1

Welcome to the MVC World

This book is about CodeIgniter and the world of Model-View-Controller (MVC) web development. Before venturing into the topic of CodeIgniter, it's helpful to put it into some kind of context. However, most programmers who are new to the world of MVC web development find some of the concepts hard to grasp at first — they've been so engrained in the old way of doing things that unlearning is difficult at first.

So even before you can learn anything about CodeIgniter or even MVC, it's probably helpful to start the discussion with something you already know and understand, and then move on from there. Therefore, that's where this book begins: with a description of a prototypical PHP project, most likely very similar to anyone's very first PHP project.

Many PHP programmers learn PHP either as their first language (having only been exposed to XHTML and CSS beforehand) or after learning another similar language (such as Perl). Most of the time, a PHP programmer's first few projects involve a pretty steep learning curve, but eventually the programmer finds a comfortable groove. He ends up with a project consisting of the following components:

- A series of PHP pages with intermingled PHP commands, SQL queries, and XHTML
- A series of JavaScript and CSS files that are linked from those PHP pages
- A series of universal includes — a footer file for all pages and a header file that contain the database connection and session initialization code
- A handful of database tables that store the application's data

In fact, a newbie programmer's first PHP page probably looks a lot like this one:

```
<?php
include_once "db.php";

$sql = "select * from pages where status='live' and type='home' limit 1";
$result = mysql_query($sql);

while($data = mysql_fetch_object($result)){
  $title = $data->title;
  $css = $data->css;
  $bodycopy = $data->bodycopy;
  $kw = $data->keywords;
  $desc = $data->description;
}
?>
<html>
<head>
<title><?php echo $title; ?></title>
<link href="<?php echo $css; ?>" rel="stylesheet" type="text/css"/>
<meta name="keywords" value="<?php echo $kw;?>"/>
<meta name="description" value="<?php echo $desc?>"/>
</head>

<body>

<?php include_once "topnav.php";?>

<div id="wrapper">
<h1><?php echo $title;?></h1>
<p><?php echo nl2br($bodycopy);?></p>
</div>
<?php include_once "footer.php";?>
</body>
</html>
```

So far, so good, right? The pages render quickly; there's CSS for layout instead of tables, so it should be easy to re-skin; and life is good. The newbie programmer figures that keeping the project small enough means averting disaster. In fact, this first project does stay under a dozen PHP files in size and has one or two JavaScript and CSS files and three or four includes. Everything goes well during development, the project goes live, and the customer is happy.

Of course, the beginner programmer is usually blissfully unaware of some of the *gotchas* of this approach. For example, since the SQL code to retrieve the home page doesn't have any exception handling in it, what happens if the database is unavailable (or someone accidentally erases the home page content from the DB)? The code is tied to mySQL at the moment. What if the customer changes database servers at some point?

If these thoughts occur, they're usually put off. For now, everything works, and it's time to celebrate and move on to the next project. "Besides," the programmer thinks to himself, "you're using includes for the important stuff, like database connections, global navigation, and footer content. Doesn't that make the code more modular?"

Six months later, the programmer gets a call from the client, and the problems begin — not big problems at first, but they have a tendency to snowball. The client wants to add a few database fields, change the way the interface looks, and make the application more flexible in certain ways. Out loud, the programmer tells the client that everything is OK, but inside, a sense of low-level panic is brewing.

Why? Because the programmer barely remembers any details about the project in question six months later. Honestly, the programmer has worked on 10 other applications since that project ended, amounting to many thousands of lines of code written, rewritten, debugged, refactored, and rewritten again to meet the needs of changing requirements. This is not to mention that the programmer isn't exactly indifferent to learning about programming, so he's been reading books on the subject and applying what he's learned to current projects. The programmer knows that the approach he took with this particular project is off the mark. The approach wasn't *wrong*, in a strictly objective way (there are many ways to skin a cat, after all), but it sacrificed long-term support and flexibility in favor of ad hoc construction and initial speed.

Without a doubt, cleaning up the previous work will mean digging through the code to untangle PHP snippets, SQL queries, and XHTML markup; retesting everything to make sure it's working right; and making sure that the application code and interface match the database. It's also likely that the client will change her mind about many things along the way, requiring many course corrections (read: "rewriting code").

Even if the request from the client is relatively simple, involving a new skin or HTML layout, the programmer could find himself carefully picking through code to make sure no vital functionality is erased. The requests get more complex from there, and all of them are filed under "Let's hope they don't ask for that": supporting mobile devices, displaying content in foreign languages, rendering different views of the same data (such as pie chart versus spreadsheet), adding extensive Ajax controls, and so on.

So the programmer takes on the project, knowing full well that the relationship with the client is the most important thing — that, and a sense of pride that won't let him hand the project over to a completely new programmer, who will probably screw it up.

The first thing he decides to take on is that crazy SQL query right on the home page. The thing to do here is to create a functions.php file in which he stores all the functions. The function for that home page content (notice the minimal exception handling) ends up looking like this:

```
<?php

function fetchHomePage(){
  $sql = "select * from pages where status='live' and type='home' limit 1";
  $result = mysql_query($sql);

  while($data = mysql_fetch_object($result)){
    $hp['title'] = $data->title;
    $hp['css'] = $data->css;
    $hp['bodycopy'] = $data->bodycopy;
    $hp['kw'] = $data->keywords;
    $hp['desc'] = $data->description;
  }

  if (count($hp)){
    return $hp;
```

```
    }else{
      $hp['title'] = "Welcome to our web site!";
      $hp['css'] = "default.css";
      $hp['bodycopy'] = "This is our web site!";
      $hp['kw'] = "welcome";
      $hp['desc'] = "our cool site";
      return $hp;
    }
  }
  ?>
```

Now that the data-fetching code is in a separate file, the home page is a bit simplified. You could even say that things are a lot better now:

```
<?php
include_once "db.php";
include_once "functions.php";
$home = fetchHomePage();
?>
<html>
<head>
<title><?php echo $home['title']; ?></title>
<link href="<?php echo $home['css']; ?>" rel="stylesheet" type="text/css"/>
<meta name="keywords" value="<?php echo $home['kw'];?>"/>
<meta name="description" value="<?php echo $home['desc']?>"/>
</head>
<body>
<?php include_once "topnav.php";?>

<div id="wrapper">
<h1><?php echo $home['title'];?></h1>
<p><?php echo nl2br($home['bodycopy']);?></p>
</div>
<?php include_once "footer.php";?>
</body>
</html>
```

Soon, the programmer goes through all the PHP files and converts raw SQL into functions, depositing them into the same functions.php file. In no time at all, there exists a library of functions that extract pages, render calendar views, create RSS feeds, and do heaven knows what else. The programmer ponders this strange little file that's become so bloated and decides to tackle it later — there are more important things on the agenda.

So he gets to work on the rest of the client's requests. They want to add several fields to the `pages` database table, and they want to check for both browser language and client type (Mozilla, Safari, IE) to push custom content out to different users. They also want to incorporate analytics tracking packages from third-party vendors. Also — and they know it's last minute — but one of the founders of the company just got back from a conference and is really hot on using Ajax.

Through it all, the programmer's probably thinking, "There's got to be a better way!" Well, there is, and it involves learning about a way of thinking called *Model-View-Controller (MVC)*. With MVC, developers can separate their code into distinct parts, making the entire application easier to develop, maintain, and

extend. Furthermore, MVC frameworks are usually pretty structured, allowing the developer to concentrate on what's important to the client and the project at hand and not worry about other issues that affect every project (such as security and caching).

The point here is not to chastise the programmer or call out deficiencies in his approach. Everyone has been there and done that. The point is to show you a better way, inspired by a belief in the transformative power of MVC, Agile methodologies, and CodeIgniter. Believe it: *All these things working together can radically change the way you work and interact with clients.*

You'll get to CodeIgniter and Agile methodologies very soon, but for now it's time to focus on MVC concepts. Once you have a strong foundation in the basics, the rest should progress naturally. As you'll soon see, MVC has been around for the past 30 years but has become increasingly popular of late, especially in the world of web development.

What's Model-View-Controller?

Model-View-Controller, as the name implies, is a design pattern that allows developers to cleanly separate their code into three categories:

- **Models** maintain data.
- **Views** display data and user interface elements.
- **Controllers** handle user events that affect models and views.

Figure 1-1 illustrates the relationship among the parts. The important thing to remember is that MVC takes the user into account — it begins with him or her. It's the *user* who clicks a link, moves the mouse, or submits a form. It's the *controller* that monitors this activity and takes the appropriate action (such as manipulating the *model* and updating the *view*).

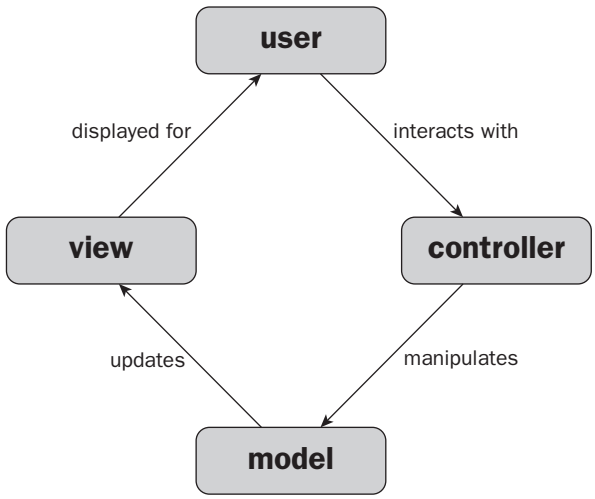

Figure 1-1

Because of MVC's three-part separation, developers can create multiple views and controllers for any given model without forcing changes in the model design. This separation allows for easily maintained, portable, and organized applications that are nothing like anything you've worked with before.

For example, imagine that most prototypical of early 21st century web applications, the blog. Blogs are everywhere these days, and they're not that hard to break into their constituent MVC parts:

- A model that keeps track of posts and comments
- Multiple views that display individual blog posts, a list of blog posts, or search results
- A controller that captures user interaction (such as clicking on an archive link) and then redirects requests to models and/or views

Drilling down further, an MVC blog application might involve the following flow of events:

1. The user visits the blog home page.
2. This simple event requires a controller action for the home page, which makes a quick call to the model to retrieve the last 10 blog posts in reverse chronological order.
3. The model's data are then transmitted to the view for the home page.
4. The view (including the data retrieved by the model) is what the user sees in his or her browser.
5. The user clicks on a link to see details on a particular blog post.
6. The underlying controller captures this user input (clicking the link), uses the model to retrieve the blog post in question from the database, and then loads the appropriate view.
7. The cycle begins anew when the user clicks a link to view comments or runs a search.

Another way of thinking about this breakdown of roles is to map it out this way:

- User Input and Traffic Control = Controller
- Processing = Model
- Output = View

This three-part analogy is a much simpler way of looking at things. Some might argue that the approach is much too simple. After all, it's possible, some might argue, to create an MVC application without a model, or to gang-press view code into the controller. It's not a good idea, but it can be done. For right now, this simple analogy holds. Different ways to approach problems in an MVC fashion are explored later in the book.

Why Bother with MVC?

Although at first glance the MVC approach seems like a lot of work, it really isn't. When developers of the hypothetical MVC blog application want to change something about the home page, they ask themselves about the nature of the change. Doing so allows them to zero in on the part of the application they need to work on.

- If they want to change the number of blog posts that get retrieved (or even the order in which they are displayed), they update the model.
- If they want to change the way the home page looks, they update the view.
- If they want to add a new page to their application, they first add a method to their controller and then build out any supporting views (to display content in the browser) and models (to gather data from a database).

The beauty of the entire system lies in the fact that none of those hypothetical actions on any given part of the MVC triumvirate affects the others. It's true that changing the model in some way (retrieving 15 blog posts instead of 10) will change the view, but the developers didn't have to dig through their view (or controller) to make that change, then later realize they had another SQL call embedded in other files that did the same thing. Theoretically, on large projects, you could have developers who focus only on views, controllers, or models, thus keeping clean lines of sight on deliverables and responsibilities. (Stop laughing. It is possible, you know!)

If you work as an in-house developer, you'll quickly learn to appreciate how MVC can help teams of developers create complex projects. Instead of lashing together ad hoc coding projects, you'll be creating systems of value to your organization and its clients.

If you're a freelancer or own a small technical consulting group or development shop, you'll love the power that MVC (and in particular, CodeIgniter) brings to your arsenal of tools. In this book, you'll learn how to combine CodeIgniter and MVC with Agile development processes to quickly build flexible and powerful applications in half the time with half the hassle.

Before you get too deeply into all that, it's time for a quick history lesson on MVC. The lesson is brief and provides some context for CodeIgniter. After this chapter, the focus is solely on CodeIgniter and how the processes and tools associated with it can help you build better applications.

A Brief History of MVC

Model-View-Controller was first described by Xerox PARC researchers working on the Smalltalk programming language in the late 1970s and early 1980s. Smalltalk was an object-oriented, dynamically typed, reflective programming language. Its first use was in the realm of educational learning, and it differed from mainframe data and control structures in that Smalltalk programs involved:

- Windowed user interfaces
- Object-oriented programming concepts
- Passing of messages between object components
- The ability to monitor and modify their own structure and behavior

In the Smalltalk world, the model knew nothing about the controllers or the views that worked with it. The model could be observed by the view and manipulated by the controller, but other than that, the model simply represented the problem domain. Whenever the model changed, it fired off messages to the views, so they could update themselves. Views themselves were subclasses of an existing `View` class

that knew how to render what you wanted to render. For example, if you wanted to draw a rectangle or a circle, you'd subclass an existing class for the appropriate rendering. Controllers, in the Smalltalk world, were originally conceived as being very thin, providing a means of sending messages to the model.

This is not a book on Smalltalk, but Smalltalk's impact on modern computing in general and programming in particular can't be overstated. The Macintosh, Windows, and X Windows development efforts have borrowed heavily from the Smalltalk windowed interface. In fact, the early Apple GUIs were indistinguishable from Smalltalk-80 v2 (as illustrated in Figure 1-2). Many of the text-based user interfaces in DOS used Smalltalk offshoots like DigiTalk (Figure 1-3).

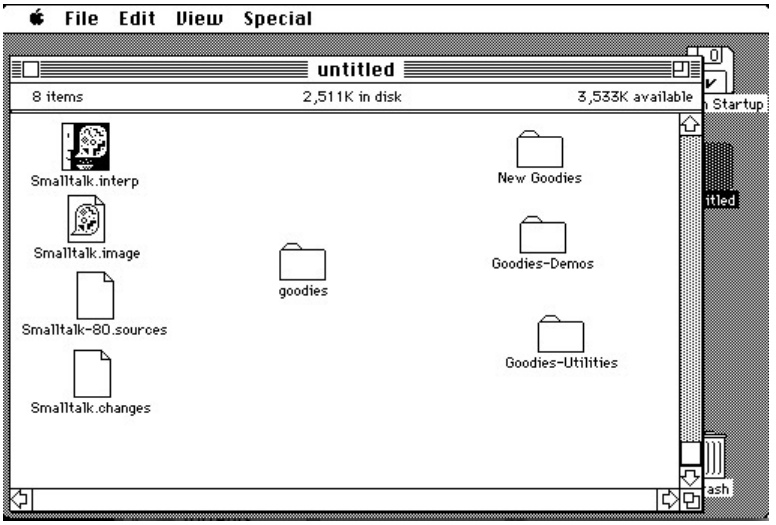

Figure 1-2

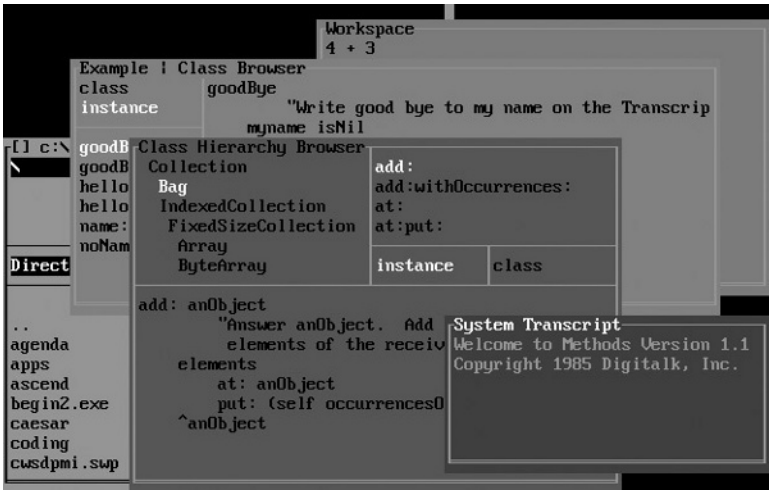

Figure 1-3

The next step in MVC (no pun intended) occurred with the arrival of the NeXT operating system and its software. NeXT was a company founded by Steve Jobs in the late 1980s that remained on the market until the early 1990s, when it was purchased by Apple. The NeXT MVC developers found a way to create ever more powerful views and ever more fine-grained Controllers (i.e., one could now track mouse movements and click events). This meant evolving the notion of the "thin" controller to a "fat" controller, as the controller took a more central role in the architecture. What's the difference? A *thin controller* is one that does just the basics and relies on models and views to do most of the work. A *fat controller* is the exact opposite, a software design choice that puts most if not all the processing responsibility in the controller.

On the desktop front, languages like Cocoa (on Mac OS X) provided developers with an MVC framework. The Cocoa MVC framework simplified the pattern by routing all messages through the controller. The controller updated the model, received a notification from the model, and then updated the view, thus mediating the flow of data.

With Sun's Model2 approach to MVC, controllers evolved again, mapping user requests to actions. Model2 featured reusable (but stateless) views, fat controllers, and independent models. However, it became quickly apparent that some of MVC's historically relevant functionality (such as views that updated themselves after receiving a message from the model) didn't really map to the stateless request/response world of HTTP.

But that didn't stop the emergence of MVC frameworks and languages for the Web. So far, there have been Django, Struts, and most famously, Ruby on Rails. In fact, if you're a PHP developer, you've probably been watching the Ruby on Rails phenomenon with a mixture of curiosity and dread. It appears from a distance that tools like Ruby on Rails allow small teams of developers to create web applications in a tenth the time, with less debugging and fewer headaches. Ruby on Rails web sites have slick interfaces, have easy to understand URLs (no more query strings!), and are generally more compact and secure than the sites you've been creating with your own custom PHP.

One overt design strategy for Ruby on Rails is the idea of "convention over configuration" — in other words, is there a way to create a set of defaults that works 80 to 90 percent of the time, thereby reducing the amount of configuration you have to keep track of? This pervasive idea has been incorporated into every single PHP framework, CodeIgniter included. The result has been an enormous amount of time saving and effort reduction in development.

From the perspective of the PHP community, several difficulties remain with Ruby on Rails — many PHP programmers don't have the time or desire to switch to it, hosting is not as ubiquitous as with PHP, and the strict database design conventions make porting legacy applications harder. All of these create an understandable reluctance on the part of the PHP programmer to switch. After all, you create and support PHP applications, and you don't feel ready to take on a brand-new programming language and the learning curve associated with it. Even if the language is easy to learn, there are still many things about it that you haven't mastered, and doing so on the client's dime seems wasteful and unethical.

Luckily for you, several PHP MVC frameworks have emerged in the past few years, among them CakePHP, Symfony, and CodeIgniter. The next section provides a brief comparison of these three PHP MVC frameworks.

Comparing PHP MVC Frameworks

When you look at CodeIgniter, Symfony, and CakePHP, you'll notice quite a few similarities. For example, all three:

- Allow you to create models that bind to a data source, views that display content, and controllers that monitor user action and allow updates of models and views.
- Use structured folders to separate your application's components from a central core.
- Use configuration files to help you maintain vital metadata, such as the location of the database, base URLs, and the like.
- Use controller callback functions that are surfaced to the GUI. If the user clicks on a link that opens /post/view/3, then the `view()` callback function in the post controller is called, with the ID of 3 passed to the model for data retrieval.
- Allow you to extend the core code and build your own libraries and helper functions.

The next few sections provide an extremely concise summary of the differences between CodeIgniter, CakePHP, and Symfony. The intent here is not to list every single difference among them, nor to proclaim that any one MVC framework is better than the others. The intent is to discuss the differences briefly, thereby giving you a more realistic appraisal of where CodeIgniter fits into the scheme of MVC thinking.

Making Life Easier for the Developer

CakePHP's automatic approach allows the developer to create web applications quickly. For example, the automation allows a model to map easily to a database table. If you don't really care how things work underneath, this approach can be strangely liberating. However, if you really want to understand what is happening, you sometimes have to dig around in the core code.

Symfony took a page from Ruby on Rails, providing developers with a series of command-line tools to help build admin panels, object-relational mapping schemes, views, and other niceties. Run a few scripts and suddenly you have a surprising amount of work done for you.

CodeIgniter has a different approach. Just about all your work will be in the controller. That's where you load libraries, extract data from the model, and pull in views. Everything's in plain sight, so it's easy to keep things organized and troubleshoot problems if and when they occur. The drawback, of course, is that beginners tend to create very general (and hence very fat) controllers that can become more difficult to maintain as time goes on.

Models

CakePHP automatically loads a model that matches the current controller. Yes, you can turn this off, but many CakePHP development efforts leave in this admittedly useful feature. CakePHP also establishes all the model associations for you, allowing developers to quickly pull in, for example, all comments associated with a blog post after only minimal association on the developer's part. This approach requires strict naming conventions for models, views, controllers, and database tables (not to mention primary key and foreign key fields). It can get a bit frustrating to remember to use the singular form of the database table for your model (your model is person, and the database table is people), but it can all be overridden. And yes, you can use raw SQL queries if you need to.

Figure 1-4 illustrates a typical CakePHP model.

Example User Model, saved in /app/models/user.php

```
<?php

//AppModel gives you all of Cake's Model functionality

class User extends AppModel
{
    // Its always good practice to include this variable.
    var $name = 'User';

    // This is used for validation, see Chapter "Data Validation".
    var $validate = array();

    // You can also define associations.
    // See section 6.3 for more information.

    var $hasMany = array('Image' =>
                   array('className' => 'Image')
                   );
```

Figure 1-4

Symfony's approach allows you to use either built-in methods or raw queries to access the database abstraction layer. Symfony has several built-in methods to create, retrieve, update, and delete database records. Sometimes, though, it makes more sense to use raw SQL (e.g., you want to return a count on a number column). Using raw SQL (or synthetic SQL) involves the following steps: getting connection, building a query string, preparing and executing a statement, and iterating on the result set returned by the executed statement.

CodeIgniter's approach is a bit more manual, but a lot more flexible. There is no standard naming convention for models and controllers. Developers can manually load models or autoload them in the main configuration, and the models don't have to be named a certain way to match up to tables. What this means is that legacy applications are extremely easy to port over to CodeIgniter, and integration with outside database systems is generally very easy.

CodeIgniter also allows a great deal of flexibility when querying tables. For example, given a database table called *users*, the following approaches will all work if you want to extract all records:

```
//place raw SQL in the query() method
$q = $this->db->query("select * from users");

//or pass in a variable
$sql = "select * from users";
$q = $this->db->query($sql);

//or use the built-in get() method
$q = $this->db->get('users');
```

Throughout the book, you'll see how easy it is to create, delete, and update database records. (In fact, some of you may think that the repetition of the key database queries might be too much, but getting the basics down is the right foundation for your development success.) You can use SQL or CodeIgniter's built-in methods to do this.

Strictly speaking, CodeIgniter doesn't require models at all. Although this may seem a bit confusing at first (after all, how is it an MVC framework without the M part?), dropping this requirement gives you a lot of flexibility. For example, it could prove useful if you're developing applications without a database or XML backend or if you need to prototype something quickly using placeholder data in your controller.

Views

CakePHP uses layouts with placeholders (`title_for_layout` and `content_for_layout`) for other views. This automatic approach is pretty handy, but again, it requires a certain level of discipline in your file naming. Of course, it's easy enough to override this feature and use any kind of naming convention you want.

Symfony uses templates that support XHTML and PHP snippets, along with a variety of helper functions [such as `input_tag()` and `link_to()` to help you create text inputs and links, respectively]. Symfony also supports the use of separate layout files that can be assigned to templates.

CodeIgniter's approach is, predictably, more straightforward and flexible. You can almost think of view files as includes. You can load them one at a time with your controller and then render the pieces into a final view, or you can just use a master view with placeholders. Layout is typically controlled with CSS, or you can use a lightweight templating class that uses pseudo-variables in much the same way that SMARTY does. There are also many helper functions available to help you create forms, links, and other HTML elements. The preferred template format is pure PHP, which means that it's easy for PHP programmers to build views and then optimize them.

Helpers and Libraries

CakePHP is pretty light on built-in helpers and libraries, but it makes up for it with the Bakery, an online resource full of tutorials, snippets, add-ons, and third-party classes. However, as previously mentioned, it comes with a nice set of query methods, such as `find()`, `findAll()`, `query()`, `findBy<fieldname>()` (where `fieldname` is your database field name), and more.

Symfony's suite of tools for unit testing, scaffolding, and admin generation is probably the best of all three. These tools may not be used every day, but they offer plenty of shortcuts for the developer.

CodeIgniter comes with an amazing range of libraries and helper files that cover a lot of what you will encounter in just about every project: caching, security, file uploads, link building, form building, text parsing, regular expressions, database queries, FTP, e-mail, calendaring, sessions, pagination, image manipulation, validation, XML-RPC, dates, cookies, XML, and more. You can also extend built-in libraries and helpers and create your own.

Revisiting the Opening Example

Now that you've got the basics of MVC down and understand how CodeIgniter differs from other PHP-based MVC frameworks, it's time to revisit the opening example. Instead of struggling with the original code, imagine that you had access to CodeIgniter and you were recoding it from scratch.

In the following sections, you see how you'd go about doing that. The examples provided in the next few pages don't stop long enough to explain how to install and configure CodeIgniter (that's Chapter 3) or talk about all the advanced features like caching, routing, and the extensive libraries. The assumption is that all that's been done for you, and that you have a properly working CodeIgniter environment. The goal is to show you, at high speed, how a CodeIgniter application might look and behave.

First Things First: The Model

Depending on your work style, you may be inclined to tackle the views or controller first, but a good place to start is the model. Why? Because the model serves as the data foundation for your entire web application. Many of the controller's functions will rely on the model, so it's a good idea to have those tools all ready to go.

All CodeIgniter models have the same initial structure:

```
<?php
class Page_model extends Model{

  function Page_model(){
    parent::Model();
  }
}
?>
```

Notice that the name of the model (in this case, Page_model) is both the name of the class and the name of the initializing function. This file is stored in the /system/application/models/ folder of your project, more than likely with a name like *page_model.php*. If this were the user model, you'd likely call your model User_model and store it in the file user_model.php.

Later in the book, you learn the ins and outs of placing and naming files. But here, you need to add a few functions to your model. The first function is the code that retrieves the home page:

```
<?php
class Page_model extends Model{

  function Page_model(){
    parent::Model();
  }

  function fetchHomePage(){
    $data = array();
    $options = array('status' => 'live', 'type'=> 'home');
    $q = $this->db->getwhere('pages', $options, 1);

    if ($q->num_rows() > 0){
      $data = $q->row_array();
    }
    return $data;
    $q->free_result();
  }
}
?>
```

The `fetchHomePage()` function is very simple, but it pays huge dividends to understand what is going on. Here's what's going on in the `fetchHomePage()` function, step by step:

1. The first line initializes the `$data` array. You're not required to do this, but doing so is good practice and is an effective way to avoid unnecessary carping by PHP if you end up returning a null set at the end of the function.
2. The second line establishes a list of options that will be passed to the `getwhere()` method. In this case, the options set are any status fields marked as "live" and any type fields marked as "home." The `getwhere()` method is built in to CodeIgniter and allows you to extract data from a table while passing in an array that serves as the `where` clause in the SQL statement.
3. The `$this->db->getwhere()` line is where the model extracts the data from the database table. You need to pass in three arguments to this method:
   a. The first argument is the name of the table.
   b. The second argument is the list of options previously discussed.
   c. The third argument is how many records you want to extract (in this case, the limit is set to 1).
4. After the query runs, use the `num_rows()` method to make sure you're getting back the number of rows you want (in this case, anything more than zero) and then dump the fields from the result set into an array with `row_array()`. In other chapters, you see how to loop through multiple rows in a result set using `result()`. In this particular example, each field from the database gets placed into the `$data` array that was initialized at the top of the function.
5. Finally, the function returns `$data` and then frees the memory being used by the result set. It's not necessary to do this, because PHP will clear out all result set objects when page execution ends, but having numerous result set objects might slow down your application. It's a good habit to get into.

Now that you understand what's going on in the model, it's time to turn your attention to the controller. Basically, everything you've done in the model will become reusable data-fetching functions for the controller's use.

Creating the Controller

Now that a working model is in place, it's time to create a controller. Controllers in CodeIgniter function as the brawn of your application. Anything that a user can do on your site, including going to destinations, should be represented in your controller.

First, here's what a standard controller would look like. Notice that in the following example the controller is for the Page application you built the model for above:

```
<?php
class Page extends Controller {
  function Page(){
    parent::Controller();
  }
}
?>
```

As before, this controller is bare-bones, consisting of just an initialization function that ties this particular controller to the master class Controller. Once again, notice that with this simple notation, CodeIgniter allows you to extend the basic core classes and create something specific and powerful with very little work.

When you're working with a controller, every function or method maps to an address in your application. If you want users to view a page with the address of `/page/foo`, there had better be a Page controller and a `foo()` method inside that controller.

For now, all that's needed is an `index()` function to represent the site's home page.

```
  function index(){
    //code goes here
  }
```

Right now, if a user were to go to your home page, they'd see a blank page. That's because nothing particularly interesting is happening inside the `index()` method. To change that, just simply load the `Page_model` so you can access the `fetchHomePage()` method you created previously.

```
<?php
  class Page extends Controller {
    function Page(){
      parent::Controller();
    }
  }

  function index(){
    $this->load->model('Page_model','',TRUE);
    $data['content']  = $this->Page_model->fetchHomePage();
  }
}
?>
```

Notice that once you load the model you want, you can access the methods inside that model by simply invoking the name of the model: `$this->Page_model->fetchHomePage()`. Storing the information from the database into the `$data` array makes it easy to display the information in a view.

You've loaded the model and invoked the method to retrieve the data; now all you need to do is print out the data to the screen. In extremely simple applications, you could get away with simply printing out what you need right there in the controller, but that isn't smart to do unless you need a quick and easy debugging method.

The best way is to load a view. You haven't created one yet, but for now, use the name `home`. Later you can create a home.php file in the Views directory. When you load a view, the name of the file (sans *.php*) is the first argument, and any data you want to pass in are the second argument. Here's what the entire controller looks like:

```
<?php
  class Page extends Controller {
    function Page(){
      parent::Controller();
    }
```

```
    }

    function index(){
      $this->load->model('Page_model','',TRUE);
      $data['content'] = $this->Page_model->fetchHomePage();
      $this->load->view('home',$data);
    }
  }
  ?>
```

When you look at this controller, notice how organized it is, and how easy it is to figure out what is going on. Any visitor to the site's index or home page will kick off a discrete process: Invoke the model, retrieve the data, and display the view with data. There's nothing hidden about any of it; there are no other configurations to look up. Simple, clean, tidy.

Creating the View

The final step, creating the view, is quite possibly the easiest. Simply create the appropriate file in the /system/application/views/ folder (in this case, a file called home.php):

```
<html>
  <head>
    <title><?php echo $content['title']; ?></title>
    <link href="<?php echo $content['css']; ?>" rel="stylesheet" type="text/css"/>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <meta name="keywords" value="<?php echo $content['keywords'];?>"/>
    <meta name="description" value="<?php echo $content['description'];?>"/>
  </head>
  <body>
    <h1><?php echo $content['title'];?></h1>
    <p><?php echo nl2br($content['bodycopy']);?></p>
  </body>
</html>
```

Please note that the `$content['css']` in this code example is just a file name! You're not storing the entire CSS file in the database. You could if you wanted to, but it makes more sense to keep this information in a separate file.

Notice that the view is mostly HTML markup with PHP commands interspersed. The view contains a title tag (with dynamic content pulled in from the database), a dynamically inserted CSS filename (again, from the database), and other information, such as the UTF-8 charset declaration. You may be wondering what the point of all this is if you're eventually going to have PHP mixed in with your HTML. It's important to understand that:

- You have a lot less PHP in your HTML at this point.
- All the important business logic and data extraction have occurred in the controller and model.
- Because the view is PHP, it can be cached by CodeIgniter or by another server-side optimization process.

Notice the use of the `$content` array whenever you access what you need. You may be wondering about that. Take a look at the model again:

```
if ($q->num_rows() > 0){
  $data = $q->row_array();
}
```

When the `fetchHomePage()` function retrieves the home page content from the pages table, notice the use of `row_array()`, which converts the results into an array. The keys of that array are the field names, with field values populating the array values. In other words, the `row_array()` method is a simple way of keeping parity between your database tables and result set objects.

If you were to inspect the `$data` array within the model with `print_r()`, you'd probably see something like this:

```
Array
(
  [id] => 4
  [title] => test
  [keywords] => test
  [description] => test
  [status] => live
  [bodycopy] => test
  [css] => default.css
)
```

Once called from the Page controller, however, this array is dropped inside `$data['content']`, which, when passed to the view, is accessible via the `$content` array. Why? Because you told the template to accept the `$data` array when you loaded the view:

```
function index(){
    $this->load->model('Page_model','',TRUE);
    $data['content'] = $this->Page_model->fetchHomePage();
    $this->load->view('home',$data);
  }
```

Once the view is loaded, it unwraps whatever is in `$data` (itself an array), and the first thing it encounters is the `$content` array embedded inside it. You see this kind of approach taken throughout the book, so it becomes second nature to you.

Figure 1-5 illustrates the application flow.

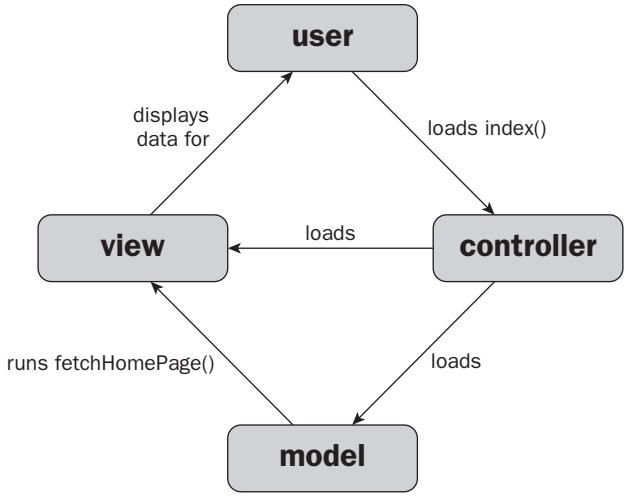

Figure 1-5

A Slightly Different Approach: Templates

Along with standard PHP templates, CodeIgniter offers a lightweight template parser. To use the parser, all you have to do is load it into your controller and pass in the data from the model. The view would then use pseudo-variables instead of actual PHP variables, thus making your views free of PHP altogether. If you are familiar with SMARTY templates, you will see that these pseudo-variables and pseudo-structures align with your expectation of how the SMARTY world works. If you are new to this type of template parsing, you won't have any direct experience with it, but it's pretty easy to pick up. For example, instead of calling `$title` as you did before, you'd use `{title}`. Semantically, the structures and naming conventions are similar enough to the PHP variables and structures that you won't have much trouble following along.

One final note before continuing the discussion: There's normally no need to modify the model if you're going to work with the template parser. Why not? Well, it just so happens that the template parser will accept the data structures you give it. The only thing that changes is how you call out and use the data structure from the parsed template.

Using Third-Party Templating Systems

Most of the time, you'll want to use PHP templates with CodeIgniter, as it offers the fastest way to both develop and generate views. However, there might be some cases in which you'll want to use a third-party templating system, such as SMARTY.

Why use a templating system? Some developers swear by them, as they allow a clean separation between PHP code and the pseudo-variables that designers and front-end developers can be trained to use without harming the rest of the application.

In most cases, these third-party templating systems work like CodeIgniter's template parser. They almost always use pseudo-code and pseudo-variables in place of "real" PHP code. If you've never worked with pseudo-code, it looks a bit like this:

```
<title>{title}</title>
<h1>{header}</h1>
```

You would normally pass variable structures into the template via your controller (establishing values for `$title` and `$header` above) so they can be replaced by their corresponding pseudo-variables.

The biggest template engine out there is SMARTY (available at `http://smarty.php.net`). Others include TinyButStrong (`www.tinybutstrong.com`) and SimpleTemplate (`http://sourceforge.net/projects/simpletpl`).

One of the common pitfalls of using third-party templating solutions is that they can provide too much complexity and overhead (both from maintenance and performance standpoints) with minimal return on convenience.

Modifying the Controller

Here's what the controller would look like:

```
<?php
  class Page extends Controller {
    function Page(){
      parent::Controller();
    }
  }

  function index(){
    $data = array();
    $this->load->library('parser');
    $this->load->model('Page_model','',TRUE);
    $data = $this->Page_model->fetchHomePage();
    $this->parser->parse('home',$data);
  }
}
?>
```

Note the two additions to the `index()` method: loading the parser library and then calling the parser, passing in as arguments the name of the view and the data retrieved from the model. Also note that the data from the model can be safely stored in the `$data` array (which has also helpfully been initialized). In the view, you'll name your pseudo-variables from the keys of this array, which helpfully map to the names of the database table fields. For example, `$data['css']` can be accessed with `{css}`.

Modifying the View

Here's the view, with pseudo-variables in place instead of real PHP variables.

```
<html>
  <head>
   <title{title}</title>
   <link href="{css}" rel="stylesheet" type="text/css"/>
   <meta name="keywords" value="{keywords}"/>
   <meta name="description" value="{description}"/>
  </head>
  <body>
    <h1>{title}</h1>
    <p>{bodycopy}</p>
  </body>
</html>
```

This approach certainly involves a lot less typing, and the overall aesthetics can be very familiar and pleasing to anyone who's worked with a template system in the past. Also, some HTML specialists who are unfamiliar with PHP find this format a bit easier to work with.

What about `{bodycopy}`?

However, notice that the `{bodycopy}` pseudo-variable in the example has lost some of its functionality. In the previous PHP incarnation of this template, the bodycopy data were processed via `nl2br()`, a PHP function that converts line breaks into HTML `<br/>` tags.

One way to fix the problem is to add two lines of processing to your controller, like so:

```
function index(){
    $data = array();
    $this->load->library('parser');
    $this->load->model('Page_model','',TRUE);
    $data = $this->Page_model->fetchHomePage();

    //fix the body copy
    $fixcopy = nl2br($data['bodycopy']);
    $data['bodycopy'] = $fixcopy;

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

Of course, you could also rewrite the model, replacing `row_array()` with `row()` and then processing each field as it comes off the database query. That way you can process every element as it is extracted from the database.

```
<?php
class Page_model extends Model{

  function Page_model(){
    parent::Model();
  }

  function fetchHomePage(){
    $data = array();
    $options = array('status' => 'live', 'type'=> 'home');
    $q = $this->db->getwhere('pages', $options, 1);

    if ($q->num_rows() > 0){
      $row = $q->row();
      $data['title'] = $row->title;
      $data['css'] = $row->css;
      $data['keywords'] = $row->keywords;
      $data['description'] = $row->description;
      $data['bodycopy'] = nl2br($row->bodycopy);

    }
    return $data;
    $q->free_result();
  }
}
?>
```

In a lot of ways, this is a more palatable solution, because now the model does all the work of preparing the data for display. However, others might argue that there may be a need in the future to have access to the raw `bodycopy` field unprocessed by `nl2br()`. In that case, you'd want to keep the processing in the controller or create a second method in the model to address this issue.

There is a much better way to do all this, of course, and it involves using a built-in CodeIgniter helper called *Typography*. This helper has an `auto_typography` function that does all the hard work of converting line breaks to `<br/>` tags and converting quotes and dashes to the appropriate entities.

First, load the helper in the controller function:

```
function index(){
  $data = array();
  $this->load->library('parser');
  $this->load->model('Page_model','',TRUE);
  $data = $this->Page_model->fetchHomePage();

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

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

Next, use the `auto_typography()` function in the view:

```
<html>
  <head>
    <title{title}</title>
    <link href="{css}" rel="stylesheet" type="text/css"/>
    <meta name="keywords" value="{keywords}"/>
    <meta name="description" value="{description}"/>
  </head>
  <body>
    <h1>{title}</h1>
    <p><?php echo auto_typography("{bodycopy}");?></p>
  </body>
</html>
```

If you put five MVC experts in the same room and asked them to tackle this particular problem, you'd probably get at least three different answers, if not more. But you get the idea: CodeIgniter is fairly easy to work with, very flexible, and leaves you, the developer, with easily maintained, well-organized code artifacts.

Conclusion

This chapter serves as a basic introduction to MVC and CodeIgniter's place in that world. You've learned about the concepts behind MVC and why you should use it, and been given a brief history lesson. Specifically, you've learned how to transform a non-MVC application into a working CodeIgniter application, with a simple model, controller, and view.

As you continue with the rest of the book, remember these key facts about Model-View-Controller frameworks and applications:

- Models maintain and update an application's data.
- Views display data and user interface elements.
- Controllers handle user events that manipulate models and render or update views.
- In CodeIgniter:
  - Most of your work will be done in the controller.
  - Views can be regular PHP files or parsed templates with pseudo-variables.
  - Models aren't required, but most of your applications will have them.

The next two chapters cover Agile development practices and provide a high-level overview of CodeIgniter's structure and installation process. By the time you finish with Chapters 1–3, you'll have the necessary background for creating the projects outlined in this book.