. This presenter base file will define a simple class that our other presenters will inherit from.

```
class Presenter
{
    public function __construct($object)
    {
        $name = strtolower(str_replace("_presenter", "",
get_class($this)));

        $this->$name = $object;
    }
}
```

The constructor here is going to automatically guess the name of the object and set it locally as an instance variable (so, effectively, we're *presenting* whatever object we pass through). We also want to be able to access the CodeIgniter superobject, so let's define the PHP magic method `__get()`.

```
public function __get($attr)
{
    if (isset(get_instance()->$attr))
    {
        return get_instance()->$attr;
    }
}
```

`__get()` is called whenever the program tries to access a variable on an object that hasn't been defined. We're passing it through to `get_instance()` so any calls to CodeIgniter in our presenter–for example, to `$this->load` or `$this->db`–will be passed straight through.

Finally, we need to load the Presenter class into our application. This is a simple `require_once`, but where should we put it? Well, you can put it in any file that is loaded globally: **index.php**, **config/config.php** and **config/autoload.php** are three great examples. Personally, I put my global calls to `require_once` in **autoload.php**, because to me it makes the most sense there.

```
require_once APPPATH . 'presenters/presenter.php';
```

Next up, let's create an **account_presenter.php** file. This file will contain the presenter class for our account object. We can then extract the

cluttered logic from our view and place them in the class. Let's start by extracting the title:

```
class Account_Presenter extends Presenter
{
    public function title()
    {
        return
$this->bank->get($this->account->bank_id)->name . "-" .
$account->title;
    }
}
```

We're essentially just porting the logic that was previously in the view into a `title()` method. We can now go ahead and tidy up the information section:

```
public function name()
{
    return $this->account->name ?: "N/A";
}

public function number()
{
    return $this->account->number ?: "N/A";
}

public function sort_code()
{
    if ($sc = $this->account->sort_code)
    {
```

```
        return substr($sc, 0, 2) . "-" . substr($sc, 2, 2) .
"-" . substr($sc, 4, 2);
    }
    else
    {
        return "N/A";
    }
}
```

Similarly, we can tidy up the balances section:

```
public function total_balance()
{
    return ($this->account->total_balance) ? "&pound;" .
number_format($this->account->total_balance) : "N/A";
}

public function available_balance()
{
    return ($this->account->available_balance) ? "&pound;" .
number_format($this->account->available_balance) : "N/A";
}
```

And finally, the statements link:

```
public function statements_link()
{
    if ($this->statements->count_by('account_id',
$this->account->id))
```

```
    {
        return anchor('/statements/' . $this->account->id,
'View Statements');
    }
    else
    {
        return "Statements Not Currently Available";
    }
}
```

Fantastic. Now here's the clever bit. Instead of passing through the account object directly from our model, we first wrap it in our presenter. In this respect, the presenter represents the public-facing output of the database row.

```
public function show($account_id)
{
    $this->data['account'] = new
Account_Presenter($this->account->get($account_id));

    $this->load->view('account/show', $this->data);
}
```

We won't need to load our presenters on every controller, so it makes more sense that we load the presenter at the top of each controller that needs it.

```
require_once APPPATH . 'presenters/account_presenter.php';
```

...and with that we're done!

By adding another layer of abstraction we've managed to convert our previously messy and cluttered views into succinct, short, aesthetically pleasing views. Plus, it's nice and DRY.

Partials

There will often be occasions when you're repeating code lots of times across multiple views. When building a CRUD system, for example, you may be replicating the add and edit form code, or you might be outputting a table or an output of a database row. In the spirit of DRY, we can extract this replicated code and move it into what's called a **partial**.

Building a loose partial system into your application makes removing duplication of code and enforcing DRY even easier. Conventionally, Rails partials begin with an underscore. This seems like a good convention to continue (allowing us to easily distinguish between views linked to controller actions and our reusable partials).

We'll start off by creating a **partial_helper.php** file in our **application/helpers** directory. We could go the whole hog and build a partial rendering library. However, we only need simple rendering functionality, so a helper will suffice.

In our helper we'll define a single, solitary function, called `partial()`. We'll start off by getting `partial()` to simply render out the partial.

```
function partial($name, $data)
{
    return get_instance()->load->view($name, $data, TRUE);
}
```

Nothing useful here yet, but not to worry. Like I mentioned above, a lot of the time we'll be looping through a result set from our models and outputting a row. It'd be nice to have this baked into our partial system so we can automatically output multiple partials based on an array.

We want the ability to enable this selectively, so we'll add a third parameter:

```
function partial($name, $data, $loop = FALSE)
{
    $output = "";

    if ($loop && is_array($data))
    {
        foreach ($data as $row)
        {
            $output .= get_instance()->load->view($name,
array( 'row' => $row ), TRUE);
        }
    }
    else
    {
        $output = get_instance()->load->view($name, $data,
TRUE);
    }

    return $output;
}
```

This means we can loop through a result set from Active Record easily, like this:

```
$result = $this->db->get('projects')->result();

<table>
    <?= partial('projects/_row', $result, TRUE) ?>
</table>
```

And then access each row in our partial with the `$row` variable, like so:

```
<tr>
    <td><?= $row->title ?></td>
    <td><?= $row->duration ?></td>
    <td><?= $row->cost ?></td>
</tr>
```

It'd be nice if this method knew what controller we were in, so it could namespace our partials for us (we'll discuss more about this convention in Part 3). We can do this by parsing the partial name and adding our controller name. We'll also add the underscore, for good measure:

```
$name = get_instance()->router->directory .
get_instance()->router->class . '/_' . $name;
```

Now, what if we want to load a partial from outside our controller's directory? Let's check for the existence of a forward slash in our partial name:

```
if (strpos('/', $name) === FALSE)
{
```

```
    $name = get_instance()->router->directory .
get_instance()->router->class . '/_' . $name;
}
```

...but, since we're trying to be clever, and we want to be as conventional as possible, let's do our best to automatically add in the underscore, even if we specify a directory:

```
else
{
    $parts = explode('/', $name);
    $last = count($parts) - 1;

    $parts[$last] = (strpos('_', $parts[$last]) === 0) ?
$parts[$last] : '_' . $parts[$last];

    $name = implode('/', $parts);
}
```

As you can see, we're also checking that we're not doubling up on underscores, because we want our function to work whatever the input. Now that we have all the magic in there, our partial method will respond correctly to the following test cases:

```
partial('row'); // projects/_row.php
partial('people/card') // people/_card
partial('users/_dropdown') // users/_dropdown
```

Fragment Caching

Caching is like flossing. Everybody says you should do it, but unless you're dentally bedevilled, chances are you forget most mornings. However, when your gums bleed every time you brush, you're going to start noticing and realizing that maybe you should have taken the advice of your pleasant, yet dull appointed doctor of dentistry, Dr. Leventhorpe.

Caching is only *really* necessary once you start seeing performance problems that affect your website. Once you notice these issues, you *really* notice them. It is equally important not to over-optimize, but a bit of select caching on some particularly intensive pages can make all the difference to the speed of your site.

CodeIgniter contains reasonably decent filesystem-based page caching, along with a pretty powerful multi-backend caching library. Both methods work well, but they do have their downsides.

CodeIgniter's page caching caches the entire page, and while this aids performance, it can often be inconvenient. If you're showing login/logout links on your pages, they need to respond to the user's session (rather than just display whatever is in the cache). With CodeIgniter's page caching, this is very tricky to do.

A good solution to this kind of problem is Fragment caching. Fragment caching allows you to cache select parts of pages rather than the entire thing, meaning that processor-intensive elements can be cached without losing the flexibility of a dynamic page.

It's important to remember that fragment caching is only really effective when heavy processing is happening *inside* the fragment. That is to say, contrary to MVC, if you're fragment caching, you'll want to call model methods directly inside the fragment/view, rather than in the controller.

We're going to build a simple fragment caching library that will piggyback on CodeIgniter's built-in caching driver. We'll use the Memcached[2] driver in which store our cache data. Memcached is multi-platform, easy to install and has a great PHP extension that is simple to enable via PECL.

From this point onward, I'm going to assume that you've installed Memcached and enabled the PHP extension. The process is very simple, and well documented online.

With Memcached up and running, create a new configuration file, **application/config/memcached.php**, and configure your memcached server connection details:

```
$config['memcached'] = array(
    'hostname' => 'localhost',
    'name'     => 'memcache',
    'port'     => 11211
);
```

Create a **Fragment.php** file in **application/libraries**. We'll start off by defining our class and grabbing an instance of the CodeIgniter superobject.

```
class Fragment
{
    public function __construct()
    {
        $this->ci =& get_instance();
    }
}
```

2.http://memcached.org

Let's load CodeIgniter's caching driver:

```
$this->load->driver('cache');
```

Due to the inherent driver system, we can then go straight ahead and access the Memcached driver.

Our fragment caching is going to work one of two ways. For people using PHP 5.3 or earlier, it's going to look like this:

```
<?php if ($this->fragment->start_cache()): ?>
    <h1><?=
$this->model->some_processor_or_database_heavy_function()
?></h1>
<?php endif; $this->fragment->end_cache(); ?>
```

We can make this code even prettier in PHP 5.3, thanks to anonymous functions:

```
<?= $this->fragment->cache(function(){ ?>
    <h1><?=
$this->model->some_processor_or_database_heavy_function()
?></h1>
<?php }); ?>
```

Firstly, let's initialize a couple of variables in our class:

```
protected $key = '';
protected $fragment = '';

protected $fresh = FALSE;

protected $expire = 3600;
```

Let's get to work on our `start_cache()` method. Next, we want to work out a reproducible cache key, under which to store the fragment. We can do this by grabbing the current URI and combining it with the current line number. We can grab the URI from CodeIgniter, and the current line number through a call to `debug_backtrace()`. We'll also allow the user to set an expiration time, which will default to 3600 seconds (one hour).

```
public function start_cache($expire = 3600)
{
    $backtrace = debug_backtrace();
    $this->key = sha1($this->ci->uri->uri_string .
$backtrace[0]['line']);

    $this->expire = $expire;
```

Now that we've figured out what key we're using, let's try to get it from the cache:

```
if ($fragment =
$this->ci->cache->memcached->get('fragments.' . $this->key))
{
    $this->fragment = $fragment;
```

```
        return FALSE;
    }
```

If it's in the cache, we can set it temporarily as a variable and return `FALSE`. By returning `FALSE`, we prevent the template code in between our `if` statement from being run. We'll output in `end_cache()` momentarily.

If it's not in the cache, we want to mark it as fresh, begin output buffering, and allow the template code to run. We can then cache the data afterward.

```
    else
    {
        $this->fresh = TRUE;
        ob_start();

        return TRUE;
    }
```

Now we can define our `end_cache()` method, which will capture and cache the output if it's fresh, or else output the cached data.

```
    public function end_cache()
    {
        if ($this->fresh)
        {
            $output = ob_get_contents();
            ob_end_clean();
```

```
            $this->ci->memcached->save($this->key, $output,
$this->expire);
        }
        else
        {
            ob_end_clean();
            $output = $this->fragment;
        }

        $this->fresh = FALSE;
        $this->key = '';
        $this->fragment = '';

        echo $output;
    }
```

That was simple! We check to see if the cache is fresh. If it is, we capture the output from the fragment and save it to the cache. Otherwise, we grab the cache. We then reset our variables and output the fragment.

Using PHP 5.3, we can make this even sweeter, by wrapping the fragment in a single call to a `cache()` function:

```
public function cache($fragment, $expire = 3600)
{
    if ($this->start_cache($expire))
    {
        $fragment();
    }
```

```
        $this->end_cache();
    }
```

And with under 50 lines of code, we've got a fully functional, smart and easy to use fragment caching library. Fragment caching is most effective with process- or database-heavy actions happening *inside* the fragment; it can be a really smart way of caching particular fragments of your views.

controllers
PART 3

...in which we examine the controller layer and thoroughly improve the way that the controller links up the views and models in our application. We discuss enforcing conventions to autoload our views and models, we build a configurable yet conventional layout system with which to load our views, and we discuss using filters to run code pre- and post-action.



Autoloading Views

Let me bring up a perfectly understandable, non-shocking and non-disruptive convention for organizing views.

Views should be housed in a directory named after the controller, and should themselves be named after the controller action.

This makes nothing but perfect sense. If we're at a URL, say, `/users/list`, it makes absolute sense that the view to be loaded would be called **views/users/list.php**. In fact, you may have a naming pattern like this in place already.

With this convention in mind, we can dramatically reduce the need for calls to `$this->load->view()`, by automatically loading our views after our controller method is run. We can DRY up our controllers immensely and create really nice, succinct, clean controller code.

In order to achieve this, we're going to utilize an often overlooked feature of CodeIgniter: `_remap()`. `_remap()` is a method that, if it exists in a controller, will be called by CodeIgniter *instead* of the controller action. This allows developers to hook straight into the call process and execute functionality before and after each action.

Let's get cracking straight away by creating a **MY_Controller.php** file inside **application/core**. We'll use the `MY_Controller` to allow every controller in our application to have magically autoloading views.

Begin with the usual stuff:

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

We'll define our `_remap()` function. It will take two parameters, `$method`, the action name, and `$parameters`, the parameters from the routed URL segments intended to be passed into the controller action.

```
public function _remap($method, $parameters)
{
    call_user_func_array(array($this, $method), $parameters);
}
```

We're using `call_user_func_array()` to call the method, passing through the parameters. Let's first make a sanity check to ensure that the method exists. If it doesn't, we want to respond with a 404.

```
if (method_exists($this, $method))
{
    call_user_func_array(array($this, $method), $parameters);
}
else
{
    show_404();
}
```

If we're successfully calling the method, we want to build up the view name from the controller and the action:

```
$view = strtolower(get_class($this)) . '/' . $method;
```

And how do we pass data through? Instead of defining a local `$data` variable, we can define an instance-level `$this->data` variable. This comes in handy not just here, but it also allows us to pass through data to each view at a global level.

```
public $data = array();
```

Now that it's set up, we can load the view:

```
$this->load->view($view, $this->data);
```

This is going fantastically. We already have a great convention set up. But what if we feel like being unconventional? What if we decide not to load a view after all, opting instead to spit out some JSON, or perhaps some raw text from the controller?

Let's add an instance variable called `$this->view`. We'll check if `$this->view` is set to `FALSE`. If it is, we won't show a view. If it isn't, we'll go right ahead and output the view.

```
public $view = TRUE;
```

...and our conditional:

```
if ($this->view !== FALSE)
{
```

```
    $this->load->view($view, $this->data);
}
```

So far so good! Finally, to make this even more configurable, there may be moments when we want to specify the view to load precisely. We may want to load a shared view, or a view out of our conventional naming pattern. Let's add to our conditional. We'll check if `$this->view` is a string–if it is, we know we want to load it.

```
$view = (is_string($this->view) && !empty($this->view)) ?
$this->view : $view;

if ($this->view !== FALSE)
{
    $this->load->view($view, $this->data);
}
```

And there we have it! It's a simple technique, but using this convention and autoloading your views can instantly clean up your controller actions. Take, for example, the following method:

```
public function index()
{
    $data['projects'] = $this->project->get_all();
    $data['title'] = 'All Projects';

    $this->load->view('projects/index', $data);
}
```

Using our autoloading code, we can quickly simplify and strip off 50% of the method body:

```
public function index()
{
    $this->data['projects'] = $this->project->get_all();
    $this->data['title'] = 'All Projects';
}
```

It's important to remember that, although this difference might not seem big, each little bit adds up, and very quickly the controller becomes a sprawling mess. A good rule of thumb to follow is:

If the code doesn't need to be there, it probably shouldn't be.

Layouts

One of the biggest issues with the previously proposed autoloading solution is loading the view into a layout. While there are a few good layout libraries available for CodeIgniter, they're not commonly used, and personally I prefer to control the loading of layouts myself.

Take this commonly seen block of code:

```
$this->load->view('shared/_header', $data);
    $this->load->view('users/all', $data);
$this->load->view('shared/_footer', $data);
```

It *seems* to be fine, but when you closely examine the way it works, it starts to show its problems. Ignoring the fact that you'll be duplicating a lot of

code, it also doesn't give you a huge amount of flexibility. A layout system like this presupposes a flat, un-dynamic layout. The moment changing sidebars and flexible mastheads come into the equation, everything starts to get even more complicated.

A better solution would be to have a single layout for the application and load the view into this as a variable. It's then at the layout's discretion where it decides to output the view. It could look like this:

```
$view = $this->load->view('users/all', $data, TRUE);
$this->load->view('layouts/application', array( 'view' =>
$view ));
```

This is a good start, and is how Rails tackles the problem. For consistency's sake, and because it's a more descriptive term, I'm going to rename the `$view` variable to `$yield`. We'll also move the layout loading into our previously created `MY_Controller`:

```
if ($this->view !== FALSE)
{
    $this->data['yield'] = $this->load->view($view,
$this->data, TRUE);
}

$this->load->view('layouts/application', $this->data);
```

We'll pass through our `$this->data` to the layout in case we want to set a title, breadcrumbs, or something else that needs to sit outside of our view.

So far, this is a really great solution. Views are automatically being loading into a layout, in which we can then specify exactly where we want to spit out the view:

```
<header>
    <h1>My Application</h1>
</header>

<div id="wrapper">
    <?= $yield ?>
</div>

<footer>
    <p>Copyright 2012</p>
</footer>
```

The first problem I can spot here is if we want to load our view into a separate layout rather than our global application layout. There are several occasions when you'll need to do this. Login forms, micro-sites and admin panels all could have distinct looks from the rest of the application.

We can solve this problem by establishing the convention that a *controller* could have its own layout. This way, if the controller has a layout, we load that, otherwise we go for the generic application layout.

Like before, we can implement this easily:

```
if (file_exists(APPPATH . 'views/layouts/' .
strtolower(get_class($this)) . '.php'))
{
```

```
    $layout = 'layouts/' . strtolower(get_class($this));
}
else
{
    $layout = 'layouts/application';
}

$this->load->view($layout, $this->data);
```

Much like our views, it'd be nice if we could choose to not display a layout at all. We might want to output a snippet of HTML in an AJAX call, for instance, and for that we probably don't want the layout. Let's add an instance variable and copy what we did with views:

```
public $layout = TRUE;
```

And, like before, check if it exists:

```
if ($this->layout)
{
    $this->load->view($layout, $this->data);
}
else
{
    echo $this->data['yield'];
}
```

It would also be useful if we could specify the layout. We can do something incredibly similar like we did with views:

```
if (is_string($this->layout) && !empty($this->layout))
{
    $layout = $this->layout;
}
elseif (file_exists(APPPATH . 'views/layouts/' .
strtolower(get_class($this)) . '.php'))
{
    $layout = 'layouts/' . strtolower(get_class($this));
}
else
{
    $layout = 'layouts/application';
}
```

With all of these changes, our `_remap()` function should look like this:

```
public function _remap($method, $parameters)
{
    if (method_exists($this, $method))
    {
        call_user_func_array(array($this, $method),
$parameters);
    }
    else
    {
        show_404();
    }

    $view = (is_string($this->view) && !empty($this->view))
? $this->view : $view;
```

```
    if ($this->view !== FALSE)
    {
        $this->data['yield'] = $this->load->view($view,
$this->data, TRUE);

        if (is_string($this->layout) &&
!empty($this->layout))
        {
            $layout = $this->layout;
        }
        elseif (file_exists(APPPATH . 'views/layouts/' .
strtolower(get_class($this)) . '.php'))
        {
            $layout = 'layouts/' .
strtolower(get_class($this));
        }
        else
        {
            $layout = 'layouts/application';
        }

        if ($this->layout)
        {
            $this->load->view($layout, $this->data);
        }
        else
        {
            echo $this->data['yield'];
        }
    }
}
```

And that is really all it takes to have a totally configurable and customizable, convention-powered automagical view loader.

Autoloading Models

Carrying on with the autoloading theme, a simple technique you can use to clear up your controller method is to provide a small model-autoloading interface. Usually, if you load a model in your controller, chances are you want to use it across multiple actions.

Loading a bunch of models in your constructor can look pretty messy. It'd be much nicer if we could take the conventions we established in Part 1 and expand on them to allow us to automatically load models based on those conventions.

I've always named my models along these lines: **singular_resource_model.php**. For a *users* table, it'd be **user_model.php**. For a model that handles files, it'd be **file_model.php**. Additionally, I access my models like this:

```
$this->user->get_all();
$this->file->upload();
```

Finally, I *usually* work to the assumption that my controller name will be a plural word, mapping to a singular model. We can use this assumption to try to automatically load the model related to our controller.

Open up our **MY_Controller.php** like before. This time, we'll work inside the constructor. We'll fetch the model name by singularizing the controller name:

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

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

    $model = strotolower(singular(get_class($this)));
```

We'll check that the model exists, and if it does, load it:

```
if (file_exists(APPPATH . 'models/' . $model . '_model.php'))
{
    $this->load->model($model . '_model', $model);
}
```

This is great so far, but it doesn't really solve our problem. Chances are we'll need to load other models too. How can we simplify this process?

My solution is to add a `$this->models` array to our controller which will automatically add the *_model* and load each model inside it. That way the list of models is confined to a single place, with as little repetition as possible:

```
public $models = array();
```

Inside our controller we can loop through this array:

```
foreach ($this->models as $model)
{
    $this->load->model($model . '_model', $model);
}
```

We can even refactor the previous autoloading code to use this array:

```
if (file_exists(APPPATH . 'models/' . $model . '_model.php'))
{
    $this->models[] = $model;
}
```

Then, in our controller itself, we can specify any of the models we want to load within this array:

```
public $models = array( 'user', 'project', 'benchmark' );
```

Comparing this to the code previously required in the constructor, you see:

```
$this->load->model('user_model', 'user');
$this->load->model('project_model', 'project');
$this->load->model('benchmark_model', 'benchmark');
```

Suddenly, the difference becomes obvious. Again, it may only be a subtle difference, but when working on larger apps with higher numbers of files and larger blocks of code, the small changes can make all the difference.

The more concise and succinct your code is, (usually) the more maintainable.

Filters

There will be occasions when you need to execute code on a controller-by-controller basis, just before or after the controller action has been run. Much like the model observers we saw in Part 1, or the autoloading of views we've been using here in Part 3, we may need to execute snippets of code before and after actions.

This comes in especially useful for things like authentication. You can easily get a method to be called pre-action across your controller methods. You're probably thinking “just put it in the constructor,” but what happens if you want to call the method on all actions but one? Suddenly, the concept of filters plays a magnificent role.

In order to maintain our already sprawling `_remap()` function, we're going to place our call to `call_user_func_array()` in between two new methods which we'll create, `_run_before_filters()` and `_run_after_filters()`.

```
$this->_run_before_filters($method, $parameters);
    call_user_func_array(array($this, $method), $parameters);
$this->_run_after_filters($method, $parameters);
```

Our methods are going to take our action and parameters, and pass them through to the filters we define at the top of the class. We want to be able to specify the filters in one of two ways. First, as a flat array:

```
public $before_filters = array( 'authenticate_user',
'fetch_account' );
```

Second, as an associative array, with the ability to specify which actions should be included or excluded:

```
public $before_filters = array( 'authenticate_user' =>
array( 'only' => 'secure' ),
                                'fetch_account' => array(
'except' => 'select_account' ) );
```

Finally, we want to be able to mix and match the two:

```
public $before_filters = array( 'authenticate_user',
                                'fetch_account' => array(
'except' => 'select_account' ) );
```

With this in mind, we'll set up our base arrays (so that we don't have to do superfluous checking), and then define our `_run_before_filters()` method.

```
public $before_filters = array();
public $after_filters = array();

protected function _run_before_filters($action, $parameters)
{
    foreach ($this->before_filters as $filter)
    {
```

```
            $this->$filter($action, $parameters);
        }
    }
```

This fulfills requirement number one. We can define a flat list of methods that will be called and pass the action and parameters. Requirement number two and three are a little more tricky. We have to check to see if the key (`$details`) is a string. If it is, we know it's a flat call. If it isn't, we know that `$filter` is the method name and that we need to check `$details` for the specific array.

If `$details` is an array, we need to check to see if the action is either in the *only* list or not in the *except* list. Finally, if we can't figure out what `$details` is, we'll ignore this filter.

```
foreach ($this->before_filters as $filter => $details)
{
    if (is_string($details))
    {
        $this->$details($action, $parameters);
    }
    elseif (is_array($details))
    {
        if (in_array($action, @$details['only']) ||
!in_array($action, @$details['except']))
        {
            $this->$filter($action, $parameters);
        }
    }
}
```

Let's extract this into a separate method, `_run_filters()`. That way, we don't have to duplicate any code in order to get this to work for “after” filters too.

```
protected _run_filters($what, $action, $parameters)
{
    $what = $what . '_filters';

    foreach ($this->$what as $filter => $details) {
```

Then we'll make the change to our `_remap()` method:

```
$this->_run_filters('before', $method, $parameters);
    call_user_func_array(array($this, $method), $parameters);
$this->_run_filters('after', $method, $parameters);
```

And there we have it! A fully functional, configurable way of DRYing up your controller code and providing pre- and post- action methods.

rest

PART 4

...in which we look at Representational State Transfer (REST) and learn why it's a brilliant way of designing your routes, and how to best implement it in CodeIgniter. We take a good, in-depth look at the core of HTTP and the concept of resources on which its foundations lie. Then we develop a series of replicable routing patterns that make it easy to have RESTful routing within CodeIgniter applications.



HTTP: The Forgotten Protocol

The vast, vast majority of developers are only aware (or only care about) two fundamental HTTP methods; `GET` and `POST`. Considering the fact that most modern web browsers only support these two, it isn't surprising, but there are a whole range of other request methods that all modern applications should be implementing.

HTTP itself is built upon a fundamental concept: that of a *resource*. The acronyms URL and URI themselves stand for Uniform Resource Locator and Uniform Resource Identifier respectively, and the HTTP spec is designed with the intention of interaction with a resource.

So, to put this all in layman's terms, what are we defining as a resource? *A resource is simply a thing*. A user, a book, a post, a comment, a project, a task, a log or an invoice. And when we begin to think about building our application around the concept of resources, we can really start to see the benefits.

Examining the different HTTP methods available and how they work with resources can be a big help in understanding the core concepts of REST. Not only do we have our faithful friends `GET` and `POST`, we're also joined by `PUT`, `DELETE`, `HEAD`, `OPTIONS`, `TRACE` and `CONNECT`. The latter four aren't helpful for us right now, so let's ignore them. I'm more interested in `PUT` and `DELETE`.

PUT

The HTTP spec describes `PUT` as a method for “storing the enclosed entity under the supplied Request-URI.” It then goes on to state that “If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.”

We can infer from this that `PUT` is a method for updating existing resources on the server (or, less frequently, creating a resource at a pre-determined URI). In terms of REST, this means that the `PUT` method should be used for updating our resource: changing the user's password, renaming a book title, or updating a counter.

According to the REST spec, the `PUT` and `DELETE` methods are *idempotent*, meaning that they are totally replicable. Multiple identical requests should have the same effect as a single request.

DELETE

This one is rather self explanatory. From the HTTP spec, we get “The DELETE method requests that the origin server delete the resource identified by the Request-URI.” So, the `DELETE` method is for deleting your resources. Simple!

What about GET and POST?

We'd forgotten about our trusty friends, `GET` and `POST`. Well, the HTTP spec says: “The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.”

The name itself, `GET`, tells us all that we need to know here. Retrieving content, retrieving resources. This could be a list of resources or an individual one. The spec also allows us to process the input in a way that enables us to retrieve a more relevant list of these resources. This refers to searching, limits and filters.

As for `POST`, the HTTP spec says, “The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.”

This is where things start to get blurry. In terms of the HTTP spec, `POST` is a general-purpose function for submitting a resource to the server. Where REST is concerned, `POST` is used to create new resources.

What does that mean?

(That boy needs therapy?) Using our knowledge of these HTTP methods, we can start to build a pattern of consistent, conventional, resource-driven routes. This routing pattern will give us a uniform set of rules that we can build our application around.

The Standard Routes

The standard set of RESTful routes are broken into two categories: collection URIs, and member URIs. Collection URIs are URIs that apply to the entire collection of resources. Member URIs are URIs that apply to a specific element in that collection. Submitting a certain HTTP request to either one of these URIs produces different effects.

Let's assume our collection is called *posts*.

Collection Routes

HTTP Method + URI	***Result***
GET /posts	List the collection
POST /posts	Create a new member in the collection
PUT /posts	Replace the entire collection
DELETE /posts	Delete the entire collection

Member Routes

HTTP Method + URI	***Result***
GET /posts/:id	Get details about a specific member
POST /posts/:id	Create a new collection based on that member
PUT /posts/:id	Update the member

DELETE /posts/:id	Delete the member

The Important Ones

I've listed the entire set of methods above, but in reality, there are only a few that we need to focus on. There will be very few times when you'll need to replace or delete the entire collection, and even fewer where you'll need to create a new collection based on a member. With that in mind, we only need to implement a select few methods:

- `GET /posts`
- `POST /posts`
- `GET /posts/:id`
- `PUT /posts/:id`
- `DELETE /posts/:id`

CodeIgniter-friendly Semi-RESTful Routing

While it's important for the adherence of REST that we obey the HTTP methods, understanding the process and then simplifying it is certainly a step in the right direction. As most browsers don't support PUT or DELETE, those methods can only really be used on APIs (or in the browser with a bit of work).

While there are ways to use PUT and DELETE, it makes much more sense to adapt the pattern and implement it in a CodeIgniter-friendly way. With a few tricks, we can easily utilize GET and POST effectively and adhere to a RESTful pattern appropriately.

Adding semi-RESTful routes is simple. Open up **config/routes.php**. We'll add our `GET /posts` and `POST /posts`:

```
$route['posts'] = ($_SERVER['REQUEST_METHOD'] == 'POST') ?
'posts/post_create' : 'posts/get_index';
```

Since our routes file is just PHP, we can check to see what the request method is, and thus limit our routing to a certain method. I'm prefixing the controller methods with the request method so that we can use reserved PHP keywords like `new`.

We can replicate that code for our pattern:

```
$route['posts/new'] = 'posts/get_new';
$route['posts/(:any)'] = ($_SERVER['REQUEST_METHOD'] ==
'POST') ? 'posts/post_update/$1' : 'posts/get_show/$1';
$route['posts/(:any)/edit'] = 'posts/get_edit/$1';

if ($_SERVER['REQUEST_METHOD'] == 'POST')
{
    $route['posts/(:any)/confirm_delete'] = 'posts/
post_confirm_delete/$1';
    $route['posts/(:any)/delete'] = 'posts/post_delete/$1';
}
```

This will give us the following URL-to-controller action mapping:

```
GET /products                   -> get_index
GET /products/new               -> get_new
GET /products/1                 -> get_show(1)
GET /products/1/edit            -> get_edit(1)
POST /products                  -> post_create
```

```
POST /products/1                 -> post_update(1)
POST /products/1/confirm_delete -> post_confirm_delete(1)
POST /products/1/delete          -> post_delete(1)
```

This doesn't obey the traditional CodeIgniter expectation of `controller/method/variable`, and it will require you to define routes for each resource you use. This change notwithstanding, it enables us to define a series of nearly RESTful routes and will clean up your routing considerably.

Some modern browsers (at the time of writing IE10 and the most recent builds of Webkit and Firefox) are starting to support `PUT` and `DELETE` as request methods in forms and AJAX2 as the popularity of RESTful interfaces has risen. If you'd like to use truly RESTful routes you'll need to be able to parse the data in PHP. You *could* do this manually with PHP's `php://input`, which would work, or, alternatively, you could use Phil Sturgeon's fantastic REST library[3]. Phil's library works like our `MY_Controller`: you inherit your controllers from `REST_Controller` and access parameters through `$this->get()`, `$this->post()`, `$this->put()` and `$this->delete()` methods.

Phil's library is simple to use, well tested, and very popular. The documentation is somewhat lacking, but in order to counter this Phil has written a rather comprehensive tutorial on the tutorial site NetTuts[4] that should help you get started.

The other benefit to using Phil's library is that it supports a bundle of features for working on APIs, including API keys, request logging, HTTP authentication and multiple output formats. It's a rather sophisticated bit of code, and I've been using it since its early incarnations to build completely

---

3. https://github.com/philsturgeon/codeigniter-restserver
4. http://net.tutsplus.com/tutorials/php/working-with-restful-services-in-codeigniter-2/

RESTful APIs with CodeIgniter. We'll discuss Phil's library and the whats and whys of API design in Volume Two of The CodeIgniter Handbook.

Summary

In this book we've delved into the depths of CodeIgniter in order to improve our process and make coding robust web applications that much more fun. We've looked at the concept of Convention over Configuration, applying it to the full MVC stack. We've set up a base model class with which we can reduce the code across our models, and we've built a flexible templating system to solidify our controllers.

We've looked at cleaning up our views with presenters and partials and we've taken a thorough look at REST, with which we can have a set of common routes for our resource-based applications. All in all, we've managed to dramatically reduce the amount of duplication across our codebase by following the mantra of Don't Repeat Yourself (DRY), and set up a series of conventions that you can now apply to any project, on any framework, in any language.

I hope you've enjoyed reading Volume One in The CodeIgniter Handbook. Part 4 was intended to whet your appetite for REST–Volume Two will focus on creating well-designed and exceedingly pleasant APIs using CodeIgniter. We'll look at the differences between RPC and REST, we'll design our entire data model over the concept of resources and we'll look at some of the lesser-known features of the HTTP protocol.

If you spot any code errors, we'd sincerely appreciate it if you could submit it to our errata at http://codeigniterhandbook.com/errata/. If you have any questions or suggestions how to improve this book, please don't hesitate to contact me at jamie@jamierumbelow.net or on Twitter, at @jamierumbelow.

It'd also be great to hear about the code you've developed with the techniques you've learned in this book. I'm looking forward to seeing what you all come up with!