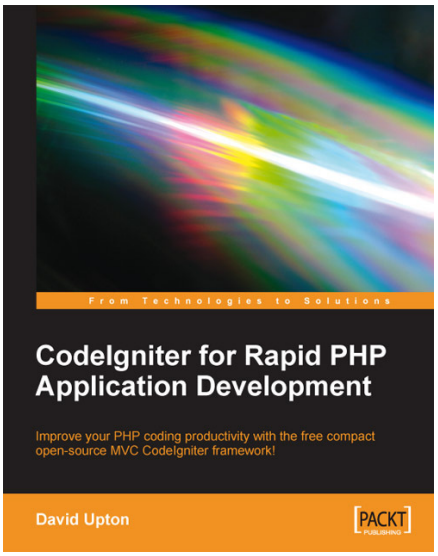
CodeIgniter for Rapid PHP Application Development

Improve your PHP coding productivity with the free compact open-source MVC CodeIgniter framework!

David Upton



Chapter No. 4
"Using CI to Simplify Databases"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter No. 4 “Using CI to Simplify Databases"

A synopsis of the book’s content

Information on where to buy this book

About the Author

David Upton is a director of a specialized management consultancy company, based in London but working around the world. His clients include some of the world's largest companies. He is increasingly interested in web-enabling his work, and seeking to turn ideas into robust professional applications by the simplest and easiest route. He has so far written applications for two major companies in the UK. His other interests include simulation, on which he writes a weblog that takes up far too much of his time, and thinking.

CodeIgniter for Rapid Application Web Development: Improve your PHP Coding productivity with the free compact open-source MVC CodeIgniter framework!

This book sets out to explain some of the main features of CI. It doesn't cover them all, or cover any of them in full detail. CI comes with an excellent on-line User Guide that explains most things. This is downloaded with the CI fi les.

This book doesn't try to duplicate the User Guide. Instead it tries to make it easier for you to pick up how the CI framework works, so you can decide whether it is right for you, and start using it quickly.

In some places, this book goes beyond the User Guide, though, when it tries to explain how CI works. (The User Guide is more practically oriented.) This means that there are some fairly theoretical chapters in between the "here's how" pages. I've found that it helps to understand what CI is doing under the hood; otherwise you sometimes get puzzling error messages that aren't easy to resolve.

I've tried to use a 'real-world' example when showing sections of CI code. I want to show that CI can be used to develop a serious website with a serious purpose. I'm currently running several websites for clients, and I want a program that will monitor them, test them in ways I specify, keep a database of what it has done, and let me have reports when I want them.

The examples in this book don't show it in full detail, of course: but they do, I hope, demonstrate that you can use CI to make pretty well any common coding simpler, and some uncommon stuff as well.

This book steps you through the main features of CodeIgniter in a systematic way, explaining them clearly with illustrative code examples.

What This Book Covers

Chapter 1 explains what CodeIgniter can do, the 'framework', and how CodeIgniter fits in. It further talks about the open-source business model and gives some disadvantages of CodeIgniter, at the end.

Chapter 2 explains what happens when you install the site, and which files will be created. It gives a detailed overview of the required software, and explains the basic configuration of CodeIgniter.

Chapter 3 explains how MVC helps to organize a dynamic website. It goes further to explain the process by which CodeIgniter analyzes an incoming Internet request and decodes which part of your code will handle it. Then CodeIgniter syntax rules and the different types of files or classes you can find—or write for yourself—on a CodeIgniter site are explained. At the end of the chapter, some practical hints on site design are given.

Chapter 4 looks at how you set up a database to work with CodeIgniter, and then how you use the Active Record class to manipulate the database.

Chapter 5 covers various ways of building views, how to create HTML forms quickly, and how to validate your forms using CodeIgniter's validation class.

Chapter 6 looks at one of the basic questions affecting any website i.e. session management and security; we also explore CodeIgniter's session class.

Chapter 7 covers the way in which CodeIgniter uses objects, and the different ways in which you can write and use your own objects.

Chapter 8 covers CodeIgniter classes to help with testing: Unit tests, Benchmarking, the 'profiler' and ways in which CodeIgniter helps you to involve your database in tests without scrambling live data.

Chapter 9 looks at using CodeIgniter's FTP class and email class to make communication easier, and then we venture into Web 2.0 territory using XML-RPC.

Chapter 10 talks about CodeIgniter classes that help in overcoming problems arising regularly when you are building a website, for example, the date helper, the text and inflector helpers, the language class, and the table class.

Chapter 11 looks at several useful CodeIgniter functions and helpers: file helper, download helper, file upload class, image manipulation class, and the ZIP class.

Chapter 12 covers exploring your config files, using diagnostic tools, and potential differences between servers, along with some notes on security.

Chapter 13 shows you how to generalize CRUD operations so that you can do them with two classes: one for the controller, and one for the CRUD model.

Chapter 14 looks at some coding examples, bringing together a lot of the functions that have been discussed bit by bit in the preceding chapters.

Chapter 15 looks at some of the resources available to you when you start to code with CodeIgniter, such as the libraries for AJAX and JavaScript, authentication, charting, and CRUD.

4
Using CI to Simplify Databases

You're looking at CI because you want to make coding easier and more productive. This chapter is about CI's **Active Record** class. If CI offered nothing more than its Active Record class, it would still be worth every penny of the purchase price. All right, it's free. I'll rephrase that—it would still be a major tool to increase your productivity.

Active Record allows you to handle databases with a minimum of fuss and a maximum of clarity. It's simple to use and maintain.

This chapter looks at how you set up a database to work with CI, and then how you use the Active Record class to manipulate the database. You'll see:

- How Active Record code compares with 'classic' PHP/ MySQL interface code
- How to write 'read' queries, and display the results
- How to do create, update, and delete queries

CI allows you to write queries in the traditional 'classic' PHP style as well, but I won't go into detail on that. It's fully covered in the online User Guide. I started off doing it the old way, but once I tried Active Record, I never looked back.

Configuration Settings

You have probably noticed that most chapters in this book keep going back to the `system/application/config` folder and the configuration files inside it. These are pretty essential for controlling the way CI works. And while you can leave most of them safely set at the defaults, the `database` config file does need tweaking before anything will work at all.

Basically, you just have to tell it where your database is, and what type it is. The default file simply says:

```
$active_group = "default";
$db['default']['hostname'] = "";
$db['default']['username'] = "";
$db['default']['password'] = "";
$db['default']['database'] = "";
$db['default']['dbdriver'] = "";
```

along with a few other options that you can leave at the default, for now. The options you must fill in are:

- `hostname`: The location of your database, e.g., 'localhost' or an IP address
- `username` and `password`: The username and password of a database user with sufficient permissions to do whatever you may want your site to do. This is not (usually) the same username and password as your site or your ISP's control panel.
- `database`: The name of your database, e.g., 'websites'
- `dbdriver`: The type of database you're using—at the time of writing, the options CI offers were MySQL, MySQLi, Postgre SQL, ODBC, and MS SQL.

In my experience, one of the most difficult things to set up on a new CI site can be the link to the database. You may need to consult your ISP if in doubt—sometimes their database runs at a different address to their web servers. If you are using MySQL, they may offer phpMyAdmin, which usually tells you the hostname—this may be 'localhost' or it may be an IP address.

You'll note that this part of the `config` file is actually a multi-dimensional array. Within `$db` is an array called `default`, and you're adding key/variable pairs like `hostname = 127.0.0.1` to that array. This is so that you can set up other databases, as other secondary arrays, and swap between them easily by simply changing the `$active_group` setting to the name of another array.

This makes it possible to run a site with several database options—for instance, a test database and a production database—and to swap between them easily. Or you might need to draw information from two separate databases.

Designing the Database for Our Site

I want to show that CI can be used to develop a serious website with a serious purpose. I am currently running several websites for clients, and I want a program that will monitor them, test them in ways I specify, keep a database of what it has

done, and let me have reports when I want them. So let's try to build it. Firstly let's set some objectives. These are:

1. To manage one or more remote websites with a minimum of human intervention
2. To run regular tests on the remote sites
3. To generate reports on demand, giving details of the site and of tests conducted

So, the first thing we will need is a database of websites to check. Set up a database called `websites` in MySQL or whatever RDBMS you're using.

Now, we need to add some tables to hold various types of data. Let's add to our `websites` database a table for sites, which includes fields for their URLs, their names and password/usernames, and their types. We'll also include an `id` field for each site—and in MySQL at least, which can be set to generate a unique new ID for each entry, using the auto-increment field type.

Each site may be hosted with a different host, or host machine, and we need another `hosts` table to store data about this. It most probably has a domain associated with it, so we need a `domains` table to keep track of data about the domain, like when it's due for renewal, the registrar, and our username/password on the registrar site. Then of course, we have those tiresome people, clients, some of whom may own more than one site, so we need a separate `people` table in which to store their names, email addresses, snail mail addresses, mobile numbers, plus pet's names, and all the other stuff so vital to good CRM.

So our site table needs to include fields for a domain ID, a host ID, and perhaps a couple of people IDs, one for the site owner or client and one for the site manager. (That's you, or one of the staff you'll have to hire to keep pace when this app hits the market.)

As you can see, this is a full relational database, and we've only just started! (Fuller details of this database are set out as an appendix to this chapter, in the form of a MySQL query, if you want to set one up yourself.)

We're going to want a simple flexible way of accessing all this. So, let's turn to what CI can offer, and in particular to its Active Record class.

Active Record

'Active Record' is a 'design pattern'—another of those highly abstract systems like MVC, which provide templates for solving common coding problems and also generate some of the dullest books on the planet. In itself, it isn't code, just a pattern

for code. There are several different interpretations of it. At its core is the creation of a relationship between your database and an object, every time you do a query. Typically, each table is a class, and each single row becomes an object. All the things you might want to do with a table row—create it, read it, update it, or delete it, for example—become 'methods', which that object inherits from its class. Ruby on Rails is built around the Active Record pattern, and so is CI—although the exact implementation in the two frameworks seems to have subtle differences.

Enough theory—what does it mean? Well, simple and clear code statements, if you don't mind putting arrows in them.

Advantages of Using the Active Record Class

Active record saves you time, brings in automatic functionality that you don't have to think about, and makes SQL statements easy to understand.

Saving Time

When you write a normal database query in PHP, you must write a connection to the database each time. With CI, you connect once to the database, by putting the following line in the constructor function of each controller or model:

```
$this->load->database();
```

Once you've done this, you don't have to repeat the connection, how many ever queries you then make in that controller or model.

You set up the database details in the config files as we saw earlier in this chapter. Once again, this makes it easier to update your site, if you ever change the database name, password, or location.

Automatic Functionality

Once you've connected to the database, CI's active record syntax brings hidden code with it. For instance, if you enter the following 'insert' query:

```
$data = array(
                'title' => $title,
                'name' => $name,
                'date' => $date
            );
$this->db->insert('mytable', $data);
```

the values you are inserting have been escaped behind the scenes by this code:

```
function escape($str)
{
```

```
    switch (gettype($str))
    {case 'string':
    $str = "'".$this->escape_str($str)."'";
    break;
    case 'boolean':     $str = ($str === FALSE) ? 0 : 1;
    break;
    default       :     $str = ($str === NULL) ? 'NULL' : $str;
    break;
    }
return $str;
}
```

In other words, the CI framework is making your code more robust. Now, let's look at how it works.

Firstly, connecting to the database is very simple. In classic PHP, you might say something like this:

```
$connection = mysql_connect("localhost","fred","12345");
 mysql_select_db("websites", $connection);
 $result = mysql_query ("SELECT * FROM sites", $connection);
 while ($row = mysql_fetch_array($result, MYSQL_NUM))
 {
    foreach ($row as $attribute)
       print "{$attribute[1]} ";
 }
```

In other words, you have to re-state the host, username, and password, make a connection, then select the database from that connection. You have to do this each time. Only then, do you get on to the actual query. CI replaces the connection stuff with one line:

```
$this->load->database();
```

which you put once, in each controller or model or class constructor that you write. After that, in each function within those controllers, etc., you just go straight into your query. The connection information is stored in your `database` config file, and CI goes and looks it up there each time.

So, in each CI function, you go straight to your query. The query above written in CI comes out as:

```
$query = $this->db->get('sites');
   foreach ($query->result() as $row)
   {
      print $row->url
   }
```

Simple, isn't it?

The rest of this chapter sets out ways of making different queries, making them more specific.

Read Queries

The most common query that we'll write simply retrieves information from the database according to our criteria. The basic instruction to perform a read query is:

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

This is a 'SELECT *' query on the `sites` table—in other words, it retrieves all the fields. If you prefer to specify the target table (`sites`) in a separate line, you can do so in this way:

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

If you want to 'SELECT' or limit the number of fields retrieved, rather than get them all, use this instruction:

```
$this->db->select('url','name','clientid');
$query = $this->db->get('sites');
```

You may want to present the results in a particular order—say by the site name—in which case you insert (before the `$this->db->get` line):

```
$this->db->orderby("name", "desc");
```

`desc` means in descending order. You can also choose `asc` (ascending) or `rand` (random).

You may also want to limit the number of results your query displays; say you want only the first five results. In this case insert:

```
$this->db->limit(5);
```

Of course, in most queries, you're not likely to ask for every record in the table. The power of databases depends on their ability to select—to pick out the one piece of data you want from the piles of stuff you don't. This is usually done by a `where` statement that CI expresses in this way:

```
$this->db->where('clientid', '1');
```

This statement would find all websites linked to the client whose ID was `1`. But that's not much help to us. We don't want to remember all the ID's in our `people` table. As humans, we prefer to remember human names. So we need to link in the `people` table:

```
$this->db->from('sites');
$this->db->join('people', 'sites.peopleid = people.id');
```

For each people ID in the `sites` table, look up the information against that ID in the `people` table as well.

Note the SQL convention that if a field name may be ambiguous between two tables, you reference it with the table name first, then a period, then the field name. So `sites.peopleid` means the `peopleid` field in the `sites` table. In fact, there isn't a field called `peopleid` in both tables, but there is an `id` field in both `sites` and `people`, so the RDBMS will protest if you try to run the query without resolving the ambiguity for it. In any case, it's a good habit to make your meaning explicit, and CI syntax happily accepts the fuller names.

You can play around with the syntax of `where` statements. For instance, add negation operators:

```
$this->db->where('url !=','www.mysite.com' );
```

or comparison operators:

```
$this->db->where('id >','3' );
```

or combine statements ("WHERE… AND…"):

```
$this->db->where('url !=','www.mysite.com');
$this->db->where('id >', '3');
```

or use `$this->db->orwhere()` to search for alternatives ("WHERE … OR"):

```
$this->db->where('url !=','www.mysite.com' );
$this->db->orwhere('url !=','www.anothersite.com' );
```

So let's say we've built up a query like this:

```
$this->db->select('url','name','clientid','people.surname AS client');
$this->db->where('clientid', '3');
$this->db->limit(5);
$this->db->from('sites');
$this->db->join('people', 'sites.clientid = people.id');
$this->db->orderby("name", "desc");
$query = $this->db->get();
```

This should give us the first five (ordered by name) websites belonging to client number 3, and fetch the client's surname as well as his or her ID number!

A hidden benefit of using Active Record is that data that may have come in from users is automatically escaped, so you don't have to worry about putting quotes around it. This applies to functions like `$this->db->where()`, and also to the data creation and update statements described in the next sections. (Security warning: this is not the same thing as preventing cross-scripting attacks—for that you need CI's `xss_clean()` function. It's also not the same as validating your data—for this you need CI's validation class. See Chapter 5.)

Displaying Query Results

Showing database query results in CI is quite simple. We define our query as above, ending in:

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

Then, if there are multiple results, they can be returned as a `$row` object through which you iterate with a `foreach` loop:

```
foreach ($query->result() as $row)
{
   print $row->url;
   print $row->name;
   print $row->client;
}
```

or if we only want a single result, it can be returned as an object, or here as a `$row` array:

```
if ($query->num_rows() > 0)
{
   $row = $query->row_array();

   print $row['url'];
   print $row['name'];
   print $row['client'];
}
```

Personally, I prefer the object syntax to the array—less typing!

When you follow the MVC pattern, you will usually want to keep your queries and database interactions in models, and display the information through views.

Create and Update Queries

Active Record has three functions that help you create new entries in your database. They are `$this->db->insert()`, `$this->db->update()`, and `$this->db->set()`.

The difference between a 'create' and an 'update' query is that when you create a new record, there is no reference to any existing record, you are writing a new one. When you update, there is an existing record, and you are changing it. So in the second case, you have to specify which record you are changing. In both cases, you have to set the values you want to leave in the database after your query. Values you don't set will be left unaltered—or, if they didn't exist before, they will still be 'null' after your query.

CI allows you to set your values either with an array, or with `$this->db-set()`; the difference is only one of syntax.

So, let's add a line to our `sites` table in the `websites` database. We've already connected to this database in our controller. The controller's constructor function included the line:

```
$this->load->database();
```

We want to add a new site, which has a URL, a name, a type, and a client ID number. As an array, this might be:

```
$data = array(
                'url' => 'www.mynewclient.com',
                'name' => 'BigCo Inc',
                'clientid' => '33',
                'type' => 'dynamic'
            );
```

To add that to the `sites` table, we follow it with:

```
$this->db->insert('sites', $data);
```

Alternatively, we could set each value using `$this->db->set()`:

```
$this->db->set('url', 'www.mynewclinet.com');
$this->db->set('name', 'BigCo Inc');
$this->db->set('clientid', '33');
$this->db->set('type', 'dynamic');
$this->db->insert('sites');
```

If we are updating an existing record, then again we can either create an array, or use `$this->db->set()`. But there are two differences.

Firstly, we have to specify the record we want to update; and second, we need to use `$this->db->update()`. If I want to update a record (say the record with its 'id' field set to 1) in my `sites` table, using the data set out in my `$data` array above, the syntax is:

```
$this->db->where('id', '1');
$this->db->update('sites', $data);
```

Or I can set out the information using `$this->db->set()`, as above.

CI gives you several functions to check what the database has done. Most usefully:

```
$this->db->affected_rows();
```

should return '1' after my insert or update statement—but might show more rows if I was altering several rows of data at one time. You can use it to check that the operation has done what you expected.

You've noticed that I didn't set an ID field when I created a new record. That's because we set the database to populate the ID field automatically when a new record is added. But I do have to specify an ID when I update an existing record, otherwise the database doesn't know which one to alter.

If I'm generating a new record, however, I don't know the ID number until I've generated it. If I then need to refer to the new record, I can get the new ID number with:

```
$new_id_number = $this->db->insert_id();
```

(This code has to go, as soon as possible, after the operation that generated the record, or it may give a misleading result.)

For a little more peace of mind, remember that CI Active Record functions, including `$this->db->insert()` and `$this->db->update()` automatically escape data passed to them as input.

From version 1.5, CI also includes support for transactions—linking two or more database actions together so that they either all succeed, or all fail. This is essential in double-entry book-keeping applications and many commercial sites. For instance, say you are selling theatre tickets. You record receiving a payment in one transaction, and then allocate a seat to the customer in another. If your system fails after doing the first database operation, but before doing the second, you may end up with an angry customer—who has been charged, but has not had a seat reserved.

CI now makes it much simpler to link two or more database operations into one transaction, so that if they all succeed, the transaction is 'committed', and if one or more fails, the transaction is 'rolled back'. We don't need to use this in our example site, but if you want more information see the CI online User Guide.

Delete Queries

Delete queries are perhaps the simplest to describe. All you need is the name of the table and the ID number of the record to delete. Let's say I want to delete record in my `sites` table with the ID number 2:

```
$this->db->where('id', '2');
$this->db->delete('sites');
```

I get slightly nervous around 'delete' queries because they are so powerful. Please remember to make sure that there is a valid value in the 'where' clause, or you may delete your whole table! Neither the author nor Packt Publishing will accept any liability if….

Mixing Active Record and 'Classic' Styles

CI doesn't insist that you use Active Record. You can also use CI to issue straight SQL queries. For instance, assuming you loaded the database in your constructor, you can still write queries like this:

```
$this->db->query("SELECT id, name, url FROM sites WHERE 'type' =
'dynamic'");
```

Personally, I find Active Record easier to use. Conceptually, setting out my query in an array makes it easier to see and manipulate as an entity than writing it in SQL syntax. It's slightly more verbose, but clearly structured; it automatically escapes data; and it may be more portable. It also minimizes typing errors with commas and quotes.

There are a few cases, however, where you may have to resort to the original SQL. You might want to do complex joins, or another example is if you need to use multiple 'where' conditions. If you want to find the websites associated with client 3, but only those of two specific types, you may need to put brackets around the SQL to make sure the query is correctly interpreted.

In cases like these, you can always write out the SQL as a string, put it in a variable, and use the variable in CI's `$this->db->where()` function, as follows:

```
$condition = "client ='3' AND (type ='dynamic' OR type='static')";
$this->db->where($condition);.
```

Without the brackets this is ambiguous. Do you mean:

```
(client='3' AND type = 'dynamic') OR type = 'static'
```

or

```
client='3' AND (type = 'dynamic' OR type = 'static')
```

Well, yes of course, it's obvious, but the machine usually guesses wrong. Incidentally, be careful with the syntax of `$condition`. The actual SQL query is:

```
client='3' AND (type = 'dynamic' OR type = 'static')
```

The double quotes come from the variable assignment:

```
$condition = "   ";
```

It's easy to get your single and double quotes confused.

Some of the CI expressions I've quoted above, like `$this->db->affected_rows()`, are not a part of its Active Record model. But they can be mixed in easily.

The only time you might run into problems is if you try to mix Active Record and straight SQL in the same query. (I haven't tried this. If you have a lot of time on your hands, you could test it out, but frankly, I think that would indicate a sad lifestyle. Try train-spotting instead. At least it gets you out into the fresh air! I use CI because I'm too busy not to!)

Summary

We've looked at CI's Active Record class and seen how easy it is to:

- Set up connections to one or more databases
- Do standard SQL read, update, create, and delete queries
- Perform other functions that we need, to use a database properly

CI's Active Record function is clean and easy to use, and makes coding much clearer to read. It automates database connections, allowing you to abstract the connection information to one config file.

It can do pretty well anything that you can do with 'classic' SQL—more than I have space to explain here. See the online User Guide for fuller details.

Chapter Appendix: MYSQL Query to Set Up 'websites' Database

```
DROP TABLE IF EXISTS `ci_sessions`;
CREATE TABLE IF NOT EXISTS `ci_sessions` (
  `session_id` varchar(40) NOT NULL default '0',
  `peopleid` int(11) NOT NULL,
  `ip_address` varchar(16) NOT NULL default '0',
  `user_agent` varchar(50) NOT NULL,
  `last_activity` int(10) unsigned NOT NULL default '0',
  `left` int(11) NOT NULL,
  `name` varchar(25) NOT NULL,
  `status` tinyint(4) NOT NULL default '0'
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `domains`;
CREATE TABLE IF NOT EXISTS `domains` (
  `id` int(10) NOT NULL auto_increment,
  `url` varchar(100) NOT NULL,
  `name` varchar(100) NOT NULL,
  `registrar` varchar(100) NOT NULL,
  `dateregd` int(11) NOT NULL default '0',
  `cost` float NOT NULL default '0',
  `regdfor` int(11) NOT NULL default '0',
  `notes` blob NOT NULL,
  `pw` varchar(25) NOT NULL,
  `un` varchar(25) NOT NULL,
  `lastupdate` int(11) NOT NULL default '0',
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=10 ;

DROP TABLE IF EXISTS `events`;
CREATE TABLE IF NOT EXISTS `events` (
  `id` int(10) NOT NULL auto_increment,
  `name` varchar(50) NOT NULL default 'not set',
  `type` enum('test','alert','report') NOT NULL,
  `testid` int(10) NOT NULL,
  `siteid` int(10) NOT NULL,
  `userid` int(10) NOT NULL,
  `reported` int(11) NOT NULL,
  `result` blob NOT NULL,
  `time` int(11) NOT NULL,
```

```
  `timetaken` float NOT NULL,
  `isalert` varchar(2) NOT NULL,
  `emailid` int(11) NOT NULL,
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=69 ;

DROP TABLE IF EXISTS `frequencies`;
CREATE TABLE IF NOT EXISTS `frequencies` (
  `id` int(10) NOT NULL,
  `name` varchar(16) NOT NULL,
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `hosts`;
CREATE TABLE IF NOT EXISTS `hosts` (
  `id` int(11) NOT NULL auto_increment,
  `cost` float NOT NULL,
  `name` varchar(100) NOT NULL,
  `hosturl` varchar(100) NOT NULL,
  `un` varchar(50) NOT NULL,
  `pw` varchar(50) NOT NULL,
  `ns1url` varchar(36) NOT NULL,
  `ns1ip` varchar(36) NOT NULL,
  `ns2url` varchar(36) NOT NULL,
  `ns2ip` varchar(36) NOT NULL,
  `ftpurl` varchar(100) NOT NULL,
  `ftpserverip` varchar(36) NOT NULL,
  `ftpun` varchar(50) NOT NULL,
  `ftppw` varchar(50) NOT NULL,
  `cpurl` varchar(36) NOT NULL,
  `cpun` varchar(36) NOT NULL,
  `cppw` varchar(36) NOT NULL,
  `pop3server` varchar(36) NOT NULL,
  `servicetel` varchar(50) NOT NULL,
  `servicetel2` varchar(50) NOT NULL,
  `serviceemail` varchar(100) NOT NULL,
  `webroot` varchar(48) NOT NULL,
  `absoluteroot` varchar(48) NOT NULL,
  `cgiroot` varchar(48) NOT NULL,
  `booked` int(11) NOT NULL,
  `duration` int(11) NOT NULL,
  `lastupdate` int(11) NOT NULL default '0',
```

```
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=6 ;

DROP TABLE IF EXISTS `people`;
CREATE TABLE IF NOT EXISTS `people` (
  `id` int(11) NOT NULL auto_increment,
  `uname` varchar(25) NOT NULL,
  `pw` varchar(25) NOT NULL,
  `status` smallint(3) NOT NULL default '1',
  `name` varchar(50) NOT NULL,
  `firstname` varchar(50) NOT NULL,
  `surname` varchar(50) NOT NULL,
  `email` varchar(120) NOT NULL,
  `lastupdate` int(11) NOT NULL default '0',
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=5 ;

DROP TABLE IF EXISTS `sites`;
CREATE TABLE IF NOT EXISTS `sites` (
  `id` int(10) NOT NULL auto_increment,
  `name` varchar(100) NOT NULL,
  `url` varchar(100) NOT NULL,
  `un` varchar(50) NOT NULL,
  `pw` varchar(50) NOT NULL,
  `client1` int(10) NOT NULL default '0',
  `client2` int(10) NOT NULL default '0',
  `admin1` int(10) NOT NULL default '0',
  `admin2` int(10) NOT NULL default '0',
  `domainid` int(10) NOT NULL default '0',
  `hostid` int(10) NOT NULL default '0',
  `webroot` varchar(50) NOT NULL,
  `files` text NOT NULL,
  `filesdate` int(11) NOT NULL default '0',
  `lastupdate` int(11) NOT NULL default '0',
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=15 ;

DROP TABLE IF EXISTS `tests`;
CREATE TABLE IF NOT EXISTS `tests` (
  `id` int(11) NOT NULL auto_increment,
  `siteid` int(11) NOT NULL default '0',
```

```
  `name` varchar(250) NOT NULL,
  `type` varchar(25) NOT NULL,
  `url` varchar(120) NOT NULL,
  `regex` varchar(250) NOT NULL,
  `p1` varchar(250) NOT NULL,
  `p2` varchar(250) NOT NULL,
  `p3` varchar(250) NOT NULL,
  `p4` varchar(250) NOT NULL,
  `p5` varchar(250) NOT NULL,
  `p6` varchar(250) NOT NULL,
  `frequency` int(10) NOT NULL default '0',
  `lastdone` int(10) NOT NULL default '0',
  `isalert` varchar(2) NOT NULL,
  `setup` int(10) NOT NULL default '0',
  `lastupdate` int(10) NOT NULL default '0',
  `notes` varchar(250) NOT NULL,
  `submit` varchar(25) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=11 ;

DROP TABLE IF EXISTS `types`;
CREATE TABLE IF NOT EXISTS `types` (
  `id` varchar(7) NOT NULL,
  `name` varchar(50) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Where to buy this book

You can buy CodeIgniter for Rapid PHP Application Development from the Packt Publishing website: http://www.packtpub.com/codeIgniter-php-application-development-mvc/book

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com