CodeIgniter User Guide Version 1.7.2

Table of Contents

CodeIgniter URLs

By default, URLs in CodeIgniter are designed to be search-engine and human friendly. Rather than using the standard "query string" approach to URLs that is synonymous with dynamic systems, CodeIgniter uses a **segment-based** approach:

```
example.com/news/article/my_article
```

Note: Query string URLs can be optionally enabled, as described below.

URI Segments

The segments in the URL, in following with the Model-View-Controller approach, usually represent:

```
example.com/class/function/ID
```

1. The first segment represents the controller **class** that should be invoked.
2. The second segment represents the class **function**, or method, that should be called.
3. The third, and any additional segments, represent the ID and any variables that will be passed to the controller.

The URI Class and the URL Helper contain functions that make it easy to work with your URI data. In addition, your URLs can be remapped using the URI Routing feature for more flexibility.

Removing the index.php file

By default, the **index.php** file will be included in your URLs:

```
example.com/index.php/news/article/my_article
```

You can easily remove this file by using a .htaccess file with some simple rules. Here is an example of such a file, using the "negative" method in which everything is redirected except the specified items:

```
RewriteEngine on
RewriteCond $1 !^(index\.php|images|robots\.txt)
RewriteRule ^(.*)$ /index.php/$1 [L]
```

In the above example, any HTTP request other than those for index.php, images, and robots.txt is treated as a request for your index.php file.

Adding a URL Suffix

In your **config/config.php** file you can specify a suffix that will be added to all URLs generated by

CodeIgniter. For example, if a URL is this:

```
example.com/index.php/products/view/shoes
```

You can optionally add a suffix, like **.html**, making the page appear to be of a certain type:

```
example.com/index.php/products/view/shoes.html
```

Enabling Query Strings

In some cases you might prefer to use query strings URLs:

```
index.php?c=products&m=view&id=345
```

CodeIgniter optionally supports this capability, which can be enabled in your **application/config.php** file. If you open your config file you'll see these items:

```
$config['enable_query_strings'] = FALSE;
$config['controller_trigger'] = 'c';
$config['function_trigger'] = 'm';
```

If you change "enable_query_strings" to TRUE this feature will become active. Your controllers and functions will then be accessible using the "trigger" words you've set to invoke your controllers and methods:

```
index.php?c=controller&m=method
```

Please note: If you are using query strings you will have to build your own URLs, rather than utilizing the URL helpers (and other helpers that generate URLs, like some of the form helpers) as these are designed to work with segment based URLs.

Controllers

Controllers are the heart of your application, as they determine how HTTP requests should be handled.

What is a Controller?

A Controller is simply a class file that is named in a way that can be associated with a URI.

Consider this URI:

```
example.com/index.php/blog/
```

In the above example, CodeIgniter would attempt to find a controller named **blog.php** and load it.

When a controller's name matches the first segment of a URI, it will be loaded.

Let's try it: Hello World!

Let's create a simple controller so you can see it in action. Using your text editor, create a file called **blog.php**, and put the following code in it:

```
<?php
class Blog extends Controller {

	function index()
	{
		echo 'Hello World!';
	}
}
?>
```

Then save the file to your **application/controllers/** folder.

Now visit the your site using a URL similar to this:

```
example.com/index.php/blog/
```

If you did it right, you should see **Hello World!**.

Note: Class names must start with an uppercase letter. In other words, this is valid:

```
<?php
class Blog extends Controller {

}
?>
```

This is **not** valid:

```
<?php
class blog extends Controller {

}
?>
```

Also, always make sure your controller **extends** the parent controller class so that it can inherit all its functions.

Functions

In the above example the function name is **index()**. The "index" function is always loaded by default if the **second segment** of the URI is empty. Another way to show your "Hello World" message would be this:

```
example.com/index.php/blog/index/
```

The second segment of the URI determines which function in the controller gets called.

Let's try it. Add a new function to your controller:

```
<?php
class Blog extends Controller {

	function index()
	{
		echo 'Hello World!';
	}

	function comments()
	{
		echo 'Look at this!';
	}
}
?>
```

Now load the following URL to see the **comment** function:

```
example.com/index.php/blog/comments/
```

You should see your new message.

Passing URI Segments to your Functions

If your URI contains more then two segments they will be passed to your function as parameters.

For example, lets say you have a URI like this:

```
example.com/index.php/products/shoes/sandals/123
```

Your function will be passed URI segments 3 and 4 ("sandals" and "123"):

```
<?php
class Products extends Controller {

    function shoes($sandals, $id)
    {
        echo $sandals;
        echo $id;
    }
}
?>
```

Important: If you are using the URI Routing feature, the segments passed to your function will be the re-routed ones.

Defining a Default Controller

CodeIgniter can be told to load a default controller when a URI is not present, as will be the case when only your site root URL is requested. To specify a default controller, open your **application/config/routes.php** file and set this variable:

```
$route['default_controller'] = 'Blog';
```

Where **Blog** is the name of the controller class you want used. If you now load your main index.php file without specifying any URI segments you'll see your Hello World message by default.

Remapping Function Calls

As noted above, the second segment of the URI typically determines which function in the controller gets called. CodeIgniter permits you to override this behavior through the use of the **_remap()** function:

```
function _remap()
{
    // Some code here...
}
```

Important: If your controller contains a function named **_remap()**, it will **always** get called regardless of what your URI contains. It overrides the normal behavior in which the URI determines which function is called, allowing you to define your own function routing rules.

The overridden function call (typically the second segment of the URI) will be passed as a parameter the **_remap()** function:

```
function _remap($method)
{
   if ($method == 'some_method')
   {
      $this->$method();
   }
   else
   {
      $this->default_method();
   }
}
```

Processing Output

CodeIgniter has an output class that takes care of sending your final rendered data to the web browser automatically. More information on this can be found in the Views and Output class pages. In some cases, however, you might want to post-process the finalized data in some way and send it to the browser yourself. CodeIgniter permits you to add a function named **_output()** to your controller that will receive the finalized output data.

Important: If your controller contains a function named **_output()**, it will **always** be called by the output class instead of echoing the finalized data directly. The first parameter of the function will contain the finalized output.

Here is an example:

```
function _output($output)
{
   echo $output;
}
```

Please note that your **_output()** function will receive the data in its finalized state. Benchmark and memory usage data will be rendered, cache files written (if you have caching enabled), and headers will be sent (if you use that feature) before it is handed off to the _output() function. If you are using this feature the page execution timer and memory usage stats might not be perfectly accurate since they will not take into acccount any further processing you do. For an alternate way to control output *before* any of the final processing is done, please see the available methods in the Output Class.

Private Functions

In some cases you may want certain functions hidden from public access. To make a function private, simply add an underscore as the name prefix and it will not be served via a URL request. For example, if you were to have a function like this:

```
function _utility()
{
  // some code
}
```

Trying to access it via the URL, like this, will not work:

```
example.com/index.php/blog/_utility/
```

Organizing Your Controllers into Sub-folders

If you are building a large application you might find it convenient to organize your controllers into sub-folders. CodeIgniter permits you to do this.

Simply create folders within your **application/controllers** directory and place your controller classes within them.

Note: When using this feature the first segment of your URI must specify the folder. For example, lets say you have a controller located here:

```
application/controllers/products/shoes.php
```

To call the above controller your URI will look something like this:

```
example.com/index.php/products/shoes/show/123
```

Each of your sub-folders may contain a default controller which will be called if the URL contains only the sub-folder. Simply name your default controller as specified in your **application/config/routes.php** file

CodeIgniter also permits you to remap your URIs using its URI Routing feature.

Class Constructors

If you intend to use a constructor in any of your Controllers, you **MUST** place the following line of code in it:

```
parent::Controller();
```

The reason this line is necessary is because your local constructor will be overriding the one in the parent controller class so we need to manually call it.

If you are not familiar with constructors, in PHP 4, a *constructor* is simply a function that has the exact same name as the class:

```
<?php
class Blog extends Controller {

     function Blog()
     {
        parent::Controller();
    }
}
?>
```

In PHP 5, constructors use the following syntax:

```
<?php
class Blog extends Controller {

    function __construct()
    {
        parent::Controller();
    }
}
?>
```

Constructors are useful if you need to set some default values, or run a default process when your class is instantiated. Constructors can't return a value, but they can do some default work.

Reserved Function Names

Since your controller classes will extend the main application controller you must be careful not to name your functions identically to the ones used by that class, otherwise your local functions will override them. See Reserved Names for a full list.

That's it!

That, in a nutshell, is all there is to know about controllers.

Reserved Names

In order to help out, CodeIgniter uses a series of functions and names in its operation. Because of this, some names cannot be used by a developer. Following is a list of reserved names that cannot be used.

Controller names

Since your controller classes will extend the main application controller you must be careful not to name your functions identically to the ones used by that class, otherwise your local functions will override them. The following is a list of reserved names. Do not name your controller functions any of these:

- Controller
- CI_Base
- _ci_initialize
- _ci_scaffolding
- index

If you are running PHP 4 there are some additional reserved names. These ONLY apply if you are running PHP 4.

- CI_Loader
- config
- database
- dbutil
- dbforge
- file
- helper
- helpers
- language
- library
- model
- plugin
- plugins
- scaffolding
- script
- view
- vars
- _ci_assign_to_models
- _ci_autoloader
- _ci_init_class

- _ci_init_scaffolding
- _ci_is_instance
- _ci_load
- _ci_load_class
- _ci_object_to_array

Functions

- is_really_writable()
- load_class()
- get_config()
- config_item()
- show_error()
- show_404()
- log_message()
- _exception_handler()
- get_instance()

Variables

- $config
- $mimes
- $lang

Constants

- EXT
- FCPATH
- SELF
- BASEPATH
- APPPATH
- CI_VERSION
- FILE_READ_MODE
- FILE_WRITE_MODE
- DIR_READ_MODE
- DIR_WRITE_MODE
- FOPEN_READ
- FOPEN_READ_WRITE
- FOPEN_WRITE_CREATE_DESTRUCTIVE
- FOPEN_READ_WRITE_CREATE_DESTRUCTIVE
- FOPEN_WRITE_CREATE
- FOPEN_READ_WRITE_CREATE

- FOPEN_WRITE_CREATE_STRICT
- FOPEN_READ_WRITE_CREATE_STRICT

Views

A **view** is simply a web page, or a page fragment, like a header, footer, sidebar, etc. In fact, views can flexibly be embedded within other views (within other views, etc., etc.) if you need this type of hierarchy.

Views are never called directly, they must be loaded by a controller. Remember that in an MVC framework, the Controller acts as the traffic cop, so it is responsible for fetching a particular view. If you have not read the Controllers page you should do so before continuing.

Using the example controller you created in the controller page, let's add a view to it.

Creating a View

Using your text editor, create a file called **blogview.php**, and put this in it:

```
<html>
<head>
<title>My Blog</title>
</head>
<body>
	<h1>Welcome to my Blog!</h1>
</body>
</html>
```

Then save the file in your **application/views/** folder.

Loading a View

To load a particular view file you will use the following function:

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

Where **name** is the name of your view file. Note: The .php file extension does not need to be specified unless you use something other than **.php**.

Now, open the controller file you made earlier called **blog.php**, and replace the echo statement with the view loading function:

```
<?php
class Blog extends Controller {

	function index()
	{
		$this->load->view('blogview');
	}
}
?>
```

If you visit the your site using the URL you did earlier you should see your new view. The URL was similar to this:

```
example.com/index.php/blog/
```

Loading multiple views

CodeIgniter will intelligently handle multiple calls to $this->load->view from within a controller. If more then one call happens they will be appended together. For example, you may wish to have a header view, a menu view, a content view, and a footer view. That might look something like this:

```
<?php

class Page extends Controller {

  function index()
  {
    $data['page_title'] = 'Your title';
    $this->load->view('header');
    $this->load->view('menu');
    $this->load->view('content', $data);
    $this->load->view('footer');
  }

}
?>
```

In the example above, we are using "dynamically added data", which you will see below.

Storing Views within Sub-folders

Your view files can also be stored within sub-folders if you prefer that type of organization. When doing so you will need to include the folder name loading the view. Example:

```
$this->load->view('folder_name/file_name');
```

Adding Dynamic Data to the View

Data is passed from the controller to the view by way of an **array** or an **object** in the second parameter of the view loading function. Here is an example using an array:

```
$data = array(
          'title' => 'My Title',
          'heading' => 'My Heading',
          'message' => 'My Message'
       );

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

And here's an example using an object:

```
$data = new Someclass();
$this->load->view('blogview', $data);
```

Note: If you use an object, the class variables will be turned into array elements.

Let's try it with your controller file. Open it add this code:

```
<?php
class Blog extends Controller {

	function index()
	{
		$data['title'] = "My Real Title";
		$data['heading'] = "My Real Heading";

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

Now open your view file and change the text to variables that correspond to the array keys in your data:

```
<html>
<head>
<title><?php echo $title;?></title>
</head>
<body>
	<h1><?php echo $heading;?></h1>
</body>
</html>
```

Then load the page at the URL you've been using and you should see the variables replaced.

Creating Loops

The data array you pass to your view files is not limited to simple variables. You can pass multi dimensional arrays, which can be looped to generate multiple rows. For example, if you pull data from your database it will typically be in the form of a multi-dimensional array.

Here's a simple example. Add this to your controller:

```
<?php
class Blog extends Controller {

     function index()
     {
          $data['todo_list'] = array('Clean House', 'Call Mom', 'Run Errands');

          $data['title'] = "My Real Title";
          $data['heading'] = "My Real Heading";

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

Now open your view file and create a loop:

```
<html>
<head>
<title><?php echo $title;?></title>
</head>
<body>
<h1><?php echo $heading;?></h1>

<h3>My Todo List</h3>

<ul>
<?php foreach($todo_list as $item):?>

<li><?php echo $item;?></li>

<?php endforeach;?>
</ul>

</body>
</html>
```

Note: You'll notice that in the example above we are using PHP's alternative syntax. If you are not familiar with it you can read about it here.

Returning views as data

There is a third **optional** parameter lets you change the behavior of the function so that it returns data as a string rather than sending it to your browser. This can be useful if you want to process the data in some way. If you set the parameter to **true** (boolean) it will return data. The default behavior is **false**, which sends it to your browser. Remember to assign it to a variable if you want the data returned:

```
$string = $this->load->view('myfile', '', true);
```

Models

Models are **optionally** available for those who want to use a more traditional MVC approach.

What is a Model?

Models are PHP classes that are designed to work with information in your database. For example, let's say you use CodeIgniter to manage a blog. You might have a model class that contains functions to insert, update, and retrieve your blog data. Here is an example of what such a model class might look like:

```
class Blogmodel extends Model {

    var $title   = '';
    var $content = '';
    var $date    = '';

    function Blogmodel()
    {
        // Call the Model constructor
        parent::Model();
    }

    function get_last_ten_entries()
    {
        $query = $this->db->get('entries', 10);
        return $query->result();
    }

    function insert_entry()
    {
        $this->title   = $_POST['title']; // please read the below note
        $this->content = $_POST['content'];
        $this->date    = time();

        $this->db->insert('entries', $this);
    }

    function update_entry()
    {
        $this->title   = $_POST['title'];
        $this->content = $_POST['content'];
        $this->date    = time();

        $this->db->update('entries', $this, array('id' => $_POST['id']));
    }

}
```

Note: The functions in the above example use the Active Record database functions.

Note: For the sake of simplicity in this example we're using $_POST directly. This is generally bad practice, and a more common approach would be to use the Input Class $this->input->post('title')

Anatomy of a Model

Model classes are stored in your **application/models/** folder. They can be nested within sub-folders if you want this type of organization.

The basic prototype for a model class is this:

```
class Model_name extends Model {

    function Model_name()
    {
        parent::Model();
    }
}
```

Where **Model_name** is the name of your class. Class names **must** have the first letter capitalized with the rest of the name lowercase. Make sure your class extends the base Model class.

The file name will be a lower case version of your class name. For example, if your class is this:

```
class User_model extends Model {

    function User_model()
    {
        parent::Model();
    }
}
```

Your file will be this:

```
application/models/user_model.php
```

Loading a Model

Your models will typically be loaded and called from within your controller functions. To load a model you will use the following function:

```
$this->load->model('Model_name');
```

If your model is located in a sub-folder, include the relative path from your models folder. For example, if you have a model located at **application/models/blog/queries.php** you'll load it using:

```
$this->load->model('blog/queries');
```

Once loaded, you will access your model functions using an object with the same name as your class:

```
$this->load->model('Model_name');

$this->Model_name->function();
```

If you would like your model assigned to a different object name you can specify it via the second parameter of the loading function:

```
$this->load->model('Model_name', 'fubar');

$this->fubar->function();
```

Here is an example of a controller, that loads a model, then serves a view:

```
class Blog_controller extends Controller {

   function blog()
   {
      $this->load->model('Blog');

      $data['query'] = $this->Blog->get_last_ten_entries();

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

Auto-loading Models

If you find that you need a particular model globally throughout your application, you can tell CodeIgniter to auto-load it during system initialization. This is done by opening the application/config/autoload.php file and adding the model to the autoload array.

Connecting to your Database

When a model is loaded it does **NOT** connect automatically to your database. The following options for connecting are available to you:

- You can connect using the standard database methods described here, either from within your Controller class or your Model class.
- You can tell the model loading function to auto-connect by passing **TRUE** (boolean) via the third parameter, and connectivity settings, as defined in your database config file will be used:

  ```
  $this->load->model('Model_name', '', TRUE);
  ```

- You can manually pass database connectivity settings via the third parameter:

  ```
  $config['hostname'] = "localhost";
  $config['username'] = "myusername";
  ```

```
$config['password'] = "mypassword";
$config['database'] = "mydatabase";
$config['dbdriver'] = "mysql";
$config['dbprefix'] = "";
$config['pconnect'] = FALSE;
$config['db_debug'] = TRUE;

$this->load->model('Model_name', '', $config);
```

Helper Functions

Helpers, as the name suggests, help you with tasks. Each helper file is simply a collection of functions in a particular category. There are **URL Helpers**, that assist in creating links, there are **Form Helpers** that help you create form elements, **Text Helpers** perform various text formatting routines, **Cookie Helpers** set and read cookies, **File Helpers** help you deal with files, etc.

Unlike most other systems in CodeIgniter, Helpers are not written in an Object Oriented format. They are simple, procedural functions. Each helper function performs one specific task, with no dependence on other functions.

CodeIgniter does not load Helper Files by default, so the first step in using a Helper is to load it. Once loaded, it becomes globally available in your controller and views.

Helpers are typically stored in your **system/helpers**, or **system/application/helpers** directory. CodeIgniter will look first in your **system/application/helpers** directory. If the directory does not exist or the specified helper is not located there CI will instead look in your global **system/helpers** folder.

Loading a Helper

Loading a helper file is quite simple using the following function:

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

Where **name** is the file name of the helper, without the .php file extension or the "helper" part.

For example, to load the **URL Helper** file, which is named **url_helper.php**, you would do this:

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

A helper can be loaded anywhere within your controller functions (or even within your View files, although that's not a good practice), as long as you load it before you use it. You can load your helpers in your controller constructor so that they become available automatically in any function, or you can load a helper in a specific function that needs it.

Note: The Helper loading function above does not return a value, so don't try to assign it to a variable. Just use it as shown.

Loading Multiple Helpers

If you need to load more than one helper you can specify them in an array, like this:

```
$this->load->helper( array('helper1', 'helper2', 'helper3') );
```

Auto-loading Helpers

If you find that you need a particular helper globally throughout your application, you can tell CodeIgniter to auto-load it during system initialization. This is done by opening the **application/config/autoload.php** file and adding the helper to the autoload array.

Using a Helper

Once you've loaded the Helper File containing the function you intend to use, you'll call it the way you would a standard PHP function.

For example, to create a link using the **anchor()** function in one of your view files you would do this:

```
<?php echo anchor('blog/comments', 'Click Here');?>
```

Where "Click Here" is the name of the link, and "blog/comments" is the URI to the controller/function you wish to link to.

"Extending" Helpers

To "extend" Helpers, create a file in your **application/helpers/** folder with an identical name to the existing Helper, but prefixed with **MY_** (this item is configurable. See below.).

If all you need to do is add some functionality to an existing helper - perhaps add a function or two, or change how a particular helper function operates - then it's overkill to replace the entire helper with your version. In this case it's better to simply "extend" the Helper. The term "extend" is used loosely since Helper functions are procedural and discrete and cannot be extended in the traditional programmatic sense. Under the hood, this gives you the ability to add to the functions a Helper provides, or to modify how the native Helper functions operate.

For example, to extend the native **Array Helper** you'll create a file named **application/helpers/MY_array_helper.php**, and add or override functions:

```
// any_in_array() is not in the Array Helper, so it defines a new function
function any_in_array($needle, $haystack)
{
   $needle = (is_array($needle)) ? $needle : array($needle);

   foreach ($needle as $item)
   {
      if (in_array($item, $haystack))
      {
         return TRUE;
      }
   }

   return FALSE;
}

// random_element() is included in Array Helper, so it overrides the native function
function random_element($array)
{
   shuffle($array);
   return array_pop($array);
}
```

Setting Your Own Prefix

The filename prefix for "extending" Helpers is the same used to extend libraries and Core classes. To set your own prefix, open your **application/config/config.php** file and look for this item:

```
$config['subclass_prefix'] = 'MY_';
```

Please note that all native CodeIgniter libraries are prefixed with **CI_** so DO NOT use that as your prefix.

Now What?

In the Table of Contents you'll find a list of all the available Helper Files. Browse each one to see what they do.

Plugins

Plugins work almost identically to Helpers. The main difference is that a plugin usually provides a single function, whereas a Helper is usually a collection of functions. Helpers are also considered a part of the core system; plugins are intended to be created and shared by our community.

Plugins should be saved to your **system/plugins** directory or you can create a folder called **plugins** inside your **application** folder and store them there. CodeIgniter will look first in your **system/application/plugins** directory. If the directory does not exist or the specified plugin is not located there CI will instead look in your global **system/plugins** folder.

Loading a Plugin

Loading a plugin file is quite simple using the following function:

```
$this->load->plugin('name');
```

Where **name** is the file name of the plugin, without the .php file extension or the "plugin" part.

For example, to load the **Captcha** plugin, which is named **captcha_pi.php**, you will do this:

```
$this->load->plugin('captcha');
```

A plugin can be loaded anywhere within your controller functions (or even within your View files, although that's not a good practice), as long as you load it before you use it. You can load your plugins in your controller constructor so that they become available automatically in any function, or you can load a plugin in a specific function that needs it.

Note: The Plugin loading function above does not return a value, so don't try to assign it to a variable. Just use it as shown.

Loading Multiple Plugins

If you need to load more than one plugin you can specify them in an array, like this:

```
$this->load->plugin( array('plugin1', 'plugin2', 'plugin3') );
```

Auto-loading Plugins

If you find that you need a particular plugin globally throughout your application, you can tell CodeIgniter to auto-load it during system initialization. This is done by opening the **application/config/autoload.php** file and adding the plugin to the autoload array.

Using a Plugin

Once you've loaded the Plugin, you'll call it the way you would a standard PHP function.

Using CodeIgniter Libraries

All of the available libraries are located in your **system/libraries** folder. In most cases, to use one of these classes involves initializing it within a controller using the following initialization function:

```
$this->load->library('class name');
```

Where **class name** is the name of the class you want to invoke. For example, to load the validation class you would do this:

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

Once initialized you can use it as indicated in the user guide page corresponding to that class.

Creating Your Own Libraries

Please read the section of the user guide that discusses how to create your own libraries

Creating Libraries

When we use the term "Libraries" we are normally referring to the classes that are located in the **libraries** directory and described in the Class Reference of this user guide. In this case, however, we will instead describe how you can create your own libraries within your **application/libraries** directory in order to maintain separation between your local resources and the global framework resources.

As an added bonus, CodeIgniter permits your libraries to **extend** native classes if you simply need to add some functionality to an existing library. Or you can even replace native libraries just by placing identically named versions in your **application/libraries** folder.

In summary:

- You can create entirely new libraries.
- You can extend native libraries.
- You can replace native libraries.

The page below explains these three concepts in detail.

Note: The Database classes can not be extended or replaced with your own classes, nor can the Loader class in PHP 4. All other classes are able to be replaced/extended.

Storage

Your library classes should be placed within your **application/libraries** folder, as this is where CodeIgniter will look for them when they are initialized.

Naming Conventions

- File names must be capitalized. For example: **Myclass.php**
- Class declarations must be capitalized. For example: **class Myclass**
- Class names and file names must match.

The Class File

Classes should have this basic prototype (Note: We are using the name **Someclass** purely as an example):

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

class Someclass {

    function some_function()
    {
    }
}
```

```
?>
```

Using Your Class

From within any of your Controller functions you can initialize your class using the standard:

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

Where *someclass* is the file name, without the ".php" file extension. You can submit the file name capitalized or lower case. CodeIgniter doesn't care.

Once loaded you can access your class using the **lower case** version:

```
$this->someclass->some_function();  // Object instances will always be lower case
```

Passing Parameters When Initializing Your Class

In the library loading function you can dynamically pass data as an array via the second parameter and it will be passed to your class constructor:

```
$params = array('type' => 'large', 'color' => 'red');

$this->load->library('Someclass', $params);
```

If you use this feature you must set up your class constructor to expect data:

```
<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

class Someclass {

    function Someclass($params)
    {
        // Do something with $params
    }
}

?>
```

You can also pass parameters stored in a config file. Simply create a config file named identically to the class **file name** and store it in your **application/config/** folder. Note that if you dynamically pass parameters as described above, the config file option will not be available.

Utilizing CodeIgniter Resources within Your Library

To access CodeIgniter's native resources within your library use the **get_instance()** function. This function returns the CodeIgniter super object.

Normally from within your controller functions you will call any of the available CodeIgniter functions using the **$this** construct:

```
$this->load->helper('url');
$this->load->library('session');
$this->config->item('base_url');
etc.
```

$this, however, only works directly within your controllers, your models, or your views. If you would like to use CodeIgniter's classes from within your own custom classes you can do so as follows:

First, assign the CodeIgniter object to a variable:

```
$CI =& get_instance();
```

Once you've assigned the object to a variable, you'll use that variable *instead* of **$this**:

```
$CI =& get_instance();

$CI->load->helper('url');
$CI->load->library('session');
$CI->config->item('base_url');
etc.
```

Note: You'll notice that the above get_instance() function is being passed by reference:

$CI =& get_instance();

This is very important. Assigning by reference allows you to use the original CodeIgniter object rather than creating a copy of it.

Also, please note: If you are running PHP 4 it's usually best to avoid calling **get_instance()** from within your class constructors. PHP 4 has trouble referencing the CI super object within application constructors since objects do not exist until the class is fully instantiated.

Replacing Native Libraries with Your Versions

Simply by naming your class files identically to a native library will cause CodeIgniter to use it instead of the native one. To use this feature you must name the file and the class declaration exactly the same as the native library. For example, to replace the native **Email** library you'll create a file named **application/libraries/Email.php**, and declare your class with:

```
class CI_Email {

}
```

Note that most native classes are prefixed with **CI_**.

To load your library you'll see the standard loading function:

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

Note: At this time the Database classes can not be replaced with your own versions.

Extending Native Libraries

If all you need to do is add some functionality to an existing library - perhaps add a function or two - then it's overkill to replace the entire library with your version. In this case it's better to simply extend the class. Extending a class is nearly identical to replacing a class with a couple exceptions:

- The class declaration must extend the parent class.
- Your new class name and filename must be prefixed with **MY_** (this item is configurable. See below.).

For example, to extend the native **Email** class you'll create a file named **application/libraries/MY_Email.php**, and declare your class with:

```
class MY_Email extends CI_Email {

}
```

Note: If you need to use a constructor in your class make sure you extend the parent constructor:

```
class MY_Email extends CI_Email {

    function My_Email()
    {
        parent::CI_Email();
    }
}
```

Loading Your Sub-class

To load your sub-class you'll use the standard syntax normally used. DO NOT include your prefix. For example, to load the example above, which extends the Email class, you will use:

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

Once loaded you will use the class variable as you normally would for the class you are extending. In the case of the email class all calls will use:

```
$this->email->some_function();
```

Setting Your Own Prefix

To set your own sub-class prefix, open your **application/config/config.php** file and look for this item:

```
$config['subclass_prefix'] = 'MY_';
```

Please note that all native CodeIgniter libraries are prefixed with **CI_** so DO NOT use that as your prefix.

Creating Core System Classes

Every time CodeIgniter runs there are several base classes that are initialized automatically as part of the core framework. It is possible, however, to swap any of the core system classes with your own versions or even extend the core versions.

Most users will never have any need to do this, but the option to replace or extend them does exist for those who would like to significantly alter the CodeIgniter core.

Note: Messing with a core system class has a lot of implications, so make sure you know what you are doing before attempting it.

System Class List

The following is a list of the core system files that are invoked every time CodeIgniter runs:

- Benchmark
- Config
- Controller
- Exceptions
- Hooks
- Input
- Language
- Loader
- Log
- Output
- Router
- URI

Replacing Core Classes

To use one of your own system classes instead of a default one simply place your version inside your local **application/libraries** directory:

```
application/libraries/some-class.php
```

If this directory does not exist you can create it.

Any file named identically to one from the list above will be used instead of the one normally used.

Please note that your class must use **CI** as a prefix. For example, if your file is named **Input.php** the class will be named:

```
class CI_Input {
}
```

Extending Core Class

If all you need to do is add some functionality to an existing library - perhaps add a function or two - then it's overkill to replace the entire library with your version. In this case it's better to simply extend the class. Extending a class is nearly identical to replacing a class with a couple exceptions:

- The class declaration must extend the parent class.
- Your new class name and filename must be prefixed with **MY_** (this item is configurable. See below.).

For example, to extend the native **Input** class you'll create a file named **application/libraries/MY_Input.php**, and declare your class with:

```
class MY_Input extends CI_Input {

}
```

Note: If you need to use a constructor in your class make sure you extend the parent constructor:

```
class MY_Input extends CI_Input {

    function MY_Input()
    {
        parent::CI_Input();
    }
}
```

Tip: Any functions in your class that are named identically to the functions in the parent class will be used instead of the native ones (this is known as "method overriding"). This allows you to substantially alter the CodeIgniter core.

If you are extending the Controller core class, then be sure to extend your new class in your application controller's constructors.

```
class Welcome extends MY_Controller {

    function Welcome()
    {
        parent::MY_Controller();
    }

    function index()
    {
        $this->load->view('welcome_message');
    }
}
```

Setting Your Own Prefix

To set your own sub-class prefix, open your **application/config/config.php** file and look for this item:

```
$config['subclass_prefix'] = 'MY_';
```

Please note that all native CodeIgniter libraries are prefixed with **CI_** so DO NOT use that as your prefix.

Hooks - Extending the Framework Core

CodeIgniter's Hooks feature provides a means to tap into and modify the inner workings of the framework without hacking the core files. When CodeIgniter runs it follows a specific execution process, diagramed in the Application Flow page. There may be instances, however, where you'd like to cause some action to take place at a particular stage in the execution process. For example, you might want to run a script right before your controllers get loaded, or right after, or you might want to trigger one of your own scripts in some other location.

Enabling Hooks

The hooks feature can be globally enabled/disabled by setting the following item in the **application/config/config.php** file:

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

Defining a Hook

Hooks are defined in **application/config/hooks.php** file. Each hook is specified as an array with this prototype:

```
$hook['pre_controller'] = array(
                                'class'    => 'MyClass',
                                'function' => 'Myfunction',
                                'filename' => 'Myclass.php',
                                'filepath' => 'hooks',
                                'params'   => array('beer', 'wine', 'snacks')
                                );
```

Notes:
The array index correlates to the name of the particular hook point you want to use. In the above example the hook point is **pre_controller**. A list of hook points is found below. The following items should be defined in your associative hook array:

- **class** The name of the class you wish to invoke. If you prefer to use a procedural function instead of a class, leave this item blank.
- **function** The function name you wish to call.
- **filename** The file name containing your class/function.
- **filepath** The name of the directory containing your script. Note: Your script must be located in a directory INSIDE your **application** folder, so the file path is relative to that folder. For example, if your script is located in **application/hooks**, you will simply use **hooks** as your filepath. If your script is located in **application/hooks/utilities** you will use **hooks/utilities** as your filepath. No trailing slash.
- **params** Any parameters you wish to pass to your script. This item is optional.

Multiple Calls to the Same Hook

If want to use the same hook point with more then one script, simply make your array declaration multi-dimensional, like this:

```
$hook['pre_controller'][] = array(
                                'class'    => 'MyClass',
                                'function' => 'Myfunction',
                                'filename' => 'Myclass.php',
                                'filepath' => 'hooks',
                                'params'   => array('beer', 'wine', 'snacks')
                                );

$hook['pre_controller'][] = array(
                                'class'    => 'MyOtherClass',
                                'function' => 'MyOtherfunction',
                                'filename' => 'Myotherclass.php',
                                'filepath' => 'hooks',
                                'params'   => array('red', 'yellow', 'blue')
                                );
```

Notice the brackets after each array index:

```
$hook['pre_controller'][]
```

This permits you to have the same hook point with multiple scripts. The order you define your array will be the execution order.

Hook Points

The following is a list of available hook points.

- **pre_system**
  Called very early during system execution. Only the benchmark and hooks class have been loaded at this point. No routing or other processes have happened.
- **pre_controller**
  Called immediately prior to any of your controllers being called. All base classes, routing, and security checks have been done.
- **post_controller_constructor**
  Called immediately after your controller is instantiated, but prior to any method calls happening.
- **post_controller**
  Called immediately after your controller is fully executed.
- **display_override**
  Overrides the **_display()** function, used to send the finalized page to the web browser at the end of system execution. This permits you to use your own display methodology. Note that you will need to reference the CI superobject with **$this->CI =& get_instance()** and then the finalized data will be available by calling **$this->CI->output->get_output()**
- **cache_override**
  Enables you to call your own function instead of the **_display_cache()** function in the output class. This permits you to use your own cache display mechanism.
- **scaffolding_override**
  Permits a scaffolding request to trigger your own script instead.
- **post_system**
  Called after the final rendered page is sent to the browser, at the end of system execution after

the finalized data is sent to the browser.

Auto-loading Resources

CodeIgniter comes with an "Auto-load" feature that permits libraries, helpers, and plugins to be initialized automatically every time the system runs. If you need certain resources globally throughout your application you should consider auto-loading them for convenience.

The following items can be loaded automatically:

- Core classes found in the "libraries" folder
- Helper files found in the "helpers" folder
- Plugins found in the "plugins" folder
- Custom config files found in the "config" folder
- Language files found in the "system/language" folder
- Models found in the "models" folder

To autoload resources, open the **application/config/autoload.php** file and add the item you want loaded to the **autoload** array. You'll find instructions in that file corresponding to each type of item.

Note: Do not include the file extension (.php) when adding items to the autoload array.

Common Functions

CodeIgniter uses a few functions for its operation that are globally defined, and are available to you at any point. These do not require loading any libraries or helpers.

is_php('**version_number**')

is_php() determines of the PHP version being used is greater than the supplied **version_number**.

```
if (is_php('5.3.0'))
{
    $str = quoted_printable_encode($str);
}
```

Returns boolean **TRUE** if the installed version of PHP is equal to or greater than the supplied version number. Returns **FALSE** if the installed version of PHP is lower than the supplied version number.

is_really_writable('**path/to/file**')

is_writable() returns TRUE on Windows servers when you really can't write to the file as the OS reports to PHP as FALSE only if the read-only attribute is marked. This function determines if a file is actually writable by attempting to write to it first. Generally only recommended on platforms where this information may be unreliable.

```
if (is_really_writable('file.txt'))
{
    echo "I could write to this if I wanted to";
}
else
{
    echo "File is not writable";
}
```

config_item('**item_key**')

The Config library is the preferred way of accessing configuration information, however config_item() can be used to retrieve single keys. See Config library documentation for more information.

show_error('**message**'), show_404('**page**'), log_message('**level**', '**message**')

These are each outlined on the Error Handling page.

set_status_header(code, 'text');

Permits you to manually set a server status header. Example:

```
set_status_header(401);
// Sets the header as: Unauthorized
```

See here for a full list of headers.

Scaffolding

Scaffolding has been deprecated from CodeIgniter as of 1.6.0.

CodeIgniter's Scaffolding feature provides a fast and very convenient way to add, edit, or delete information in your database during development.

Very Important: Scaffolding is intended for development use only. It provides very little security other than a "secret" word, so anyone who has access to your CodeIgniter site can potentially edit or delete your information. If you use scaffolding make sure you disable it immediately after you are through using it. DO NOT leave it enabled on a live site. And please, set a secret word before you use it.

Why would someone use scaffolding?

Here's a typical scenario: You create a new database table during development and you'd like a quick way to insert some data into it to work with. Without scaffolding your choices are either to write some inserts using the command line or to use a database management tool like phpMyAdmin. With CodeIgniter's scaffolding feature you can quickly add some data using its browser interface. And when you are through using the data you can easily delete it.

Setting a Secret Word

Before enabling scaffolding please take a moment to set a secret word. This word, when encountered in your URL, will launch the scaffolding interface, so please pick something obscure that no one is likely to guess.

To set a secret word, open your **application/config/routes.php** file and look for this item:

```
$route['scaffolding_trigger'] = '';
```

Once you've found it add your own unique word.

Note: The scaffolding word can **not** start with an underscore.

Enabling Scaffolding

Note: The information on this page assumes you already know how controllers work, and that you have a working one available. It also assumes you have configured CodeIgniter to auto-connect to your database. If not, the information here won't be very relevant, so you are encouraged to go through those sections first. Lastly, it assumes you understand what a class constructor is. If not, read the last section of the controllers page.

To enable scaffolding you will initialize it in your constructor like this:

```
<?php
```

```
class Blog extends Controller {

     function Blog()
     {
        parent::Controller();

        $this->load->scaffolding('table_name');
     }
}
?>
```

Where **table_name** is the name of the table (table, not database) you wish to work with.

Once you've initialized scaffolding, you will access it with this URL prototype:

```
example.com/index.php/class/secret_word/
```

For example, using a controller named **Blog**, and **abracadabra** as the secret word, you would access scaffolding like this:

```
example.com/index.php/blog/abracadabra/
```

The scaffolding interface should be self-explanatory. You can add, edit or delete records.

A Final Note:

The scaffolding feature will only work with tables that contain a primary key, as this is information is needed to perform the various database functions.

URI Routing

Typically there is a one-to-one relationship between a URL string and its corresponding controller class/method. The segments in a URI normally follow this pattern:

```
example.com/class/function/id/
```

In some instances, however, you may want to remap this relationship so that a different class/function can be called instead of the one corresponding to the URL.

For example, lets say you want your URLs to have this prototype:

example.com/product/1/
example.com/product/2/
example.com/product/3/
example.com/product/4/

Normally the second segment of the URL is reserved for the function name, but in the example above it instead has a product ID. To overcome this, CodeIgniter allows you to remap the URI handler.

Setting your own routing rules

Routing rules are defined in your **application/config/routes.php** file. In it you'll see an array called **$route** that permits you to specify your own routing criteria. Routes can either be specified using **wildcards** or **Regular Expressions**

Wildcards

A typical wildcard route might look something like this:

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

In a route, the array key contains the URI to be matched, while the array value contains the destination it should be re-routed to. In the above example, if the literal word "product" is found in the first segment of the URL, and a number is found in the second segment, the "catalog" class and the "product_lookup" method are instead used.

You can match literal values or you can use two wildcard types:

:num
:any

:num will match a segment containing only numbers.
:any will match a segment containing any character.

Note: Routes will run in the order they are defined. Higher routes will always take precedence over lower ones.

Examples

Here are a few routing examples:

```
$route['journals'] = "blogs";
```

A URL containing the word "journals" in the first segment will be remapped to the "blogs" class.

```
$route['blog/joe'] = "blogs/users/34";
```

A URL containing the segments blog/joe will be remapped to the "blogs" class and the "users" method. The ID will be set to "34".

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

A URL with "product" as the first segment, and anything in the second will be remapped to the "catalog" class and the "product_lookup" method.

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

A URL with "product" as the first segment, and anything in the second will be remapped to the "catalog" class and the "product_lookup_by_id" method passing in the match as a variable to the function.

Important: Do not use leading/trailing slashes.

Regular Expressions

If you prefer you can use regular expressions to define your routing rules. Any valid regular expression is allowed, as are back-references.

Note: If you use back-references you must use the dollar syntax rather than the double backslash syntax.

A typical RegEx route might look something like this:

```
$route['products/([a-z]+)/(\d+)'] = "$1/id_$2";
```

In the above example, a URI similar to **products/shirts/123** would instead call the **shirts** controller class and the **id_123** function.

You can also mix and match wildcards with regular expressions.

Reserved Routes

There are two reserved routes:

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

This route indicates which controller class should be loaded if the URI contains no data, which will be the case when people load your root URL. In the above example, the "welcome" class would be loaded. You are encouraged to always have a default route otherwise a 404 page will appear by default.

```
$route['scaffolding_trigger'] = 'scaffolding';
```

This route lets you set a secret word, which when present in the URL, triggers the scaffolding feature. Please read the Scaffolding page for details.

Important: The reserved routes must come before any wildcard or regular expression routes.

Error Handling

CodeIgniter lets you build error reporting into your applications using the functions described below. In addition, it has an error logging class that permits error and debugging messages to be saved as text files.

Note: By default, CodeIgniter displays all PHP errors. You might wish to change this behavior once your development is complete. You'll find the **error_reporting()** function located at the top of your main index.php file. Disabling error reporting will NOT prevent log files from being written if there are errors.

Unlike most systems in CodeIgniter, the error functions are simple procedural interfaces that are available globally throughout the application. This approach permits error messages to get triggered without having to worry about class/function scoping.

The following functions let you generate errors:

show_error('message' [, int $status_code= 500])

This function will display the error message supplied to it using the following error template:

application/errors/error_general.php

The optional parameter $status_code determines what HTTP status code should be sent with the error.

show_404('page')

This function will display the 404 error message supplied to it using the following error template:

application/errors/error_404.php

The function expects the string passed to it to be the file path to the page that isn't found. Note that CodeIgniter automatically shows 404 messages if controllers are not found.

log_message('level', 'message')

This function lets you write messages to your log files. You must supply one of three "levels" in the first parameter, indicating what type of message it is (debug, error, info), with the message itself in the second parameter. Example:

```
if ($some_var == "")
{
   log_message('error', 'Some variable did not contain a value.');
}
else
{
   log_message('debug', 'Some variable was correctly set');
}
```

```
log_message('info', 'The purpose of some variable is to provide some value.');
```

There are three message types:

1. Error Messages. These are actual errors, such as PHP errors or user errors.
2. Debug Messages. These are messages that assist in debugging. For example, if a class has been initialized, you could log this as debugging info.
3. Informational Messages. These are the lowest priority messages, simply giving information regarding some process. CodeIgniter doesn't natively generate any info messages but you may want to in your application.

Note: In order for the log file to actually be written, the "logs" folder must be writable. In addition, you must set the "threshold" for logging. You might, for example, only want error messages to be logged, and not the other two types. If you set it to zero logging will be disabled.

Web Page Caching

CodeIgniter lets you cache your pages in order to achieve maximum performance.

Although CodeIgniter is quite fast, the amount of dynamic information you display in your pages will correlate directly to the server resources, memory, and processing cycles utilized, which affect your page load speeds. By caching your pages, since they are saved in their fully rendered state, you can achieve performance that nears that of static web pages.

How Does Caching Work?

Caching can be enabled on a per-page basis, and you can set the length of time that a page should remain cached before being refreshed. When a page is loaded for the first time, the cache file will be written to your **system/cache** folder. On subsequent page loads the cache file will be retrieved and sent to the requesting user's browser. If it has expired, it will be deleted and refreshed before being sent to the browser.

Note: The Benchmark tag is not cached so you can still view your page load speed when caching is enabled.

Enabling Caching

To enable caching, put the following tag in any of your controller functions:

```
$this->output->cache(n);
```

Where **n** is the number of **minutes** you wish the page to remain cached between refreshes.

The above tag can go anywhere within a function. It is not affected by the order that it appears, so place it wherever it seems most logical to you. Once the tag is in place, your pages will begin being cached.

Warning: Because of the way CodeIgniter stores content for output, caching will only work if you are generating display for your controller with a view.

Note: Before the cache files can be written you must set the file permissions on your **system/cache** folder such that it is writable.

Deleting Caches

If you no longer wish to cache a file you can remove the caching tag and it will no longer be refreshed when it expires. Note: Removing the tag will not delete the cache immediately. It will have to expire normally. If you need to remove it earlier you will need to manually delete it from your cache folder.

Profiling Your Application

The Profiler Class will display benchmark results, queries you have run, and $_POST data at the bottom of your pages. This information can be useful during development in order to help with debugging and optimization.

Initializing the Class

Important: This class does **NOT** need to be initialized. It is loaded automatically by the Output Class if profiling is enabled as shown below.

Enabling the Profiler

To enable the profiler place the following function anywhere within your Controller functions:

```
$this->output->enable_profiler(TRUE);
```

When enabled a report will be generated and inserted at the bottom of your pages.

To disable the profiler you will use:

```
$this->output->enable_profiler(FALSE);
```

Setting Benchmark Points

In order for the Profiler to compile and display your benchmark data you must name your mark points using specific syntax.

Please read the information on setting Benchmark points in Benchmark Class page.

Managing your Applications

By default it is assumed that you only intend to use CodeIgniter to manage one application, which you will build in your **system/application/** directory. It is possible, however, to have multiple sets of applications that share a single CodeIgniter installation, or even to rename or relocate your **application** folder.

Renaming the Application Folder

If you would like to rename your **application** folder you may do so as long as you open your main **index.php** file and set its name using the **$application_folder** variable:

```
$application_folder = "application";
```

Relocating your Application Folder

It is possible to move your **application** folder to a different location on your server than your **system** folder. To do so open your main **index.php** and set a *full server path* in the **$application_folder** variable.

```
$application_folder = "/Path/to/your/application";
```

Running Multiple Applications with one CodeIgniter Installation

If you would like to share a common CodeIgniter installation to manage several different applications simply put all of the directories located inside your **application** folder into their own sub-folder.

For example, let's say you want to create two applications, "foo" and "bar". You will structure your application folder like this:

```
system/application/foo/
system/application/foo/config/
system/application/foo/controllers/
system/application/foo/errors/
system/application/foo/libraries/
system/application/foo/models/
system/application/foo/views/
system/application/bar/
system/application/bar/config/
system/application/bar/controllers/
system/application/bar/errors/
system/application/bar/libraries/
system/application/bar/models/
system/application/bar/views/
```

To select a particular application for use requires that you open your main **index.php** file and set the **$application_folder** variable. For example, to select the "foo" application for use you would do this:

```
$application_folder = "application/foo";
```

Note: Each of your applications will need its own **index.php** file which calls the desired application. The index.php file can be named anything you want.

Alternate PHP Syntax for View Files

If you do not utilize CodeIgniter's template engine, you'll be using pure PHP in your View files. To minimize the PHP code in these files, and to make it easier to identify the code blocks it is recommended that you use PHPs alternative syntax for control structures and short tag echo statements. If you are not familiar with this syntax, it allows you to eliminate the braces from your code, and eliminate "echo" statements.

Automatic Short Tag Support

Note: If you find that the syntax described in this page does not work on your server it might be that "short tags" are disabled in your PHP ini file. CodeIgniter will optionally rewrite short tags on-the-fly, allowing you to use that syntax even if your server doesn't support it. This feature can be enabled in your **config/config.php** file.

Please note that if you do use this feature, if PHP errors are encountered in your **view files**, the error message and line number will not be accurately shown. Instead, all errors will be shown as **eval()** errors.

Alternative Echos

Normally to echo, or print out a variable you would do this:

```
<?php echo $variable; ?>
```

With the alternative syntax you can instead do it this way:

```
<?=$variable?>
```

Alternative Control Structures

Controls structures, like **if**, **for**, **foreach**, and **while** can be written in a simplified format as well. Here is an example using foreach:

```
<ul>

<?php foreach($todo as $item): ?>

<li><?=$item?></li>

<?php endforeach; ?>

</ul>
```

Notice that there are no braces. Instead, the end brace is replaced with **endforeach**. Each of the control structures listed above has a similar closing syntax: **endif**, **endfor**, **endforeach**, and

endwhile

Also notice that instead of using a semicolon after each structure (except the last one), there is a colon. This is important!

Here is another example, using if/elseif/else. Notice the colons:

```
<?php if ($username == 'sally'): ?>

  <h3>Hi Sally</h3>

<?php elseif ($username == 'joe'): ?>

  <h3>Hi Joe</h3>

<?php else: ?>

  <h3>Hi unknown user</h3>

<?php endif; ?>
```

Security

This page describes some "best practices" regarding web security, and details CodeIgniter's internal security features.

URI Security

CodeIgniter is fairly restrictive regarding which characters it allows in your URI strings in order to help minimize the possibility that malicious data can be passed to your application. URIs may only contain the following:

- Alpha-numeric text
- Tilde: ~
- Period: .
- Colon: :
- Underscore: _
- Dash: -

GET, POST, and COOKIE Data

GET data is simply disallowed by CodeIgniter since the system utilizes URI segments rather than traditional URL query strings (unless you have the query string option enabled in your config file). The global GET array is **unset** by the Input class during system initialization.

Register_globals

During system initialization all global variables are unset, except those found in the $_POST and $_COOKIE arrays. The unsetting routine is effectively the same as register_globals = off.

magic_quotes_runtime

The magic_quotes_runtime directive is turned off during system initialization so that you don't have to remove slashes when retrieving data from your database.

Best Practices

Before accepting any data into your application, whether it be POST data from a form submission, COOKIE data, URI data, XML-RPC data, or even data from the SERVER array, you are encouraged to practice this three step approach:

1. Filter the data as if it were tainted.
2. Validate the data to ensure it conforms to the correct type, length, size, etc. (sometimes this step can replace step one)

3. Escape the data before submitting it into your database.

CodeIgniter provides the following functions to assist in this process:

XSS Filtering

CodeIgniter comes with a Cross Site Scripting filter. This filter looks for commonly used techniques to embed malicious Javascript into your data, or other types of code that attempt to hijack cookies or do other malicious things. The XSS Filter is described here.

Validate the data

CodeIgniter has a Form Validation Class that assists you in validating, filtering, and prepping your data.

Escape all data before database insertion

Never insert information into your database without escaping it. Please see the section that discusses queries for more information.

General Style and Syntax

The following page describes the coding rules use adhere to when developing CodeIgniter.

Table of Contents

File Format

Files should be saved with Unicode (UTF-8) encoding. The BOM should *not* be used. Unlike UTF-16 and UTF-32, there's no byte order to indicate in a UTF-8 encoded file, and the BOM can have a negative side effect in PHP of sending output, preventing the application from being able to set its own headers. Unix line endings should be used (LF).

Here is how to apply these settings in some of the more common text editors. Instructions for your text editor may vary; check your text editor's documentation.

TextMate

1. Open the Application Preferences
2. Click Advanced, and then the "Saving" tab
3. In "File Encoding", select "UTF-8 (recommended)"
4. In "Line Endings", select "LF (recommended)"
5. *Optional:* Check "Use for existing files as well" if you wish to modify the line endings of files you open to your new preference.

BBEdit

1. Open the Application Preferences
2. Select "Text Encodings" on the left.
3. In "Default text encoding for new documents", select "Unicode (UTF-8, no BOM)"
4. *Optional:* In "If file's encoding can't be guessed, use", select "Unicode (UTF-8, no BOM)"
5. Select "Text Files" on the left.
6. In "Default line breaks", select "Mac OS X and Unix (LF)"

PHP Closing Tag

The PHP closing tag on a PHP document **?>** is optional to the PHP parser. However, if used, any whitespace following the closing tag, whether introduced by the developer, user, or an FTP application, can cause unwanted output, PHP errors, or if the latter are suppressed, blank pages. For this reason, all PHP files should **OMIT** the closing PHP tag, and instead use a comment block to mark the end of file and it's location relative to the application root. This allows you to still identify a file as being complete and not truncated.

```
INCORRECT:
<?php

echo "Here's my code!";

?>

CORRECT:
<?php

echo "Here's my code!";

/* End of file myfile.php */
/* Location: ./system/modules/mymodule/myfile.php */
```

Class and Method Naming

Class names should always have their first letter uppercase, and the constructor method should match identically. Multiple words should be separated with an underscore, and not CamelCased. All other class methods should be entirely lowercased and named to clearly indicate their function, preferably including a verb. Try to avoid overly long and verbose names.

```
INCORRECT:
class superclass
class SuperClass

CORRECT:
class Super_class
```

Notice that the Class and constructor methods are identically named and cased:

```
class Super_class {

	function Super_class()
	{

	}
}
```

Examples of improper and proper method naming:

```
INCORRECT:
function fileproperties()		// not descriptive and needs underscore separator
function fileProperties()		// not descriptive and uses CamelCase
function getfileproperties()		// Better!  But still missing underscore separator
function getFileProperties()		// uses CamelCase
function get_the_file_properties_from_the_file()	// wordy

CORRECT:
function get_file_properties()	// descriptive, underscore separator, and all lowercase letters
```

Variable Names

The guidelines for variable naming is very similar to that used for class methods. Namely, variables should contain only lowercase letters, use underscore separators, and be reasonably named to indicate their purpose and contents. Very short, non-word variables should only be used as iterators in for() loops.

```
INCORRECT:
$j = 'foo';		// single letter variables should only be used in for() loops
$Str			// contains uppercase letters
$bufferedText		// uses CamelCasing, and could be shortened without losing semantic meaning
$groupid		// multiple words, needs underscore separator
$name_of_last_city_used	// too long

CORRECT:
for ($j = 0; $j < 10; $j++)
$str
```

```
$buffer
$group_id
$last_city
```

Commenting

In general, code should be commented prolifically. It not only helps describe the flow and intent of the code for less experienced programmers, but can prove invaluable when returning to your own code months down the line. There is not a required format for comments, but the following are recommended.

DocBlock style comments preceding class and method declarations so they can be picked up by IDEs:

```
/**
 * Super Class
 *
 * @package     Package Name
 * @subpackage       Subpackage
 * @category    Category
 * @author      Author Name
 * @link    http://example.com
 */
class Super_class {
```

```
/**
 * Encodes string for use in XML
 *
 * @access      public
 * @param       string
 * @return      string
 */
function xml_encode($str)
```

Use single line comments within code, leaving a blank line between large comment blocks and code.

```
// break up the string by newlines
$parts = explode("\n", $str);

// A longer comment that needs to give greater detail on what is
// occurring and why can use multiple single-line comments.  Try to
// keep the width reasonable, around 70 characters is the easiest to
// read.  Don't hesitate to link to permanent external resources
// that may provide greater detail:
//
// http://example.com/information_about_something/in_particular/

$parts = $this->foo($parts);
```

Constants

Constants follow the same guidelines as do variables, except constants should always be fully uppercase. *Always use CodeIgniter constants when appropriate, i.e. SLASH, LD, RD, PATH_CACHE, etc.*

```
INCORRECT:
myConstant      // missing underscore separator and not fully uppercase
N           // no single-letter constants
S_C_VER         // not descriptive
$str = str_replace('{foo}', 'bar', $str);      // should use LD and RD constants

CORRECT:
MY_CONSTANT
NEWLINE
SUPER_CLASS_VERSION
$str = str_replace(LD.'foo'.RD, 'bar', $str);
```

TRUE, FALSE, and NULL

TRUE, **FALSE**, and **NULL** keywords should always be fully uppercase.

```
INCORRECT:
if ($foo == true)
$bar = false;
function foo($bar = null)

CORRECT:
if ($foo == TRUE)
$bar = FALSE;
function foo($bar = NULL)
```

Logical Operators

Use of **||** is discouraged as its clarity on some output devices is low (looking like the number 11 for instance). **&&** is preferred over **AND** but either are acceptable, and a space should always precede and follow **!**.

```
INCORRECT:
if ($foo || $bar)
if ($foo AND $bar)  // okay but not recommended for common syntax highlighting applications
if (!$foo)
if (! is_array($foo))

CORRECT:
if ($foo OR $bar)
if ($foo && $bar) // recommended
if ( ! $foo)
if ( ! is_array($foo))
```

Comparing Return Values and Typecasting

Some PHP functions return FALSE on failure, but may also have a valid return value of "" or 0, which would evaluate to FALSE in loose comparisons. Be explicit by comparing the variable type when using these return values in conditionals to ensure the return value is indeed what you expect, and not a value that has an equivalent loose-type evaluation.

Use the same stringency in returning and checking your own variables. Use **===** and **!==** as necessary.

```
INCORRECT:
// If 'foo' is at the beginning of the string, strpos will return a 0,
// resulting in this conditional evaluating as TRUE
if (strpos($str, 'foo') == FALSE)

CORRECT:
if (strpos($str, 'foo') === FALSE)
```

```
INCORRECT:
function build_string($str = "")
{
        if ($str == "")   // uh-oh!  What if FALSE or the integer 0 is passed as an argument?
        {

        }
}

CORRECT:
function build_string($str = "")
{
        if ($str === "")
        {

        }
}
```

See also information regarding typecasting, which can be quite useful. Typecasting has a slightly different effect which may be desirable. When casting a variable as a string, for instance, NULL and boolean FALSE variables become empty strings, 0 (and other numbers) become strings of digits, and boolean TRUE becomes "1":

```
$str = (string) $str;   // cast $str as a string
```

Debugging Code

No debugging code can be left in place for submitted add-ons unless it is commented out, i.e. no var_dump(), print_r(), die(), and exit() calls that were used while creating the add-on, unless they are commented out.

```
// print_r($foo);
```

Whitespace in Files

No whitespace can precede the opening PHP tag or follow the closing PHP tag. Output is buffered, so whitespace in your files can cause output to begin before CodeIgniter outputs its content, leading to errors and an inability for CodeIgniter to send proper headers. In the examples below, select the text with your mouse to reveal the incorrect whitespace.

INCORRECT:

```

<?php
	// ...there is whitespace and a linebreak above the opening PHP tag
	// as well as whitespace after the closing PHP tag
?>
```

CORRECT:

```
<?php
	// this sample has no whitespace before or after the opening and closing PHP tags
?>
```

Compatibility

Unless specifically mentioned in your add-on's documentation, all code must be compatible with PHP version 4.3+. Additionally, do not use PHP functions that require non-default libraries to be installed unless your code contains an alternative method when the function is not available, or you implicitly document that your add-on requires said PHP libraries.

Class and File Names using Common Words

When your class or filename is a common word, or might quite likely be identically named in another PHP script, provide a unique prefix to help prevent collision. Always realize that your end users may be running other add-ons or third party PHP scripts. Choose a prefix that is unique to your identity as a developer or company.

```
INCORRECT:
class Email		pi.email.php
class Xml		ext.xml.php
class Import		mod.import.php

CORRECT:
class Pre_email		pi.pre_email.php
class Pre_xml		ext.pre_xml.php
class Pre_import	mod.pre_import.php
```

Database Table Names

Any tables that your add-on might use must use the 'exp_' prefix, followed by a prefix uniquely identifying you as the developer or company, and then a short descriptive table name. You do not need to be concerned about the database prefix being used on the user's installation, as CodeIgniter's database class will automatically convert 'exp_' to what is actually being used.

```
INCORRECT:
email_addresses		// missing both prefixes
pre_email_addresses	// missing exp_ prefix
exp_email_addresses	// missing unique prefix

CORRECT:
exp_pre_email_addresses
```

NOTE: Be mindful that MySQL has a limit of 64 characters for table names. This should not be an issue as table names that would exceed this would likely have unreasonable names. For instance, the following table name exceeds this limitation by one character. Silly, no?
exp_pre_email_addresses_of_registered_users_in_seattle_washington

One File per Class

Use separate files for each class your add-on uses, unless the classes are *closely related*. An example of CodeIgniter files that contains multiple classes is the Database class file, which contains both the DB class and the DB_Cache class, and the Magpie plugin, which contains both the Magpie and Snoopy classes.

Whitespace

Use tabs for whitespace in your code, not spaces. This may seem like a small thing, but using tabs instead of whitespace allows the developer looking at your code to have indentation at levels that they prefer and customize in whatever application they use. And as a side benefit, it results in (slightly) more compact files, storing one tab character versus, say, four space characters.

Line Breaks

Files must be saved with Unix line breaks. This is more of an issue for developers who work in Windows, but in any case ensure that your text editor is setup to save files with Unix line breaks.

Code Indenting

Use Allman style indenting. With the exception of Class declarations, braces are always placed on a line by themselves, and indented at the same level as the control statement that "owns" them.

INCORRECT:
```
function foo($bar) {
	// ...
}

foreach ($arr as $key => $val) {
	// ...
}

if ($foo == $bar) {
	// ...
} else {
	// ...
}

for ($i = 0; $i < 10; $i++)
	{
	for ($j = 0; $j < 10; $j++)
		{
		// ...
		}
	}
```

CORRECT:
```
function foo($bar)
{
	// ...
}

foreach ($arr as $key => $val)
{
	// ...
}

if ($foo == $bar)
{
	// ...
}
else
{
	// ...
}

for ($i = 0; $i < 10; $i++)
{
	for ($j = 0; $j < 10; $j++)
	{
		// ...
	}
}
```

Bracket and Parenthetic Spacing

In general, parenthesis and brackets should not use any additional spaces. The exception is that a space should always follow PHP control structures that accept arguments with parenthesis (declare, do-while, elseif, for, foreach, if, switch, while), to help distinguish them from functions and increase readability.

```
INCORRECT:
$arr[ $foo ] = 'foo';

CORRECT:
$arr[$foo] = 'foo'; // no spaces around array keys


INCORRECT:
function foo ( $bar )
{

}

CORRECT:
function foo($bar) // no spaces around parenthesis in function declarations
{

}


INCORRECT:
foreach( $query->result() as $row )

CORRECT:
```

```
foreach ($query->result() as $row) // single space following PHP control structures, but not in interior parenthesis
```

Localized Text in Control Panel

Any text that is output in the control panel should use language variables in your module's lang file to allow localization.

```
INCORRECT:
return "Invalid Selection";

CORRECT:
return $LANG->line('invalid_selection');
```

Private Methods and Variables

Methods and variables that are only accessed internally by your class, such as utility and helper functions that your public methods use for code abstraction, should be prefixed with an underscore.

```
convert_text()        // public method
_convert_text()       // private method
```

PHP Errors

Code must run error free and not rely on warnings and notices to be hidden to meet this requirement. For instance, never access a variable that you did not set yourself (such as $_POST array keys) without first checking to see that it isset().

Make sure that while developing your add-on, error reporting is enabled for ALL users, and that display_errors is enabled in the PHP environment. You can check this setting with:

```
if (ini_get('display_errors') == 1)
{
	exit "Enabled";
}
```

On some servers where display_errors is disabled, and you do not have the ability to change this in the php.ini, you can often enable it with:

```
ini_set('display_errors', 1);
```

NOTE: Setting the display_errors setting with ini_set() at runtime is not identical to having it enabled in the PHP environment. Namely, it will not have any effect if the script has fatal errors

Short Open Tags

Always use full PHP opening tags, in case a server does not have short_open_tag enabled.

```
INCORRECT:
<? echo $foo; ?>

<?=$foo?>

CORRECT:
<?php echo $foo; ?>
```

One Statement Per Line

Never combine statements on one line.

```
INCORRECT:
$foo = 'this'; $bar = 'that'; $bat = str_replace($foo, $bar, $bag);

CORRECT:
$foo = 'this';
$bar = 'that';
$bat = str_replace($foo, $bar, $bag);
```

Strings

Always use single quoted strings unless you need variables parsed, and in cases where you do need variables parsed, use braces to prevent greedy token parsing. You may also use double-quoted strings if the string contains single quotes, so you do not have to use escape characters.

```
INCORRECT:
"My String"                              // no variable parsing, so no use for double quotes
"My string $foo"                         // needs braces
'SELECT foo FROM bar WHERE baz = \'bag\''     // ugly

CORRECT:
'My String'
"My string {$foo}"
"SELECT foo FROM bar WHERE baz = 'bag'"
```

SQL Queries

MySQL keywords are always capitalized: SELECT, INSERT, UPDATE, WHERE, AS, JOIN, ON, IN, etc.

Break up long queries into multiple lines for legibility, preferably breaking for each clause.

```
INCORRECT:
// keywords are lowercase and query is too long for
// a single line (... indicates continuation of line)
$query = $this->db->query("select foo, bar, baz, foofoo, foobar as raboof, foobaz from exp_pre_email_addresses
...where foo != 'oof' and baz != 'zab' order by foobaz limit 5, 100");

CORRECT:
```

```
$query = $this->db->query("SELECT foo, bar, baz, foofoo, foobar AS raboof, foobaz
                           FROM exp_pre_email_addresses
                           WHERE foo != 'oof'
                           AND baz != 'zab'
                           ORDER BY foobaz
                           LIMIT 5, 100");
```

Default Function Arguments

Whenever appropriate, provide function argument defaults, which helps prevent PHP errors with mistaken calls and provides common fallback values which can save a few lines of code. Example:

```
function foo($bar = '', $baz = FALSE)
```

Overlapping Tag Parameters

Avoid multiple tag parameters that have effect on the same thing. For instance, instead of **include=** and **exclude=**, perhaps allow **include=** to handle the parameter alone, with the addition of "not", e.g. **include="not bar"**. This will prevent problems of parameters overlapping or having to worry about which parameter has priority over another.

Benchmarking Class

CodeIgniter has a Benchmarking class that is always active, enabling the time difference between any two marked points to be calculated.

Note: This class is initialized automatically by the system so there is no need to do it manually.

In addition, the benchmark is always started the moment the framework is invoked, and ended by the output class right before sending the final view to the browser, enabling a very accurate timing of the entire system execution to be shown.

Table of Contents

Using the Benchmark Class

The Benchmark class can be used within your controllers, views, or your Models. The process for usage is this:

1. Mark a start point
2. Mark an end point
3. Run the "elapsed time" function to view the results

Here's an example using real code:

```
$this->benchmark->mark('code_start');

// Some code happens here

$this->benchmark->mark('code_end');

echo $this->benchmark->elapsed_time('code_start', 'code_end');
```

Note: The words "code_start" and "code_end" are arbitrary. They are simply words used to set two markers. You can use any words you want, and you can set multiple sets of markers. Consider this example:

```
$this->benchmark->mark('dog');

// Some code happens here

$this->benchmark->mark('cat');

// More code happens here
```

```
$this->benchmark->mark('bird');

echo $this->benchmark->elapsed_time('dog', 'cat');
echo $this->benchmark->elapsed_time('cat', 'bird');
echo $this->benchmark->elapsed_time('dog', 'bird');
```

Profiling Your Benchmark Points

If you want your benchmark data to be available to the Profiler all of your marked points must be set up in pairs, and each mark point name must end with **_start** and **_end**. Each pair of points must otherwise be named identically. Example:

```
$this->benchmark->mark('my_mark_start');

// Some code happens here...

$this->benchmark->mark('my_mark_end');

$this->benchmark->mark('another_mark_start');

// Some more code happens here...

$this->benchmark->mark('another_mark_end');
```

Please read the Profiler page for more information.

Displaying Total Execution Time

If you would like to display the total elapsed time from the moment CodeIgniter starts to the moment the final output is sent to the browser, simply place this in one of your view templates:

```
<?php echo $this->benchmark->elapsed_time();?>
```

You'll notice that it's the same function used in the examples above to calculate the time between two point, except you are **not** using any parameters. When the parameters are absent, CodeIgniter does not stop the benchmark until right before the final output is sent to the browser. It doesn't matter where you use the function call, the timer will continue to run until the very end.

An alternate way to show your elapsed time in your view files is to use this pseudo-variable, if you prefer not to use the pure PHP:

```
{elapsed_time}
```

Note: If you want to benchmark anything within your controller functions you must set your own start/end points.

Displaying Memory Consumption

If your PHP installation is configured with --enable-memory-limit, you can display the amount of memory consumed by the entire system using the following code in one of your view file:

```
<?php echo $this->benchmark->memory_usage();?>
```

Note: This function can only be used in your view files. The consumption will reflect the total memory used by the entire app.

An alternate way to show your memory usage in your view files is to use this pseudo-variable, if you prefer not to use the pure PHP:

```
{memory_usage}
```

Calendaring Class

The Calendar class enables you to dynamically create calendars. Your calendars can be formatted through the use of a calendar template, allowing 100% control over every aspect of its design. In addition, you can pass data to your calendar cells.

Initializing the Class

Like most other classes in CodeIgniter, the Calendar class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Calendar object will be available using: **$this->calendar**

Displaying a Calendar

Here is a very simple example showing how you can display a calendar:

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

echo $this->calendar->generate();
```

The above code will generate a calendar for the current month/year based on your server time. To show a calendar for a specific month and year you will pass this information to the calendar generating function:

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

echo $this->calendar->generate(2006, 6);
```

The above code will generate a calendar showing the month of June in 2006. The first parameter specifies the year, the second parameter specifies the month.

Passing Data to your Calendar Cells

To add data to your calendar cells involves creating an associative array in which the keys correspond to the days you wish to populate and the array value contains the data. The array is passed to the third parameter of the calendar generating function. Consider this example:

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

$data = array(
          3  => 'http://example.com/news/article/2006/03/',
          7  => 'http://example.com/news/article/2006/07/',
          13 => 'http://example.com/news/article/2006/13/',
          26 => 'http://example.com/news/article/2006/26/'
        );

echo $this->calendar->generate(2006, 6, $data);
```

Using the above example, day numbers 3, 7, 13, and 26 will become links pointing to the URLs you've provided.

Note: By default it is assumed that your array will contain links. In the section that explains the calendar template below you'll see how you can customize how data passed to your cells is handled so you can pass different types of information.

Setting Display Preferences

There are seven preferences you can set to control various aspects of the calendar. Preferences are set by passing an array of preferences in the second parameter of the loading function. Here is an example:

```
$prefs = array (
          'start_day'    => 'saturday',
          'month_type'   => 'long',
          'day_type'     => 'short'
        );

$this->load->library('calendar', $prefs);

echo $this->calendar->generate();
```

The above code would start the calendar on saturday, use the "long" month heading, and the "short" day names. More information regarding preferences below.

Preference	Default Value	Options	Description
template	None	None	A string containing your calendar template. See the template section below.
local_time	time()	None	A Unix timestamp corresponding to the current time.
start_day	sunday	Any week day (sunday, monday, tuesday, etc.)	Sets the day of the week the calendar should start on.
month_type	long	long, short	Determines what version of the month name to use in the header. long = January, short = Jan.
day_type	abr	long, short, abr	Determines what version of the weekday names to use in the column headers. long = Sunday, short = Sun, abr = Su.
show_next_prev	FALSE	TRUE/FALSE (boolean)	Determines whether to display links allowing you to toggle to next/previous months. See information on this feature below.
next_prev_url	None	A URL	Sets the basepath used in the next/previous calendar links.

Showing Next/Previous Month Links

To allow your calendar to dynamically increment/decrement via the next/previous links requires that you set up your calendar code similar to this example:

```
$prefs = array (
          'show_next_prev'  => TRUE,
          'next_prev_url'   => 'http://example.com/index.php/calendar/show/'
        );

$this->load->library('calendar', $prefs);

echo $this->calendar->generate($this->uri->segment(3), $this->uri->segment(4));
```

You'll notice a few things about the above example:

- You must set the "show_next_prev" to TRUE.
- You must supply the URL to the controller containing your calendar in the "next_prev_url" preference.
- You must supply the "year" and "month" to the calendar generating function via the URI segments where they appear (Note: The calendar class automatically adds the year/month to the base URL you provide.).

Creating a Calendar Template

By creating a calendar template you have 100% control over the design of your calendar. Each component of your calendar will be placed within a pair of pseudo-variables as shown here:

```
$prefs['template'] = '

  {table_open}<table border="0" cellpadding="0" cellspacing="0">{/table_open}

  {heading_row_start}<tr>{/heading_row_start}

  {heading_previous_cell}<th><a href="{previous_url}">&lt;&lt;</a>
</th>{/heading_previous_cell}
  {heading_title_cell}<th colspan="{colspan}">{heading}</th>{/heading_title_cell}
  {heading_next_cell}<th><a href="{next_url}">&gt;&gt;</a></th>{/heading_next_cell}

  {heading_row_end}</tr>{/heading_row_end}

  {week_row_start}<tr>{/week_row_start}
  {week_day_cell}<td>{week_day}</td>{/week_day_cell}
  {week_row_end}</tr>{/week_row_end}

  {cal_row_start}<tr>{/cal_row_start}
  {cal_cell_start}<td>{/cal_cell_start}

  {cal_cell_content}<a href="{content}">{day}</a>{/cal_cell_content}
  {cal_cell_content_today}<div class="highlight"><a href="{content}">{day}</a></div>
{/cal_cell_content_today}

  {cal_cell_no_content}{day}{/cal_cell_no_content}
  {cal_cell_no_content_today}<div class="highlight">{day}</div>{/cal_cell_no_content_today}

  {cal_cell_blank}&nbsp;{/cal_cell_blank}
```

```
   {cal_cell_end}</td>{/cal_cell_end}
   {cal_row_end}</tr>{/cal_row_end}

   {table_close}</table>{/table_close}
';

$this->load->library('calendar', $prefs);

echo $this->calendar->generate();
```

Shopping Cart Class

The Cart Class permits items to be added to a session that stays active while a user is browsing your site. These items can be retrieved and displayed in a standard "shopping cart" format, allowing the user to update the quantity or remove items from the cart.

Please note that the Cart Class ONLY provides the core "cart" functionality. It does not provide shipping, credit card authorization, or other processing components.

Initializing the Shopping Cart Class

Important: The Cart class utilizes CodeIgniter's Session Class to save the cart information to a database, so before using the Cart class you must set up a database table as indicated in the Session Documentation , and set the session preferences in your **appliction/config/config.php** file to utilize a database.

To initialize the Shopping Cart Class in your controller constructor, use the **$this->load->library** function:

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

Once loaded, the Cart object will be available using: **$this->cart**

Note: The Cart Class will load and initialize the Session Class automatically, so unless you are using sessions elsewhere in your application, you do not need to load the Session class.

Adding an Item to The Cart

To add an item to the shopping cart, simply pass an array with the product information to the **$this->cart->insert()** function, as shown below:

```
$data = array(
           'id'      => 'sku_123ABC',
           'qty'     => 1,
           'price'   => 39.95,
           'name'    => 'T-Shirt',
           'options' => array('Size' => 'L', 'Color' => 'Red')
        );

$this->cart->insert($data);
```

Important: The first four array indexes above (**id**, **qty**, **price**, and **name**) are **required**. If you omit any of them the data will not be saved to the cart. The fifth index (**options**) is optional. It is intended to be used in cases where your product has options associated with it. Use an array for options, as shown above.

The five reserved indexes are:

- **id** - Each product in your store must have a unique identifier. Typically this will be an "sku" or other such identifier.

- **qty** - The quantity being purchased.
- **price** - The price of the item.
- **name** - The name of the item.
- **options** - Any additional attributes that are needed to identify the product. These must be passed via an array.

In addition to the five indexes above, there are two reserved words: **rowid** and **subtotal**. These are used internally by the Cart class, so please do NOT use those words as index names when inserting data into the cart.

Your array may contain additional data. Anything you include in your array will be stored in the session. However, it is best to standardize your data among all your products in order to make displaying the information in a table easier.

Adding Multiple Items to The Cart

By using a multi-dimensional array, as shown below, it is possible to add multiple products to the cart in one action. This is useful in cases where you wish to allow people to select from among several items on the same page.

```
$data = array(
           array(
                'id'      => 'sku_123ABC',
                'qty'     => 1,
                'price'   => 39.95,
                'name'    => 'T-Shirt',
                'options' => array('Size' => 'L', 'Color' => 'Red')
              ),
           array(
                'id'      => 'sku_567ZYX',
                'qty'     => 1,
                'price'   => 9.95,
                'name'    => 'Coffee Mug'
              ),
           array(
                'id'      => 'sku_965QRS',
                'qty'     => 1,
                'price'   => 29.95,
                'name'    => 'Shot Glass'
              )
        );

$this->cart->insert($data);
```

Displaying the Cart

To display the cart you will create a view file with code similar to the one shown below.

Please note that this example uses the form helper.

```
<?php echo form_open('path/to/controller/update/function'); ?>

<table cellpadding="6" cellspacing="1" style="width:100%" border="0">

<tr>
  <th>QTY</th>
  <th>Item Description</th>
  <th style="text-align:right">Item Price</th>
  <th style="text-align:right">Sub-Total</th>
</tr>

<?php $i = 1; ?>

<?php foreach($this->cart->contents() as $items): ?>

	<?php echo form_hidden($i.'[rowid]', $items['rowid']); ?>

	<tr>
	  <td><?php echo form_input(array('name' => $i.'[qty]', 'value' => $items['qty'], 'maxlength'
=> '3', 'size' => '5')); ?></td>
	  <td>
		<?php echo $items['name']; ?>

			<?php if ($this->cart->has_options($items['rowid']) == TRUE): ?>

				<p>
					<?php foreach ($this->cart->product_options($items['rowid']) as
$option_name => $option_value): ?>

						<strong><?php echo $option_name; ?>:</strong> <?php echo
$option_value; ?><br />

					<?php endforeach; ?>
				</p>

			<?php endif; ?>

	  </td>
	  <td style="text-align:right"><?php echo $this->cart->format_number($items['price']);
?></td>
	  <td style="text-align:right">$<?php echo $this->cart->format_number($items['subtotal']);
?></td>
	</tr>

<?php $i++; ?>

<?php endforeach; ?>

<tr>
  <td colspan="2"> </td>
  <td class="right"><strong>Total</strong></td>
  <td class="right">$<?php echo $this->cart->format_number($this->cart->total()); ?></td>
</tr>

</table>
```

Updating The Cart

To update the information in your cart, you must pass an array containing the **Row ID** and quantity to the **$this->cart->update()** function:

Note: If the quantity is set to zero, the item will be removed from the cart.

```
$data = array(
          'rowid' => 'b99ccdf16028f015540f341130b6d8ec',
          'qty'   => 3
        );

$this->cart->update($data);

// Or a multi-dimensional array

$data = array(
          array(
               'rowid'   => 'b99ccdf16028f015540f341130b6d8ec',
               'qty'     => 3
             ),
          array(
               'rowid'   => 'xw82g9q3r495893iajdh473990rikw23',
               'qty'     => 4
             ),
          array(
               'rowid'   => 'fh4kdkkkaoe30njgoe92rkdkkobec333',
               'qty'     => 2
             )
        );

$this->cart->update($data);
```

What is a Row ID? The **row ID** is a unique identifier that is generated by the cart code when an item is added to the cart. The reason a unique ID is created is so that identical products with different options can be managed by the cart.

For example, let's say someone buys two identical t-shirts (same product ID), but in different sizes. The product ID (and other attributes) will be identical for both sizes because it's the same shirt. The only difference will be the size. The cart must therefore have a means of identifying this difference so that the two sizes of shirts can be managed independently. It does so by creating a unique "row ID" based on the product ID and any options associated with it.

In nearly all cases, updating the cart will be something the user does via the "view cart" page, so as a developer, it is unlikely that you will ever have to concern yourself with the "row ID", other then making sure your "view cart" page contains this information in a hidden form field, and making sure it gets passed to the update function when the update form is submitted. Please examine the construction of the "view cart" page above for more information.

Function Reference

$this->cart->insert();

Permits you to add items to the shopping cart, as outlined above.

$this->cart->update();

Permits you to update items in the shopping cart, as outlined above.

$this->cart->total();

Displays the total amount in the cart.

$this->cart->total_items();

Displays the total number of items in the cart.

$this->cart->contents();

Returns an array containing everything in the cart.

$this->cart->has_options(rowid);

Returns TRUE (boolean) if a particular row in the cart contains options. This function is designed to be used in a loop with **$this->cart->contents()**, since you must pass the **rowid** to this function, as shown in the **Displaying the Cart** example above.

$this->cart->options(rowid);

Returns an array of options for a particular product. This function is designed to be used in a loop with **$this->cart->contents()**, since you must pass the **rowid** to this function, as shown in the **Displaying the Cart** example above.

$this->cart->destroy();

Permits you to destroy the cart. This function will likely be called when you are finished processing the customer's order.

Config Class

The Config class provides a means to retrieve configuration preferences. These preferences can come from the default config file (**application/config/config.php**) or from your own custom config files.

> **Note:** This class is initialized automatically by the system so there is no need to do it manually.

Anatomy of a Config File

By default, CodeIgniter has a one primary config file, located at **application/config/config.php**. If you open the file using your text editor you'll see that config items are stored in an array called **$config**.

You can add your own config items to this file, or if you prefer to keep your configuration items separate (assuming you even need config items), simply create your own file and save it in **config** folder.

Note: If you do create your own config files use the same format as the primary one, storing your items in an array called **$config**. CodeIgniter will intelligently manage these files so there will be no conflict even though the array has the same name (assuming an array index is not named the same as another).

Loading a Config File

Note: CodeIgniter automatically loads the primary config file (**application/config/config.php**), so you will only need to load a config file if you have created your own.

There are two ways to load a config file:

1. **Manual Loading**

   To load one of your custom config files you will use the following function within the controller that needs it:

   ```
   $this->config->load('filename');
   ```

   Where **filename** is the name of your config file, without the .php file extension.

   If you need to load multiple config files normally they will be merged into one master config array. Name collisions can occur, however, if you have identically named array indexes in different config files. To avoid collisions you can set the second parameter to **TRUE** and each config file will be stored in an array index corresponding to the name of the config file. Example:

   ```
   // Stored in an array with this prototype: $this->config['blog_settings'] = $config
   $this->config->load('blog_settings', TRUE);
   ```

   Please see the section entitled **Fetching Config Items** below to learn how to retrieve config items set this way.

The third parameter allows you to suppress errors in the event that a config file does not exist:

```
$this->config->load('blog_settings', FALSE, TRUE);
```

2. **Auto-loading**

   If you find that you need a particular config file globally, you can have it loaded automatically by the system. To do this, open the **autoload.php** file, located at **application/config/autoload.php**, and add your config file as indicated in the file.

Fetching Config Items

To retrieve an item from your config file, use the following function:

```
$this->config->item('item name');
```

Where **item name** is the $config array index you want to retrieve. For example, to fetch your language choice you'll do this:

```
$lang = $this->config->item('language');
```

The function returns FALSE (boolean) if the item you are trying to fetch does not exist.

If you are using the second parameter of the **$this->config->load** function in order to assign your config items to a specific index you can retrieve it by specifying the index name in the second parameter of the **$this->config->item()** function. Example:

```
// Loads a config file named blog_settings.php and assigns it to an index named "blog_settings"
$this->config->load('blog_settings', TRUE);

// Retrieve a config item named site_name contained within the blog_settings array
$site_name = $this->config->item('site_name', 'blog_settings');

// An alternate way to specify the same item:
$blog_config = $this->config->item('blog_settings');
$site_name = $blog_config['site_name'];
```

Setting a Config Item

If you would like to dynamically set a config item or change an existing one, you can so using:

```
$this->config->set_item('item_name', 'item_value');
```

Where **item_name** is the $config array index you want to change, and **item_value** is its value.

Helper Functions

The config class has the following helper functions:

$this->config->site_url();

This function retrieves the URL to your site, along with the "index" value you've specified in the config file.

$this->config->system_url();

This function retrieves the URL to your **system folder**.

The Database Class

CodeIgniter comes with a full-featured and very fast abstracted database class that supports both traditional structures and Active Record patterns. The database functions offer clear, simple syntax.

- Quick Start: Usage Examples
- Database Configuration
- Connecting to a Database
- Running Queries
- Generating Query Results
- Query Helper Functions
- Active Record Class
- Transactions
- Table MetaData
- Field MetaData
- Custom Function Calls
- Query Caching
- Database manipulation with Database Forge
- Database Utilities Class

Database Quick Start: Example Code

The following page contains example code showing how the database class is used. For complete details please read the individual pages describing each function.

Initializing the Database Class

The following code loads and initializes the database class based on your configuration settings:

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

Once loaded the class is ready to be used as described below.

Note: If all your pages require database access you can connect automatically. See the connecting page for details.

Standard Query With Multiple Results (Object Version)

```
$query = $this->db->query('SELECT name, title, email FROM my_table');

foreach ($query->result() as $row)
{
   echo $row->title;
   echo $row->name;
   echo $row->email;
}

echo 'Total Results: ' . $query->num_rows();
```

The above **result()** function returns an array of **objects**. Example: $row->title

Standard Query With Multiple Results (Array Version)

```
$query = $this->db->query('SELECT name, title, email FROM my_table');

foreach ($query->result_array() as $row)
{
   echo $row['title'];
   echo $row['name'];
   echo $row['email'];
}
```

The above **result_array()** function returns an array of standard array indexes. Example: $row['title']

Testing for Results

If you run queries that might **not** produce a result, you are encouraged to test for a result first using the **num_rows()** function:

```
$query = $this->db->query("YOUR QUERY");

if ($query->num_rows() > 0)
{
  foreach ($query->result() as $row)
  {
    echo $row->title;
    echo $row->name;
    echo $row->body;
  }
}
```

Standard Query With Single Result

```
$query = $this->db->query('SELECT name FROM my_table LIMIT 1');

$row = $query->row();
echo $row->name;
```

The above **row()** function returns an **object**. Example: $row->name

Standard Query With Single Result (Array version)

```
$query = $this->db->query('SELECT name FROM my_table LIMIT 1');

$row = $query->row_array();
echo $row['name'];
```

The above **row_array()** function returns an **array**. Example: $row['name']

Standard Insert

```
$sql = "INSERT INTO mytable (title, name)
      VALUES (".$this->db->escape($title).", ".$this->db->escape($name).")";

$this->db->query($sql);

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

Active Record Query

The Active Record Pattern gives you a simplified means of retrieving data:

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

```
foreach ($query->result() as $row)
{
   echo $row->title;
}
```

The above **get()** function retrieves all the results from the supplied table. The Active Record class contains a full compliment of functions for working with data.

Active Record Insert

```
$data = array(
           'title' => $title,
           'name' => $name,
           'date' => $date
        );

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

// Produces: INSERT INTO mytable (title, name, date) VALUES ('{$title}', '{$name}', '{$date}')
```

Database Configuration

CodeIgniter has a config file that lets you store your database connection values (username, password, database name, etc.). The config file is located at:

application/config/database.php

The config settings are stored in a multi-dimensional array with this prototype:

```
$db['default']['hostname'] = "localhost";
$db['default']['username'] = "root";
$db['default']['password'] = "";
$db['default']['database'] = "database_name";
$db['default']['dbdriver'] = "mysql";
$db['default']['dbprefix'] = "";
$db['default']['pconnect'] = TRUE;
$db['default']['db_debug'] = FALSE;
$db['default']['cache_on'] = FALSE;
$db['default']['cachedir'] = "";
$db['default']['char_set'] = "utf8";
$db['default']['dbcollat'] = "utf8_general_ci";
```

The reason we use a multi-dimensional array rather than a more simple one is to permit you to optionally store multiple sets of connection values. If, for example, you run multiple environments (development, production, test, etc.) under a single installation, you can set up a connection group for each, then switch between groups as needed. For example, to set up a "test" environment you would do this:

```
$db['test']['hostname'] = "localhost";
$db['test']['username'] = "root";
$db['test']['password'] = "";
$db['test']['database'] = "database_name";
$db['test']['dbdriver'] = "mysql";
$db['test']['dbprefix'] = "";
$db['test']['pconnect'] = TRUE;
$db['test']['db_debug'] = FALSE;
$db['test']['cache_on'] = FALSE;
$db['test']['cachedir'] = "";
$db['test']['char_set'] = "utf8";
$db['test']['dbcollat'] = "utf8_general_ci";
```

Then, to globally tell the system to use that group you would set this variable located in the config file:

```
$active_group = "test";
```

Note: The name "test" is arbitrary. It can be anything you want. By default we've used the word "default" for the primary connection, but it too can be renamed to something more relevant to your project.

Active Record

The Active Record Class is globally enabled or disabled by setting the $active_record variable in the database configuration file to TRUE/FALSE (boolean). If you are not using the active record class, setting it to FALSE will utilize fewer resources when the database classes are initialized.

```
$active_record = TRUE;
```

Note: that some CodeIgniter classes such as Sessions require Active Records be enabled to access certain functionality.

Explanation of Values:

- **hostname** - The hostname of your database server. Often this is "localhost".
- **username** - The username used to connect to the database.
- **password** - The password used to connect to the database.
- **database** - The name of the database you want to connect to.
- **dbdriver** - The database type. ie: mysql, postgres, odbc, etc. Must be specified in lower case.
- **dbprefix** - An optional table prefix which will added to the table name when running Active Record queries. This permits multiple CodeIgniter installations to share one database.
- **pconnect** - TRUE/FALSE (boolean) - Whether to use a persistent connection.
- **db_debug** - TRUE/FALSE (boolean) - Whether database errors should be displayed.
- **cache_on** - TRUE/FALSE (boolean) - Whether database query caching is enabled, see also Database Caching Class.
- **cachedir** - The absolute server path to your database query cache directory.
- **char_set** - The character set used in communicating with the database.
- **dbcollat** - The character collation used in communicating with the database.
- **port** - The database port number. Currently only used with the Postgres driver. To use this value you have to add a line to the database config array.

  ```
  $db['default']['port'] = 5432;
  ```

Note: Depending on what database platform you are using (MySQL, Postgres, etc.) not all values will be needed. For example, when using SQLite you will not need to supply a username or password, and the database name will be the path to your database file. The information above assumes you are using MySQL.

Connecting to your Database

There are two ways to connect to a database:

Automatically Connecting

The "auto connect" feature will load and instantiate the database class with every page load. To enable "auto connecting", add the word **database** to the library array, as indicated in the following file:

application/config/autoload.php

Manually Connecting

If only some of your pages require database connectivity you can manually connect to your database by adding this line of code in any function where it is needed, or in your class constructor to make the database available globally in that class.

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

If the above function does **not** contain any information in the first parameter it will connect to the group specified in your database config file. For most people, this is the preferred method of use.

Available Parameters

1. The database connection values, passed either as an array or a DSN string.
2. TRUE/FALSE (boolean). Whether to return the connection ID (see Connecting to Multiple Databases below).
3. TRUE/FALSE (boolean). Whether to enable the Active Record class. Set to TRUE by default.

Manually Connecting to a Database

The first parameter of this function can **optionally** be used to specify a particular database group from your config file, or you can even submit connection values for a database that is not specified in your config file. Examples:

To choose a specific group from your config file you can do this:

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

Where **group_name** is the name of the connection group from your config file.

To connect manually to a desired database you can pass an array of values:

```
$config['hostname'] = "localhost";
$config['username'] = "myusername";
$config['password'] = "mypassword";
```

```
$config['database'] = "mydatabase";
$config['dbdriver'] = "mysql";
$config['dbprefix'] = "";
$config['pconnect'] = FALSE;
$config['db_debug'] = TRUE;
$config['cache_on'] = FALSE;
$config['cachedir'] = "";
$config['char_set'] = "utf8";
$config['dbcollat'] = "utf8_general_ci";

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

For information on each of these values please see the configuration page.

Or you can submit your database values as a Data Source Name. DSNs must have this prototype:

```
$dsn = 'dbdriver://username:password@hostname/database';

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

To override default config values when connecting with a DSN string, add the config variables as a query string.

```
$dsn = 'dbdriver://username:password@hostname/database?char_set=utf8&dbcollat=utf8_general_ci&
cache_on=true&cachedir=/path/to/cache';

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

Connecting to Multiple Databases

If you need to connect to more than one database simultaneously you can do so as follows:

```
$DB1 = $this->load->database('group_one', TRUE);
$DB2 = $this->load->database('group_two', TRUE);
```

Note: Change the words "group_one" and "group_two" to the specific group names you are connecting to (or you can pass the connection values as indicated above).

By setting the second parameter to TRUE (boolean) the function will return the database object.

When you connect this way, you will use your object name to issue commands rather than the syntax used throughout this guide. In other words, rather than issuing commands with:

$this->db->query();
$this->db->result();
etc...

You will instead use:

$DB1->query();
$DB1->result();
etc...

Reconnecting / Keeping the Connection Alive

If the database server's idle timeout is exceeded while you're doing some heavy PHP lifting (processing an image, for instance), you should consider pinging the server by using the **reconnect()** method before sending further queries, which can gracefully keep the connection alive or re-establish it.

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

Queries

$this->db->query();

To submit a query, use the following function:

```
$this->db->query('YOUR QUERY HERE');
```

The **query()** function returns a database result **object** when "read" type queries are run, which you can use to show your results. When "write" type queries are run it simply returns TRUE or FALSE depending on success or failure. When retrieving data you will typically assign the query to your own variable, like this:

```
$query = $this->db->query('YOUR QUERY HERE');
```

$this->db->simple_query();

This is a simplified version of the **$this->db->query()** function. It ONLY returns TRUE/FALSE on success or failure. It DOES NOT return a database result set, nor does it set the query timer, or compile bind data, or store your query for debugging. It simply lets you submit a query. Most users will rarely use this function.

Adding Database prefixes manually

If you have configured a database prefix and would like to add it in manually for, you can use the following.

```
$this->db->dbprefix('tablename');
// outputs prefix_tablename
```

Protecting identifiers

In many databases it is advisable to protect table and field names - for example with backticks in MySQL. **Active Record queries are automatically protected**, however if you need to manually protect an identifier you can use:

```
$this->db->protect_identifiers('table_name');
```

This function will also add a table prefix to your table, assuming you have a prefix specified in your database config file. To enable the prefixing set **TRUE** (boolen) via the second parameter:

```
$this->db->protect_identifiers('table_name', TRUE);
```

Escaping Queries

It's a very good security practice to escape your data before submitting it into your database. CodeIgniter has three methods that help you do this:

1. **$this->db->escape()** This function determines the data type so that it can escape only string data. It also automatically adds single quotes around the data so you don't have to:

   ```
   $sql = "INSERT INTO table (title) VALUES(".$this->db->escape($title).")";
   ```

2. **$this->db->escape_str()** This function escapes the data passed to it, regardless of type. Most of the time you'll use the above function rather than this one. Use the function like this:

   ```
   $sql = "INSERT INTO table (title) VALUES('".$this->db->escape_str($title)."')";
   ```

3. **$this->db->escape_like_str()** This method should be used when strings are to be used in LIKE conditions so that LIKE wildcards ('%', '_') in the string are also properly escaped.

   ```
   $search = '20% raise';
   $sql = "SELECT id FROM table WHERE column LIKE '%".$this->db->escape_like_str($search)."%'";
   ```

Query Bindings

Bindings enable you to simplify your query syntax by letting the system put the queries together for you. Consider the following example:

```
$sql = "SELECT * FROM some_table WHERE id = ? AND status = ? AND author = ?";

$this->db->query($sql, array(3, 'live', 'Rick'));
```

The question marks in the query are automatically replaced with the values in the array in the second parameter of the query function.

The secondary benefit of using binds is that the values are automatically escaped, producing safer queries. You don't have to remember to manually escape data; the engine does it automatically for you.

Generating Query Results

There are several ways to generate query results:

result()

This function returns the query result as an array of **objects**, or **an empty array** on failure. Typically you'll use this in a foreach loop, like this:

```
$query = $this->db->query("YOUR QUERY");

foreach ($query->result() as $row)
{
  echo $row->title;
  echo $row->name;
  echo $row->body;
}
```

The above **function** is an alias of **result_object()**.

If you run queries that might **not** produce a result, you are encouraged to test the result first:

```
$query = $this->db->query("YOUR QUERY");

if ($query->num_rows() > 0)
{
  foreach ($query->result() as $row)
  {
    echo $row->title;
    echo $row->name;
    echo $row->body;
  }
}
```

result_array()

This function returns the query result as a pure array, or an empty array when no result is produced. Typically you'll use this in a foreach loop, like this:

```
$query = $this->db->query("YOUR QUERY");

foreach ($query->result_array() as $row)
{
  echo $row['title'];
  echo $row['name'];
  echo $row['body'];
}
```

row()

This function returns a single result row. If your query has more than one row, it returns only the first row. The result is returned as an **object**. Here's a usage example:

```
$query = $this->db->query("YOUR QUERY");

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

  echo $row->title;
  echo $row->name;
  echo $row->body;
}
```

If you want a specific row returned you can submit the row number as a digit in the first parameter:

```
$row = $query->row(5);
```

row_array()

Identical to the above **row()** function, except it returns an array. Example:

```
$query = $this->db->query("YOUR QUERY");

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

  echo $row['title'];
  echo $row['name'];
  echo $row['body'];
}
```

If you want a specific row returned you can submit the row number as a digit in the first parameter:

```
$row = $query->row_array(5);
```

In addition, you can walk forward/backwards/first/last through your results using these variations:

$row = $query->first_row()
$row = $query->last_row()
$row = $query->next_row()
$row = $query->previous_row()

By default they return an object unless you put the word "array" in the parameter:

$row = $query->first_row('array')
$row = $query->last_row('array')
$row = $query->next_row('array')
$row = $query->previous_row('array')

Result Helper Functions

$query->num_rows()

The number of rows returned by the query. Note: In this example, **$query** is the variable that the query result object is assigned to:

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

echo $query->num_rows();
```

$query->num_fields()

The number of FIELDS (columns) returned by the query. Make sure to call the function using your query result object:

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

echo $query->num_fields();
```

$query->free_result()

It frees the memory associated with the result and deletes the result resource ID. Normally PHP frees its memory automatically at the end of script execution. However, if you are running a lot of queries in a particular script you might want to free the result after each query result has been generated in order to cut down on memory consumptions. Example:

```
$query = $this->db->query('SELECT title FROM my_table');

foreach ($query->result() as $row)
{
   echo $row->title;
}
$query->free_result(); // The $query result object will no longer be available

$query2 = $this->db->query('SELECT name FROM some_table');

$row = $query2->row();
echo $row->name;
$query2->free_result(); // The $query2 result object will no longer be available
```

Query Helper Functions

$this->db->insert_id()

The insert ID number when performing database inserts.

$this->db->affected_rows()

Displays the number of affected rows, when doing "write" type queries (insert, update, etc.).

Note: In MySQL "DELETE FROM TABLE" returns 0 affected rows. The database class has a small hack that allows it to return the correct number of affected rows. By default this hack is enabled but it can be turned off in the database driver file.

$this->db->count_all();

Permits you to determine the number of rows in a particular table. Submit the table name in the first parameter. Example:

```
echo $this->db->count_all('my_table');

// Produces an integer, like 25
```

$this->db->platform()

Outputs the database platform you are running (MySQL, MS SQL, Postgres, etc...):

```
echo $this->db->platform();
```

$this->db->version()

Outputs the database version you are running:

```
echo $this->db->version();
```

$this->db->last_query();

Returns the last query that was run (the query string, not the result). Example:

```
$str = $this->db->last_query();
```

```
// Produces: SELECT * FROM sometable....
```

The following two functions help simplify the process of writing database INSERTs and UPDATEs.

$this->db->insert_string();

This function simplifies the process of writing database inserts. It returns a correctly formatted SQL insert string. Example:

```
$data = array('name' => $name, 'email' => $email, 'url' => $url);

$str = $this->db->insert_string('table_name', $data);
```

The first parameter is the table name, the second is an associative array with the data to be inserted. The above example produces:

```
INSERT INTO table_name (name, email, url) VALUES ('Rick', 'rick@example.com', 'example.com')
```

Note: Values are automatically escaped, producing safer queries.

$this->db->update_string();

This function simplifies the process of writing database updates. It returns a correctly formatted SQL update string. Example:

```
$data = array('name' => $name, 'email' => $email, 'url' => $url);

$where = "author_id = 1 AND status = 'active'";

$str = $this->db->update_string('table_name', $data, $where);
```

The first parameter is the table name, the second is an associative array with the data to be updated, and the third parameter is the "where" clause. The above example produces:

```
UPDATE table_name SET name = 'Rick', email = 'rick@example.com', url = 'example.com' WHERE author_id = 1
AND status = 'active'
```

Note: Values are automatically escaped, producing safer queries.

Active Record Class

CodeIgniter uses a modified version of the Active Record Database Pattern. This pattern allows information to be retrieved, inserted, and updated in your database with minimal scripting. In some cases only one or two lines of code are necessary to perform a database action. CodeIgniter does not require that each database table be its own class file. It instead provides a more simplified interface.

Beyond simplicity, a major benefit to using the Active Record features is that it allows you to create database independent applications, since the query syntax is generated by each database adapter. It also allows for safer queries, since the values are escaped automatically by the system.

Note: If you intend to write your own queries you can disable this class in your database config file, allowing the core database library and adapter to utilize fewer resources.

Selecting Data

The following functions allow you to build SQL **SELECT** statements.

Note: If you are using PHP 5 you can use method chaining for more compact syntax. This is described at the end of the page.

$this->db->get();

Runs the selection query and returns the result. Can be used by itself to retrieve all records from a table:

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

// Produces: SELECT * FROM mytable
```

The second and third parameters enable you to set a limit and offset clause:

```
$query = $this->db->get('mytable', 10, 20);

// Produces: SELECT * FROM mytable LIMIT 20, 10 (in MySQL. Other databases have slightly different syntax)
```

You'll notice that the above function is assigned to a variable named **$query**, which can be used to show the results:

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

foreach ($query->result() as $row)
{
   echo $row->title;
}
```

Please visit the result functions page for a full discussion regarding result generation.

$this->db->get_where();

Identical to the above function except that it permits you to add a "where" clause in the second parameter, instead of using the db->where() function:

```
$query = $this->db->get_where('mytable', array('id' => $id), $limit, $offset);
```

Please read the about the where function below for more information.

Note: get_where() was formerly known as getwhere(), which has been deprecated

$this->db->select();

Permits you to write the SELECT portion of your query:

```
$this->db->select('title, content, date');

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

// Produces: SELECT title, content, date FROM mytable
```

Note: If you are selecting all (*) from a table you do not need to use this function. When omitted, CodeIgniter assumes you wish to SELECT *

$this->db->select() accepts an optional second parameter. If you set it to FALSE, CodeIgniter will not try to protect your field or table names with backticks. This is useful if you need a compound select statement.

```
$this->db->select('(SELECT SUM(payments.amount) FROM payments WHERE payments.invoice_id=4') AS
amount_paid', FALSE);
$query = $this->db->get('mytable');
```

$this->db->select_max();

Writes a "SELECT MAX(field)" portion for your query. You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_max('age');
$query = $this->db->get('members');
// Produces: SELECT MAX(age) as age FROM members

$this->db->select_max('age', 'member_age');
$query = $this->db->get('members');
// Produces: SELECT MAX(age) as member_age FROM members
```

$this->db->select_min();

Writes a "SELECT MIN(field)" portion for your query. As with **select_max()**, You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_min('age');
$query = $this->db->get('members');
// Produces: SELECT MIN(age) as age FROM members
```

$this->db->select_avg();

Writes a "SELECT AVG(field)" portion for your query. As with **select_max()**, You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_avg('age');
$query = $this->db->get('members');
// Produces: SELECT AVG(age) as age FROM members
```

$this->db->select_sum();

Writes a "SELECT SUM(field)" portion for your query. As with **select_max()**, You can optionally include a second parameter to rename the resulting field.

```
$this->db->select_sum('age');
$query = $this->db->get('members');
// Produces: SELECT SUM(age) as age FROM members
```

$this->db->from();

Permits you to write the FROM portion of your query:

```
$this->db->select('title, content, date');
$this->db->from('mytable');

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

// Produces: SELECT title, content, date FROM mytable
```

Note: As shown earlier, the FROM portion of your query can be specified in the **$this->db->get()** function, so use whichever method you prefer.

$this->db->join();

Permits you to write the JOIN portion of your query:

```
$this->db->select('*');
$this->db->from('blogs');
$this->db->join('comments', 'comments.id = blogs.id');

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

// Produces:
// SELECT * FROM blogs
// JOIN comments ON comments.id = blogs.id
```

Multiple function calls can be made if you need several joins in one query.

If you need something other than a natural JOIN you can specify it via the third parameter of the function. Options are: left, right, outer, inner, left outer, and right outer.

```
$this->db->join('comments', 'comments.id = blogs.id', 'left');

// Produces: LEFT JOIN comments ON comments.id = blogs.id
```

$this->db->where();

This function enables you to set **WHERE** clauses using one of four methods:

Note: All values passed to this function are escaped automatically, producing safer queries.

1. **Simple key/value method:**

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

   // Produces: WHERE name = 'Joe'
   ```

   Notice that the equal sign is added for you.

   If you use multiple function calls they will be chained together with **AND** between them:

   ```
   $this->db->where('name', $name);
   $this->db->where('title', $title);
   $this->db->where('status', $status);

   // WHERE name 'Joe' AND title = 'boss' AND status = 'active'
   ```

2. **Custom key/value method:**

You can include an operator in the first parameter in order to control the comparison:

```
$this->db->where('name !=', $name);
$this->db->where('id <', $id);

// Produces: WHERE name != 'Joe' AND id < 45
```

3. **Associative array method:**

```
$array = array('name' => $name, 'title' => $title, 'status' => $status);

$this->db->where($array);

// Produces: WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

   You can include your own operators using this method as well:

```
$array = array('name !=' => $name, 'id <' => $id, 'date >' => $date);

$this->db->where($array);
```

4. **Custom string:**

   You can write your own clauses manually:

```
$where = "name='Joe' AND status='boss' OR status='active'";

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

$this->db->where() accepts an optional third parameter. If you set it to FALSE, CodeIgniter will not try to protect your field or table names with backticks.

```
$this->db->where('MATCH (field) AGAINST ("value")', NULL, FALSE);
```

$this->db->or_where();

This function is identical to the one above, except that multiple instances are joined by OR:

```
$this->db->where('name !=', $name);
$this->db->or_where('id >', $id);

// Produces: WHERE name != 'Joe' OR id > 50
```

Note: or_where() was formerly known as orwhere(), which has been deprecated.

$this->db->where_in();

Generates a WHERE field IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->where_in('username', $names);
// Produces: WHERE username IN ('Frank', 'Todd', 'James')
```

$this->db->or_where_in();

Generates a WHERE field IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->or_where_in('username', $names);
// Produces: OR username IN ('Frank', 'Todd', 'James')
```

$this->db->where_not_in();

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with AND if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->where_not_in('username', $names);
// Produces: WHERE username NOT IN ('Frank', 'Todd', 'James')
```

$this->db->or_where_not_in();

Generates a WHERE field NOT IN ('item', 'item') SQL query joined with OR if appropriate

```
$names = array('Frank', 'Todd', 'James');
$this->db->or_where_not_in('username', $names);
// Produces: OR username NOT IN ('Frank', 'Todd', 'James')
```

$this->db->like();

This function enables you to generate **LIKE** clauses, useful for doing searches.

Note: All values passed to this function are escaped automatically.

1. **Simple key/value method:**

   ```
   $this->db->like('title', 'match');

   // Produces: WHERE title LIKE '%match%'
   ```

   If you use multiple function calls they will be chained together with **AND** between them:

   ```
   $this->db->like('title', 'match');
   $this->db->like('body', 'match');
   ```

```
// WHERE title LIKE '%match%' AND body LIKE '%match%
```

If you want to control where the wildcard (%) is placed, you can use an optional third argument. Your options are 'before', 'after' and 'both' (which is the default).

```
$this->db->like('title', 'match', 'before');
// Produces: WHERE title LIKE '%match'

$this->db->like('title', 'match', 'after');
// Produces: WHERE title LIKE 'match%'

$this->db->like('title', 'match', 'both');
// Produces: WHERE title LIKE '%match%'
```

2. **Associative array method:**

```
$array = array('title' => $match, 'page1' => $match, 'page2' => $match);

$this->db->like($array);

// WHERE title LIKE '%match%' AND page1 LIKE '%match%' AND page2 LIKE '%match%'
```

$this->db->or_like();

This function is identical to the one above, except that multiple instances are joined by OR:

```
$this->db->like('title', 'match');
$this->db->or_like('body', $match);

// WHERE title LIKE '%match%' OR body LIKE '%match%'
```

Note: or_like() was formerly known as orlike(), which has been deprecated.

$this->db->not_like();

This function is identical to **like()**, except that it generates NOT LIKE statements:

```
$this->db->not_like('title', 'match');

// WHERE title NOT LIKE '%match%
```

$this->db->or_not_like();

This function is identical to **not_like()**, except that multiple instances are joined by OR:

```
$this->db->like('title', 'match');
```

```
$this->db->or_not_like('body', 'match');

// WHERE title LIKE '%match% OR body NOT LIKE '%match%'
```

$this->db->group_by();

Permits you to write the GROUP BY portion of your query:

```
$this->db->group_by("title");

// Produces: GROUP BY title
```

You can also pass an array of multiple values as well:

```
$this->db->group_by(array("title", "date"));

// Produces: GROUP BY title, date
```

Note: group_by() was formerly known as groupby(), which has been deprecated.

$this->db->distinct();

Adds the "DISTINCT" keyword to a query

```
$this->db->distinct();
$this->db->get('table');

// Produces: SELECT DISTINCT * FROM table
```

$this->db->having();

Permits you to write the HAVING portion of your query. There are 2 possible syntaxe, 1 argument or 2:

```
$this->db->having('user_id = 45');
// Produces: HAVING user_id = 45

$this->db->having('user_id', 45);
// Produces: HAVING user_id = 45
```

You can also pass an array of multiple values as well:

```
$this->db->having(array('title =' => 'My Title', 'id <' => $id));

// Produces: HAVING title = 'My Title', id < 45
```

If you are using a database that CodeIgniter escapes queries for, you can prevent escaping content by passing an optional third argument, and setting it to FALSE.

```
$this->db->having('user_id', 45);
// Produces: HAVING `user_id` = 45 in some databases such as MySQL
$this->db->having('user_id', 45, FALSE);
// Produces: HAVING user_id = 45
```

$this->db->or_having();

Identical to having(), only separates multiple clauses with "OR".

$this->db->order_by();

Lets you set an ORDER BY clause. The first parameter contains the name of the column you would like to order by. The second parameter lets you set the direction of the result. Options are **asc** or **desc**, or **random**.

```
$this->db->order_by("title", "desc");

// Produces: ORDER BY title DESC
```

You can also pass your own string in the first parameter:

```
$this->db->order_by('title desc, name asc');

// Produces: ORDER BY title DESC, name ASC
```

Or multiple function calls can be made if you need multiple fields.

```
$this->db->order_by("title", "desc");
$this->db->order_by("name", "asc");

// Produces: ORDER BY title DESC, name ASC
```

Note: order_by() was formerly known as orderby(), which has been deprecated.

Note: random ordering is not currently supported in Oracle or MSSQL drivers. These will default to 'ASC'.

$this->db->limit();

Lets you limit the number of rows you would like returned by the query:

```
$this->db->limit(10);
```

```
// Produces: LIMIT 10
```

The second parameter lets you set a result offset.

```
$this->db->limit(10, 20);

// Produces: LIMIT 20, 10 (in MySQL. Other databases have slightly different syntax)
```

$this->db->count_all_results();

Permits you to determine the number of rows in a particular Active Record query. Queries will accept Active Record restrictors such as where(), or_where(), like(), or_like(), etc. Example:

```
echo $this->db->count_all_results('my_table');
// Produces an integer, like 25

$this->db->like('title', 'match');
$this->db->from('my_table');
echo $this->db->count_all_results();
// Produces an integer, like 17
```

$this->db->count_all();

Permits you to determine the number of rows in a particular table. Submit the table name in the first parameter. Example:

```
echo $this->db->count_all('my_table');

// Produces an integer, like 25
```

Inserting Data

$this->db->insert();

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
           'title' => 'My title' ,
           'name' => 'My Name' ,
           'date' => 'My date'
        );

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

// Produces: INSERT INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My date')
```

The first parameter will contain the table name, the second is an associative array of values.

Here is an example using an object:

```
/*
   class Myclass {
      var $title = 'My Title';
      var $content = 'My Content';
      var $date = 'My Date';
   }
*/

$object = new Myclass;

$this->db->insert('mytable', $object);

// Produces: INSERT INTO mytable (title, content, date) VALUES ('My Title', 'My Content', 'My Date')
```

The first parameter will contain the table name, the second is an associative array of values.

Note: All values are escaped automatically producing safer queries.

$this->db->set();

This function enables you to set values for **inserts** or **updates**.

It can be used instead of passing a data array directly to the insert or update functions:

```
$this->db->set('name', $name);
$this->db->insert('mytable');

// Produces: INSERT INTO mytable (name) VALUES ('{$name}')
```

If you use multiple function called they will be assembled properly based on whether you are doing an insert or an update:

```
$this->db->set('name', $name);
$this->db->set('title', $title);
$this->db->set('status', $status);
$this->db->insert('mytable');
```

set() will also accept an optional third parameter ($escape), that will prevent data from being escaped if set to FALSE. To illustrate the difference, here is set() used both with and without the escape parameter.

```
$this->db->set('field', 'field+1', FALSE);
$this->db->insert('mytable');
// gives INSERT INTO mytable (field) VALUES (field+1)

$this->db->set('field', 'field+1');
$this->db->insert('mytable');
// gives INSERT INTO mytable (field) VALUES ('field+1')
```

You can also pass an associative array to this function:

```
$array = array('name' => $name, 'title' => $title, 'status' => $status);

$this->db->set($array);
$this->db->insert('mytable');
```

Or an object:

```
/*
   class Myclass {
      var $title = 'My Title';
      var $content = 'My Content';
      var $date = 'My Date';
   }
*/

$object = new Myclass;

$this->db->set($object);
$this->db->insert('mytable');
```

Updating Data

$this->db->update();

Generates an update string and runs the query based on the data you supply. You can pass an **array** or an **object** to the function. Here is an example using an array:

```
$data = array(
            'title' => $title,
            'name' => $name,
            'date' => $date
         );

$this->db->where('id', $id);
$this->db->update('mytable', $data);

// Produces:
// UPDATE mytable
// SET title = '{$title}', name = '{$name}', date = '{$date}'
// WHERE id = $id
```

Or you can supply an object:

```
/*
   class Myclass {
      var $title = 'My Title';
      var $content = 'My Content';
      var $date = 'My Date';
   }
*/

$object = new Myclass;
```

```
$this->db->where('id', $id);
$this->db->update('mytable', $object);

// Produces:
// UPDATE mytable
// SET title = '{$title}', name = '{$name}', date = '{$date}'
// WHERE id = $id
```

Note: All values are escaped automatically producing safer queries.

You'll notice the use of the **$this->db->where()** function, enabling you to set the WHERE clause. You can optionally pass this information directly into the update function as a string:

```
$this->db->update('mytable', $data, "id = 4");
```

Or as an array:

```
$this->db->update('mytable', $data, array('id' => $id));
```

You may also use the **$this->db->set()** function described above when performing updates.

Deleting Data

$this->db->delete();

Generates a delete SQL string and runs the query.

```
$this->db->delete('mytable', array('id' => $id));

// Produces:
// DELETE FROM mytable
// WHERE id = $id
```

The first parameter is the table name, the second is the where clause. You can also use the **where()** or **or_where()** functions instead of passing the data to the second parameter of the function:

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

// Produces:
// DELETE FROM mytable
// WHERE id = $id
```

An array of table names can be passed into delete() if you would like to delete data from more than 1 table.

```
$tables = array('table1', 'table2', 'table3');
```

```
$this->db->where('id', '5');
$this->db->delete($tables);
```

If you want to delete all data from a table, you can use the **truncate()** function, or **empty_table()**.

$this->db->empty_table();

Generates a delete SQL string and runs the query.

```
$this->db->empty_table('mytable');

// Produces
// DELETE FROM mytable
```

$this->db->truncate();

Generates a truncate SQL string and runs the query.

```
$this->db->from('mytable');
$this->db->truncate();
// or
$this->db->truncate('mytable');

// Produce:
// TRUNCATE mytable
```

Note: If the TRUNCATE command isn't available, truncate() will execute as "DELETE FROM table".

Method Chaining

Method chaining allows you to simplify your syntax by connecting multiple functions. Consider this example:

```
$this->db->select('title')->from('mytable')->where('id', $id)->limit(10, 20);

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

Note: Method chaining only works with PHP 5.

Active Record Caching

While not "true" caching, Active Record enables you to save (or "cache") certain parts of your queries for reuse at a later point in your script's execution. Normally, when an Active Record call is completed, all stored information is reset for the next call. With caching, you can prevent this reset, and reuse information easily.

Cached calls are cumulative. If you make 2 cached select() calls, and then 2 uncached select() calls, this will result in 4 select() calls. There are three Caching functions available:

$this->db->start_cache()

This function must be called to begin caching. All Active Record queries of the correct type (see below for supported queries) are stored for later use.

$this->db->stop_cache()

This function can be called to stop caching.

$this->db->flush_cache()

This function deletes all items from the Active Record cache.

Here's a usage example:

```
$this->db->start_cache();
$this->db->select('field1');
$this->db->stop_cache();

$this->db->get('tablename');

//Generates: SELECT `field1` FROM (`tablename`)

$this->db->select('field2');
$this->db->get('tablename');

//Generates: SELECT `field1`, `field2` FROM (`tablename`)

$this->db->flush_cache();

$this->db->select('field2');
$this->db->get('tablename');

//Generates: SELECT `field2` FROM (`tablename`)
```

Note: The following statements can be cached: select, from, join, where, like, groupby, having, orderby, set

Transactions

CodeIgniter's database abstraction allows you to use **transactions** with databases that support transaction-safe table types. In MySQL, you'll need to be running InnoDB or BDB table types rather than the more common MyISAM. Most other database platforms support transactions natively.

If you are not familiar with transactions we recommend you find a good online resource to learn about them for your particular database. The information below assumes you have a basic understanding of transactions.

CodeIgniter's Approach to Transactions

CodeIgniter utilizes an approach to transactions that is very similar to the process used by the popular database class ADODB. We've chosen that approach because it greatly simplifies the process of running transactions. In most cases all that is required are two lines of code.

Traditionally, transactions have required a fair amount of work to implement since they demand that you to keep track of your queries and determine whether to **commit** or **rollback** based on the success or failure of your queries. This is particularly cumbersome with nested queries. In contrast, we've implemented a smart transaction system that does all this for you automatically (you can also manage your transactions manually if you choose to, but there's really no benefit).

Running Transactions

To run your queries using transactions you will use the **$this->db->trans_start()** and **$this->db->trans_complete()** functions as follows:

```
$this->db->trans_start();
$this->db->query('AN SQL QUERY...');
$this->db->query('ANOTHER QUERY...');
$this->db->query('AND YET ANOTHER QUERY...');
$this->db->trans_complete();
```

You can run as many queries as you want between the start/complete functions and they will all be committed or rolled back based on success or failure of any given query.

Strict Mode

By default CodeIgniter runs all transactions in **Strict Mode**. When strict mode is enabled, if you are running multiple groups of transactions, if one group fails all groups will be rolled back. If strict mode is disabled, each group is treated independently, meaning a failure of one group will not affect any others.

Strict Mode can be disabled as follows:

```
$this->db->trans_strict(FALSE);
```

Managing Errors

If you have error reporting enabled in your **config/database.php** file you'll see a standard error message if the commit was unsuccessful. If debugging is turned off, you can manage your own errors like this:

```
$this->db->trans_start();
$this->db->query('AN SQL QUERY...');
$this->db->query('ANOTHER QUERY...');
$this->db->trans_complete();

if ($this->db->trans_status() === FALSE)
{
   // generate an error... or use the log_message() function to log your error
}
```

Enabling Transactions

Transactions are enabled automatically the moment you use **$this->db->trans_start()**. If you would like to disable transactions you can do so using **$this->db->trans_off()**:

```
$this->db->trans_off()

$this->db->trans_start();
$this->db->query('AN SQL QUERY...');
$this->db->trans_complete();
```

When transactions are disabled, your queries will be auto-commited, just as they are when running queries without transactions.

Test Mode

You can optionally put the transaction system into "test mode", which will cause your queries to be rolled back -- even if the queries produce a valid result. To use test mode simply set the first parameter in the **$this->db->trans_start()** function to **TRUE**:

```
$this->db->trans_start(TRUE); // Query will be rolled back
$this->db->query('AN SQL QUERY...');
$this->db->trans_complete();
```

Running Transactions Manually

If you would like to run transactions manually you can do so as follows:

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

$this->db->query('AN SQL QUERY...');
```

```
$this->db->query('ANOTHER QUERY...');
$this->db->query('AND YET ANOTHER QUERY...');

if ($this->db->trans_status() === FALSE)
{
   $this->db->trans_rollback();
}
else
{
   $this->db->trans_commit();
}
```

Note: Make sure to use **$this->db->trans_begin()** when running manual transactions, **NOT** **$this->db->trans_start()**.

Table Data

These functions let you fetch table information.

$this->db->list_tables();

Returns an array containing the names of all the tables in the database you are currently connected to. Example:

```
$tables = $this->db->list_tables();

foreach ($tables as $table)
{
   echo $table;
}
```

$this->db->table_exists();

Sometimes it's helpful to know whether a particular table exists before running an operation on it. Returns a boolean TRUE/FALSE. Usage example:

```
if ($this->db->table_exists('table_name'))
{
   // some code...
}
```

Note: Replace *table_name* with the name of the table you are looking for.

Field Data

$this->db->list_fields()

Returns an array containing the field names. This query can be called two ways:

1. You can supply the table name and call it from the **$this->db->** object:

```
$fields = $this->db->list_fields('table_name');

foreach ($fields as $field)
{
   echo $field;
}
```

2. You can gather the field names associated with any query you run by calling the function from your query result object:

```
$query = $this->db->query('SELECT * FROM some_table');

foreach ($query->list_fields() as $field)
{
   echo $field;
}
```

$this->db->field_exists()

Sometimes it's helpful to know whether a particular field exists before performing an action. Returns a boolean TRUE/FALSE. Usage example:

```
if ($this->db->field_exists('field_name', 'table_name'))
{
   // some code...
}
```

Note: Replace *field_name* with the name of the column you are looking for, and replace *table_name* with the name of the table you are looking for.

$this->db->field_data()

Returns an array of objects containing field information.

Sometimes it's helpful to gather the field names or other metadata, like the column type, max length, etc.

Note: Not all databases provide meta-data.

Usage example:

```
$fields = $this->db->field_data('table_name');

foreach ($fields as $field)
{
  echo $field->name;
  echo $field->type;
  echo $field->max_length;
  echo $field->primary_key;
}
```

If you have run a query already you can use the result object instead of supplying the table name:

```
$query = $this->db->query("YOUR QUERY");
$fields = $query->field_data();
```

The following data is available from this function if supported by your database:

- name - column name
- max_length - maximum length of the column
- primary_key - 1 if the column is a primary key
- type - the type of the column

Custom Function Calls

$this->db->call_function();

This function enables you to call PHP database functions that are not natively included in CodeIgniter, in a platform independent manner. For example, lets say you want to call the **mysql_get_client_info()** function, which is **not** natively supported by CodeIgniter. You could do so like this:

```
$this->db->call_function('get_client_info');
```

You must supply the name of the function, **without** the **mysql_** prefix, in the first parameter. The prefix is added automatically based on which database driver is currently being used. This permits you to run the same function on different database platforms. Obviously not all function calls are identical between platforms, so there are limits to how useful this function can be in terms of portability.

Any parameters needed by the function you are calling will be added to the second parameter.

```
$this->db->call_function('some_function', $param1, $param2, etc..);
```

Often, you will either need to supply a database connection ID or a database result ID. The connection ID can be accessed using:

```
$this->db->conn_id;
```

The result ID can be accessed from within your result object, like this:

```
$query = $this->db->query("SOME QUERY");

$query->result_id;
```

Database Caching Class

The Database Caching Class permits you to cache your queries as text files for reduced database load.

Important: This class is initialized automatically by the database driver when caching is enabled. Do NOT load this class manually.

Also note: Not all query result functions are available when you use caching. Please read this page carefully.

Enabling Caching

Caching is enabled in three steps:

- Create a writable directory on your server where the cache files can be stored.
- Set the path to your cache folder in your **application/config/database.php** file.
- Enable the caching feature, either globally by setting the preference in your **application/config/database.php** file, or manually as described below.

Once enabled, caching will happen automatically whenever a page is loaded that contains database queries.

How Does Caching Work?

CodeIgniter's query caching system happens dynamically when your pages are viewed. When caching is enabled, the first time a web page is loaded, the query result object will be serialized and stored in a text file on your server. The next time the page is loaded the cache file will be used instead of accessing your database. Your database usage can effectively be reduced to zero for any pages that have been cached.

Only **read-type** (SELECT) queries can be cached, since these are the only type of queries that produce a result. **Write-type** (INSERT, UPDATE, etc.) queries, since they don't generate a result, will not be cached by the system.

Cache files DO NOT expire. Any queries that have been cached will remain cached until you delete them. The caching system permits you clear caches associated with individual pages, or you can delete the entire collection of cache files. Typically you'll want to use the housekeeping functions described below to delete cache files after certain events take place, like when you've added new information to your database.

Will Caching Improve Your Site's Performance?

Getting a performance gain as a result of caching depends on many factors. If you have a highly optimized database under very little load, you probably won't see a performance boost. If your database is under heavy use you probably will see an improved response, assuming your file-system is not overly taxed. Remember that caching simply changes how your information is retrieved, shifting it from being a database operation to a file-system one.

In some clustered server environments, for example, caching may be detrimental since file-system operations are so intense. On single servers in shared environments, caching will probably be beneficial. Unfortunately there is no single answer to the question of whether you should cache your database. It really depends on your situation.

How are Cache Files Stored?

CodeIgniter places the result of EACH query into its own cache file. Sets of cache files are further organized into sub-folders corresponding to your controller functions. To be precise, the sub-folders are named identically to the first two segments of your URI (the controller class name and function name).

For example, let's say you have a controller called **blog** with a function called **comments** that contains three queries. The caching system will create a cache folder called **blog+comments**, into which it will write three cache files.

If you use dynamic queries that change based on information in your URI (when using pagination, for example), each instance of the query will produce its own cache file. It's possible, therefore, to end up with many times more cache files than you have queries.

Managing your Cache Files

Since cache files do not expire, you'll need to build deletion routines into your application. For example, let's say you have a blog that allows user commenting. Whenever a new comment is submitted you'll want to delete the cache files associated with the controller function that serves up your comments. You'll find two delete functions described below that help you clear data.

Not All Database Functions Work with Caching

Lastly, we need to point out that the result object that is cached is a simplified version of the full result object. For that reason, some of the query result functions are not available for use.

The following functions **ARE NOT** available when using a cached result object:

- num_fields()
- field_names()
- field_data()
- free_result()

Also, the two database resources (result_id and conn_id) are not available when caching, since result resources only pertain to run-time operations.

Function Reference

$this->db->cache_on() / $this->db->cache_off()

Manually enables/disables caching. This can be useful if you want to keep certain queries from being cached. Example:

```
// Turn caching on
$this->db->cache_on();
$query = $this->db->query("SELECT * FROM mytable");

// Turn caching off for this one query
$this->db->cache_off();
$query = $this->db->query("SELECT * FROM members WHERE member_id = '$current_user'");

// Turn caching back on
$this->db->cache_on();
$query = $this->db->query("SELECT * FROM another_table");
```

$this->db->cache_delete()

Deletes the cache files associated with a particular page. This is useful if you need to clear caching after you update your database.

The caching system saves your cache files to folders that correspond to the URI of the page you are viewing. For example, if you are viewing a page at **example.com/index.php/blog/comments**, the caching system will put all cache files associated with it in a folder called **blog+comments**. To delete those particular cache files you will use:

```
$this->db->cache_delete('blog', 'comments');
```

If you do not use any parameters the current URI will be used when determining what should be cleared.

$this->db->cache_delete_all()

Clears all existing cache files. Example:

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

Database Forge Class

The Database Forge Class contains functions that help you manage your database.

Table of Contents

Initializing the Forge Class

Important: In order to initialize the Forge class, your database driver must already be running, since the forge class relies on it.

Load the Forge Class as follows:

```
$this->load->dbforge()
```

Once initialized you will access the functions using the **$this->dbforge** object:

```
$this->dbforge->some_function()
```

$this->dbforge->create_database('db_name')

Permits you to create the database specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($this->dbforge->create_database('my_db'))
{
    echo 'Database created!';
}
```

$this->dbforge->drop_database('db_name')

Permits you to drop the database specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($this->dbforge->drop_database('my_db'))
{
    echo 'Database deleted!';
}
```

Creating and Dropping Tables

There are several things you may wish to do when creating tables. Add fields, add keys to the table, alter columns. CodeIgniter provides a mechanism for this.

Adding fields

Fields are created via an associative array. Within the array you must include a 'type' key that relates to the datatype of the field. For example, INT, VARCHAR, TEXT, etc. Many datatypes (for example VARCHAR) also require a 'constraint' key.

```
$fields = array(
                        'users' => array(
                                                 'type' => 'VARCHAR',
                                                 'constraint' => '100',
                                          ),
                );

// will translate to "users VARCHAR(100)" when the field is added.
```

Additionally, the following key/values can be used:

- unsigned/true : to generate "UNSIGNED" in the field definition.
- default/value : to generate a default value in the field definition.
- null/true : to generate "NULL" in the field definition. Without this, the field will default to "NOT NULL".
- auto_increment/true : generates an auto_increment flag on the field. Note that the field type must be a type that supports this, such as integer.

```
$fields = array(
                        'blog_id' => array(
                                                 'type' => 'INT',
                                                 'constraint' => 5,
                                                 'unsigned' => TRUE,
                                                 'auto_increment' => TRUE
                                          ),
                        'blog_title' => array(
                                                 'type' => 'VARCHAR',
                                                 'constraint' => '100',
                                          ),
                        'blog_author' => array(
                                                 'type' =>'VARCHAR',
                                                 'constraint' => '100',
                                                 'default' => 'King of Town',
                          ),
                'blog_description' => array(
                                'type' => 'TEXT',
                                'null' => TRUE,
                          ),
          );
```

After the fields have been defined, they can be added using **$this->dbforge->add_field($fields);** followed by a call to the **create_table()** function.

$this->dbforge->add_field()

The add fields function will accept the above array.

Passing strings as fields

If you know exactly how you want a field to be created, you can pass the string into the field definitions with add_field()

```
$this->dbforge->add_field("label varchar(100) NOT NULL DEFAULT 'default label'");
```

Note: Multiple calls to **add_field()** are cumulative.

Creating an id field

There is a special exception for creating id fields. A field with type id will automatically be assinged as an INT(9) auto_incrementing Primary Key.

```
$this->dbforge->add_field('id');
// gives id INT(9) NOT NULL AUTO_INCREMENT
```

Adding Keys

Generally speaking, you'll want your table to have Keys. This is accomplished with **$this->dbforge->add_key('field')**. An optional second parameter set to TRUE will make it a primary key. Note that **add_key()** must be followed by a call to **create_table()**.

Multiple column non-primary keys must be sent as an array. Sample output below is for MySQL.

```
$this->dbforge->add_key('blog_id', TRUE);
// gives PRIMARY KEY `blog_id` (`blog_id`)

$this->dbforge->add_key('blog_id', TRUE);
$this->dbforge->add_key('site_id', TRUE);
// gives PRIMARY KEY `blog_id_site_id` (`blog_id`, `site_id`)

$this->dbforge->add_key('blog_name');
// gives KEY `blog_name` (`blog_name`)

$this->dbforge->add_key(array('blog_name', 'blog_label'));
// gives KEY `blog_name_blog_label` (`blog_name`, `blog_label`)
```

Creating a table

After fields and keys have been declared, you can create a new table with

```
$this->dbforge->create_table('table_name');
// gives CREATE TABLE table_name
```

An optional second parameter set to TRUE adds an "IF NOT EXISTS" clause into the definition

```
$this->dbforge->create_table('table_name', TRUE);
// gives CREATE TABLE IF NOT EXISTS table_name
```

Dropping a table

Executes a DROP TABLE sql

```
$this->dbforge->drop_table('table_name');
// gives DROP TABLE IF EXISTS table_name
```

Renaming a table

Executes a TABLE rename

```
$this->dbforge->rename_table('old_table_name', 'new_table_name');
// gives ALTER TABLE old_table_name RENAME TO new_table_name
```

Modifying Tables

$this->dbforge->add_column()

The add_column() function is used to modify an existing table. It accepts the same field array as above, and can be used for an unlimited number of additional fields.

```
$fields = array(
                'preferences' => array('type' => 'TEXT')
);
$this->dbforge->add_column('table_name', $fields);

// gives ALTER TABLE table_name ADD preferences TEXT
```

$this->dbforge->drop_column()

Used to remove a column from a table.

```
$this->dbforge->drop_column('table_name', 'column_to_drop');
```

$this->dbforge->modify_column()

The usage of this function is identical to add_column(), except it alters an existing column rather than adding a new one. In order to use it you must add a "name" key into the field defining array.

```
$fields = array(
                'old_name' => array(
                                    'name' => 'new_name',
                                    'type' => 'TEXT',
                                ),
);
$this->dbforge->modify_column('table_name', $fields);

// gives ALTER TABLE table_name CHANGE old_name new_name TEXT
```

Database Utility Class

The Database Utility Class contains functions that help you manage your database.

Table of Contents

Initializing the Utility Class

Important: In order to initialize the Utility class, your database driver must already be running, since the utilities class relies on it.

Load the Utility Class as follows:

```
$this->load->dbutil()
```

Once initialized you will access the functions using the **$this->dbutil** object:

```
$this->dbutil->some_function()
```

$this->dbutil->list_databases()

Returns an array of database names:

```
$dbs = $this->dbutil->list_databases();

foreach($dbs as $db)
{
   echo $db;
}
```

$this->dbutil->optimize_table('table_name');

Note: This features is only available for MySQL/MySQLi databases.

Permits you to optimize a table using the table name specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($this->dbutil->optimize_table('table_name'))
{
   echo 'Success!';
}
```

Note: Not all database platforms support table optimization.

$this->dbutil->repair_table('table_name');

Note: This features is only available for MySQL/MySQLi databases.

Permits you to repair a table using the table name specified in the first parameter. Returns TRUE/FALSE based on success or failure:

```
if ($this->dbutil->repair_table('table_name'))
{
   echo 'Success!';
}
```

Note: Not all database platforms support table repairs.

$this->dbutil->optimize_database();

Note: This features is only available for MySQL/MySQLi databases.

Permits you to optimize the database your DB class is currently connected to. Returns an array containing the DB status messages or FALSE on failure.

```
$result = $this->dbutil->optimize_database();

if ($result !== FALSE)
{
   print_r($result);
}
```

Note: Not all database platforms support table optimization.

$this->dbutil->csv_from_result($db_result)

Permits you to generate a CSV file from a query result. The first parameter of the function must contain the result object from your query. Example:

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

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

echo $this->dbutil->csv_from_result($query);
```

The second and third parameters allows you to set the delimiter and newline character. By default tabs are used as the delimiter and "\n" is used as a new line. Example:

```
$delimiter = ",";
$newline = "\r\n";

echo $this->dbutil->csv_from_result($query, $delimiter, $newline);
```

Important: This function will NOT write the CSV file for you. It simply creates the CSV layout. If you need to write the file use the File Helper.

$this->dbutil->xml_from_result($db_result)

Permits you to generate an XML file from a query result. The first parameter expects a query result object, the second may contain an optional array of config parameters. Example:

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

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

$config = array (
              'root'    => 'root',
              'element' => 'element',
              'newline' => "\n",
              'tab'     => "\t"
            );

echo $this->dbutil->xml_from_result($query, $config);
```

Important: This function will NOT write the XML file for you. It simply creates the XML layout. If you need to write the file use the File Helper.

$this->dbutil->backup()

Permits you to backup your full database or individual tables. The backup data can be compressed in either Zip or Gzip format.

Note: This features is only available for MySQL databases.

Note: Due to the limited execution time and memory available to PHP, backing up very large databases may not be possible. If your database is very large you might need to backup directly from your SQL server via the command line, or have your server admin do it for you if you do not have root privileges.

Usage Example

```
// Load the DB utility class
$this->load->dbutil();

// Backup your entire database and assign it to a variable
$backup =& $this->dbutil->backup();

// Load the file helper and write the file to your server
$this->load->helper('file');
write_file('/path/to/mybackup.gz', $backup);

// Load the download helper and send the file to your desktop
$this->load->helper('download');
force_download('mybackup.gz', $backup);
```

Setting Backup Preferences

Backup preferences are set by submitting an array of values to the first parameter of the backup function. Example:

```
$prefs = array(
                'tables'      => array('table1', 'table2'),  // Array of tables to backup.
                'ignore'      => array(),           // List of tables to omit from the backup
                'format'      => 'txt',             // gzip, zip, txt
                'filename'    => 'mybackup.sql',    // File name - NEEDED ONLY WITH ZIP FILES
                'add_drop'    => TRUE,              // Whether to add DROP TABLE statements to backup file
                'add_insert'  => TRUE,              // Whether to add INSERT data to backup file
                'newline'     => "\n"               // Newline character used in backup file
              );

$this->dbutil->backup($prefs);
```

Description of Backup Preferences

Preference	Default Value	Options	Description
tables	empty array	None	An array of tables you want backed up. If left blank all tables will be exported.
ignore	empty array	None	An array of tables you want the backup routine to ignore.
format	gzip	gzip, zip, txt	The file format of the export file.
filename	the current date/time	None	The name of the backed-up file. The name is needed only if you are using zip compression.
add_drop	TRUE	TRUE/FALSE	Whether to include DROP TABLE statements in your SQL export file.
add_insert	TRUE	TRUE/FALSE	Whether to include INSERT statements in your SQL export file.
newline	"\n"	"\n", "\r", "\r\n"	Type of newline to use in your SQL export file.

Email Class

CodeIgniter's robust Email Class supports the following features:

- Multiple Protocols: Mail, Sendmail, and SMTP
- Multiple recipients
- CC and BCCs
- HTML or Plaintext email
- Attachments
- Word wrapping
- Priorities
- BCC Batch Mode, enabling large email lists to be broken into small BCC batches.
- Email Debugging tools

Sending Email

Sending email is not only simple, but you can configure it on the fly or set your preferences in a config file.

Here is a basic example demonstrating how you might send email. Note: This example assumes you are sending the email from one of your controllers.

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

$this->email->from('your@example.com', 'Your Name');
$this->email->to('someone@example.com');
$this->email->cc('another@another-example.com');
$this->email->bcc('them@their-example.com');

$this->email->subject('Email Test');
$this->email->message('Testing the email class.');

$this->email->send();

echo $this->email->print_debugger();
```

Setting Email Preferences

There are 17 different preferences available to tailor how your email messages are sent. You can either set them manually as described here, or automatically via preferences stored in your config file, described below:

Preferences are set by passing an array of preference values to the email **initialize** function. Here is an example of how you might set some preferences:

```
$config['protocol'] = 'sendmail';
$config['mailpath'] = '/usr/sbin/sendmail';
```

```
$config['charset'] = 'iso-8859-1';
$config['wordwrap'] = TRUE;

$this->email->initialize($config);
```

Note: Most of the preferences have default values that will be used if you do not set them.

Setting Email Preferences in a Config File

If you prefer not to set preferences using the above method, you can instead put them into a config file. Simply create a new file called the **email.php**, add the **$config** array in that file. Then save the file at **config/email.php** and it will be used automatically. You will NOT need to use the **$this->email->initialize()** function if you save your preferences in a config file.

Email Preferences

The following is a list of all the preferences that can be set when sending email.

Preference	Default Value	Options	Description
useragent	CodeIgniter	None	The "user agent".
protocol	mail	mail, sendmail, or smtp	The mail sending protocol.
mailpath	/usr/sbin /sendmail	None	The server path to Sendmail.
smtp_host	No Default	None	SMTP Server Address.
smtp_user	No Default	None	SMTP Username.
smtp_pass	No Default	None	SMTP Password.
smtp_port	25	None	SMTP Port.
smtp_timeout	5	None	SMTP Timeout (in seconds).
wordwrap	TRUE	TRUE or FALSE (boolean)	Enable word-wrap.
wrapchars	76		Character count to wrap at.
mailtype	text	text or html	Type of mail. If you send HTML email you must send it as a complete web page. Make sure you don't have any relative links or relative image paths otherwise they will not work.
charset	utf-8		Character set (utf-8, iso-8859-1, etc.).
validate	FALSE	TRUE or FALSE (boolean)	Whether to validate the email address.
priority	3	1, 2, 3, 4, 5	Email Priority. 1 = highest. 5 = lowest. 3 = normal.
crlf	\n	"\r\n" or "\n" or "\r"	Newline character. (Use "\r\n" to comply with RFC 822).
newline	\n	"\r\n" or "\n" or "\r"	Newline character. (Use "\r\n" to comply with RFC 822).

bcc_batch_mode	FALSE	TRUE or FALSE (boolean)	Enable BCC Batch Mode.
bcc_batch_size	200	None	Number of emails in each BCC batch.

Email Function Reference

$this->email->from()

Sets the email address and name of the person sending the email:

```
$this->email->from('you@example.com', 'Your Name');
```

$this->email->reply_to()

Sets the reply-to address. If the information is not provided the information in the "from" function is used. Example:

```
$this->email->reply_to('you@example.com', 'Your Name');
```

$this->email->to()

Sets the email address(s) of the recipient(s). Can be a single email, a comma-delimited list or an array:

```
$this->email->to('someone@example.com');
```

```
$this->email->to('one@example.com, two@example.com, three@example.com');
```

```
$list = array('one@example.com', 'two@example.com', 'three@example.com');

$this->email->to($list);
```

$this->email->cc()

Sets the CC email address(s). Just like the "to", can be a single email, a comma-delimited list or an array.

$this->email->bcc()

Sets the BCC email address(s). Just like the "to", can be a single email, a comma-delimited list or an array.

$this->email->subject()

Sets the email subject:

```
$this->email->subject('This is my subject');
```

$this->email->message()

Sets the email message body:

```
$this->email->message('This is my message');
```

$this->email->set_alt_message()

Sets the alternative email message body:

```
$this->email->set_alt_message('This is the alternative message');
```

This is an optional message string which can be used if you send HTML formatted email. It lets you specify an alternative message with no HTML formatting which is added to the header string for people who do not accept HTML email. If you do not set your own message CodeIgniter will extract the message from your HTML email and strip the tags.

$this->email->clear()

Initializes all the email variables to an empty state. This function is intended for use if you run the email sending function in a loop, permitting the data to be reset between cycles.

```
foreach ($list as $name => $address)
{
   $this->email->clear();

   $this->email->to($address);
   $this->email->from('your@example.com');
   $this->email->subject('Here is your info '.$name);
   $this->email->message('Hi '.$name.' Here is the info you requested.');
   $this->email->send();
}
```

If you set the parameter to TRUE any attachments will be cleared as well:

```
$this->email->clear(TRUE);
```

$this->email->send()

The Email sending function. Returns boolean TRUE or FALSE based on success or failure, enabling it to be used conditionally:

```
if ( ! $this->email->send())
{
   // Generate error
}
```

$this->email->attach()

Enables you to send an attachment. Put the file path/name in the first parameter. Note: Use a file path, not a URL. For multiple attachments use the function multiple times. For example:

```
$this->email->attach('/path/to/photo1.jpg');
$this->email->attach('/path/to/photo2.jpg');
$this->email->attach('/path/to/photo3.jpg');

$this->email->send();
```

$this->email->print_debugger()

Returns a string containing any server messages, the email headers, and the email messsage. Useful for debugging.

Overriding Word Wrapping

If you have word wrapping enabled (recommended to comply with RFC 822) and you have a very long link in your email it can get wrapped too, causing it to become un-clickable by the person receiving it. CodeIgniter lets you manually override word wrapping within part of your message like this:

```
The text of your email that
gets wrapped normally.

{unwrap}http://example.com/a_long_link_that_should_not_be_wrapped.html{/unwrap}

More text that will be
wrapped normally.
```

Place the item you do not want word-wrapped between: **{unwrap} {/unwrap}**

Encryption Class

The Encryption Class provides two-way data encryption. It uses a scheme that pre-compiles the message using a randomly hashed bitwise XOR encoding scheme, which is then encrypted using the Mcrypt library. If Mcrypt is not available on your server the encoded message will still provide a reasonable degree of security for encrypted sessions or other such "light" purposes. If Mcrypt is available, you'll effectively end up with a double-encrypted message string, which should provide a very high degree of security.

Setting your Key

A *key* is a piece of information that controls the cryptographic process and permits an encrypted string to be decoded. In fact, the key you chose will provide the **only** means to decode data that was encrypted with that key, so not only must you choose the key carefully, you must never change it if you intend use it for persistent data.

It goes without saying that you should guard your key carefully. Should someone gain access to your key, the data will be easily decoded. If your server is not totally under your control it's impossible to ensure key security so you may want to think carefully before using it for anything that requires high security, like storing credit card numbers.

To take maximum advantage of the encryption algorithm, your key should be 32 characters in length (128 bits). The key should be as random a string as you can concoct, with numbers and uppercase and lowercase letters. Your key should **not** be a simple text string. In order to be cryptographically secure it needs to be as random as possible.

Your key can be either stored in your **application/config/config.php**, or you can design your own storage mechanism and pass the key dynamically when encoding/decoding.

To save your key to your **application/config/config.php**, open the file and set:

```
$config['encryption_key'] = "YOUR KEY";
```

Message Length

It's important for you to know that the encoded messages the encryption function generates will be approximately 2.6 times longer than the original message. For example, if you encrypt the string "my super secret data", which is 21 characters in length, you'll end up with an encoded string that is roughly 55 characters (we say "roughly" because the encoded string length increments in 64 bit clusters, so it's not exactly linear). Keep this information in mind when selecting your data storage mechanism. Cookies, for example, can only hold 4K of information.

Initializing the Class

Like most other classes in CodeIgniter, the Encryption class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Encrypt library object will be available using: **$this->encrypt**

$this->encrypt->encode()

Performs the data encryption and returns it as a string. Example:

```
$msg = 'My secret message';

$encrypted_string = $this->encrypt->encode($msg);
```

You can optionally pass your encryption key via the second parameter if you don't want to use the one in your config file:

```
$msg = 'My secret message';
$key = 'super-secret-key';

$encrypted_string = $this->encrypt->encode($msg, $key);
```

$this->encrypt->decode()

Decrypts an encoded string. Example:

```
$encrypted_string = 'APANtByIGI1BpVXZTJgcsAG8GZl8pdwwa84';

$plaintext_string = $this->encrypt->decode($encrypted_string);
```

$this->encrypt->set_cipher();

Permits you to set an Mcrypt cipher. By default it uses **MCRYPT_RIJNDAEL_256**. Example:

```
$this->encrypt->set_cipher(MCRYPT_BLOWFISH);
```

Please visit php.net for a list of available ciphers.

If you'd like to manually test whether your server supports Mcrypt you can use:

```
echo ( ! function_exists('mcrypt_encrypt')) ? 'Nope' : 'Yup';
```

$this->encrypt->set_mode();

Permits you to set an Mcrypt mode. By default it uses **MCRYPT_MODE_ECB**. Example:

```
$this->encrypt->set_mode(MCRYPT_MODE_CFB);
```

Please visit php.net for a list of available modes.

$this->encrypt->sha1();

SHA1 encoding function. Provide a string and it will return a 160 bit one way hash. Note: SHA1, just like MD5 is non-decodable. Example:

```
$hash = $this->encrypt->sha1('Some string');
```

Many PHP installations have SHA1 support by default so if all you need is to encode a hash it's simpler to use the native function:

```
$hash = sha1('Some string');
```

If your server does not support SHA1 you can use the provided function.

File Uploading Class

CodeIgniter's File Uploading Class permits files to be uploaded. You can set various preferences, restricting the type and size of the files.

The Process

Uploading a file involves the following general process:

- An upload form is displayed, allowing a user to select a file and upload it.
- When the form is submitted, the file is uploaded to the destination you specify.
- Along the way, the file is validated to make sure it is allowed to be uploaded based on the preferences you set.
- Once uploaded, the user will be shown a success message.

To demonstrate this process here is brief tutorial. Afterward you'll find reference information.

Creating the Upload Form

Using a text editor, create a form called **upload_form.php**. In it, place this code and save it to your **applications/views/** folder:

```
<html>
<head>
<title>Upload Form</title>
</head>
<body>

<?php echo $error;?>

<?php echo form_open_multipart('upload/do_upload');?>

<input type="file" name="userfile" size="20" />

<br /><br />

<input type="submit" value="upload" />

</form>

</body>
</html>
```

You'll notice we are using a form helper to create the opening form tag. File uploads require a multipart form, so the helper creates the proper syntax for you. You'll also notice we have an $error variable. This is so we can show error messages in the event the user does something wrong.

The Success Page

Using a text editor, create a form called **upload_success.php**. In it, place this code and save it to your **applications/views/** folder:

```
<html>
<head>
<title>Upload Form</title>
</head>
<body>

<h3>Your file was successfully uploaded!</h3>

<ul>
<?php foreach($upload_data as $item => $value):?>
<li><?php echo $item;?>: <?php echo $value;?></li>
<?php endforeach; ?>
</ul>

<p><?php echo anchor('upload', 'Upload Another File!'); ?></p>

</body>
</html>
```

The Controller

Using a text editor, create a controller called **upload.php**. In it, place this code and save it to your **applications/controllers/** folder:

```
<?php

class Upload extends Controller {

	function Upload()
	{
		parent::Controller();
		$this->load->helper(array('form', 'url'));
	}

	function index()
	{
		$this->load->view('upload_form', array('error' => ' ' ));
	}

	function do_upload()
	{
		$config['upload_path'] = './uploads/';
		$config['allowed_types'] = 'gif|jpg|png';
		$config['max_size']	= '100';
		$config['max_width']  = '1024';
		$config['max_height']  = '768';

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

		if ( ! $this->upload->do_upload())
		{
			$error = array('error' => $this->upload->display_errors());

			$this->load->view('upload_form', $error);
		}
		else
		{
			$data = array('upload_data' => $this->upload->data());

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

The Upload Folder

You'll need a destination folder for your uploaded images. Create a folder at the root of your CodeIgniter installation called **uploads** and set its file permissions to 777.

Try it!

To try your form, visit your site using a URL similar to this one:

```
example.com/index.php/upload/
```

You should see an upload form. Try uploading an image file (either a jpg, gif, or png). If the path in your controller is correct it should work.

Reference Guide

Initializing the Upload Class

Like most other classes in CodeIgniter, the Upload class is initialized in your controller using the **$this->load->library** function:

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

Once the Upload class is loaded, the object will be available using: **$this->upload**

Setting Preferences

Similar to other libraries, you'll control what is allowed to be upload based on your preferences. In the controller you built above you set the following preferences:

```
$config['upload_path'] = './uploads/';
$config['allowed_types'] = 'gif|jpg|png';
$config['max_size'] = '100';
$config['max_width'] = '1024';
$config['max_height'] = '768';

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

// Alternately you can set preferences by calling the initialize function. Useful if you auto-load the class:
$this->upload->initialize($config);
```

The above preferences should be fairly self-explanatory. Below is a table describing all available preferences.

Preferences

The following preferences are available. The default value indicates what will be used if you do not specify that preference.

Preference	Default Value	Options	Description
upload_path	None	None	The path to the folder where the upload should be placed. The folder must be writable and the path can be absolute or relative.
allowed_types	None	None	The mime types corresponding to the types of files you allow to be uploaded. Usually the file extension can be used as the mime type. Separate multiple types with a pipe.

file_name	None	Desired file name	If set CodeIgniter will rename the uploaded file to this name. **Note:**The filename should not include a file extension.
overwrite	FALSE	TRUE/FALSE (boolean)	If set to true, if a file with the same name as the one you are uploading exists, it will be overwritten. If set to false, a number will be appended to the filename if another with the same name exists.
max_size	0	None	The maximum size (in kilobytes) that the file can be. Set to zero for no limit. Note: Most PHP installations have their own limit, as specified in the php.ini file. Usually 2 MB (or 2048 KB) by default.
max_width	0	None	The maximum width (in pixels) that the file can be. Set to zero for no limit.
max_height	0	None	The maximum height (in pixels) that the file can be. Set to zero for no limit.
max_filename	0	None	The maximum length that a file name can be. Set to zero for no limit.
encrypt_name	FALSE	TRUE/FALSE (boolean)	If set to TRUE the file name will be converted to a random encrypted string. This can be useful if you would like the file saved with a name that can not be discerned by the person uploading it.
remove_spaces	TRUE	TRUE/FALSE (boolean)	If set to TRUE, any spaces in the file name will be converted to underscores. This is recommended.

Setting preferences in a config file

If you prefer not to set preferences using the above method, you can instead put them into a config file. Simply create a new file called the **upload.php**, add the **$config** array in that file. Then save the file in: **config/upload.php** and it will be used automatically. You will NOT need to use the **$this->upload->initialize** function if you save your preferences in a config file.

Function Reference

The following functions are available

$this->upload->do_upload()

Performs the upload based on the preferences you've set. Note: By default the upload routine expects the file to come from a form field called **userfile**, and the form must be a "multipart type:

```
<form method="post" action="some_action" enctype="multipart/form-data" />
```

If you would like to set your own field name simply pass its value to the **do_upload** function:

```
$field_name = "some_field_name";
$this->upload->do_upload($field_name)
```

$this->upload->display_errors()

Retrieves any error messages if the **do_upload()** function returned false. The function does not echo automatically, it returns the data so you can assign it however you need.

Formatting Errors

By default the above function wraps any errors within <p> tags. You can set your own delimiters like this:

```
$this->upload->display_errors('<p>', '</p>');
```

$this->upload->data()

This is a helper function that returns an array containing all of the data related to the file you uploaded. Here is the array prototype:

```
Array
(
    [file_name]    => mypic.jpg
    [file_type]    => image/jpeg
    [file_path]    => /path/to/your/upload/
    [full_path]    => /path/to/your/upload/jpg.jpg
    [raw_name]     => mypic
    [orig_name]    => mypic.jpg
    [file_ext]     => .jpg
    [file_size]    => 22.2
    [is_image]     => 1
    [image_width]  => 800
    [image_height] => 600
    [image_type]   => jpeg
    [image_size_str] => width="800" height="200"
)
```

Explanation

Here is an explanation of the above array items.

Item	Description
file_name	The name of the file that was uploaded including the file extension.
file_type	The file's Mime type
file_path	The absolute server path to the file
full_path	The absolute server path including the file name
raw_name	The file name without the extension
orig_name	The original file name. This is only useful if you use the encrypted name option.
file_ext	The file extension with period
file_size	The file size in kilobytes
is_image	Whether the file is an image or not. 1 = image. 0 = not.

image_width	Image width.
image_heigth	Image height
image_type	Image type. Typically the file extension without the period.
image_size_str	A string containing the width and height. Useful to put into an image tag.

Form Validation

CodeIgniter provides a comprehensive form validation and data prepping class that helps minimize the amount of code you'll write.

Note: As of CodeIgniter 1.7.0, this Form Validation class supercedes the old Validation class, which is now deprecated. We have left the old class in the library so applications currently using it will not break, but you are encouraged to migrate to this new version.

Overview

Before explaining CodeIgniter's approach to data validation, let's describe the ideal scenario:

1. A form is displayed.
2. You fill it in and submit it.
3. If you submitted something invalid, or perhaps missed a required item, the form is redisplayed containing your data along with an error message describing the problem.
4. This process continues until you have submitted a valid form.

On the receiving end, the script must:

1. Check for required data.
2. Verify that the data is of the correct type, and meets the correct criteria. For example, if a username is submitted it must be validated to contain only permitted characters. It must be of a minimum length, and not exceed a maximum length. The username can't be someone else's existing username, or perhaps even a reserved word. Etc.
3. Sanitize the data for security.
4. Pre-format the data if needed (Does the data need to be trimmed? HTML encoded? Etc.)
5. Prep the data for insertion in the database.

Although there is nothing terribly complex about the above process, it usually requires a significant amount of code, and to display error messages, various control structures are usually placed within the form HTML. Form validation, while simple to create, is generally very messy and tedious to implement.

Form Validation Tutorial

What follows is a "hands on" tutorial for implementing CodeIgniters Form Validation.

In order to implement form validation you'll need three things:

1. A View file containing a form.
2. A View file containing a "success" message to be displayed upon successful submission.
3. A controller function to receive and process the submitted data.

Let's create those three things, using a member sign-up form as the example.

The Form

Using a text editor, create a form called **myform.php**. In it, place this code and save it to your **applications/views/** folder:

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<?php echo validation_errors(); ?>

<?php echo form_open('form'); ?>

<h5>Username</h5>
<input type="text" name="username" value="" size="50" />

<h5>Password</h5>
<input type="text" name="password" value="" size="50" />

<h5>Password Confirm</h5>
<input type="text" name="passconf" value="" size="50" />

<h5>Email Address</h5>
<input type="text" name="email" value="" size="50" />

<div><input type="submit" value="Submit" /></div>

</form>

</body>
</html>
```

The Success Page

Using a text editor, create a form called **formsuccess.php**. In it, place this code and save it to your **applications/views/** folder:

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<h3>Your form was successfully submitted!</h3>

<p><?php echo anchor('form', 'Try it again!'); ?></p>

</body>
</html>
```

The Controller

Using a text editor, create a controller called **form.php**. In it, place this code and save it to your

applications/controllers/ folder:

```
<?php

class Form extends Controller {

	function index()
	{
		$this->load->helper(array('form', 'url'));

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

		if ($this->form_validation->run() == FALSE)
		{
			$this->load->view('myform');
		}
		else
		{
			$this->load->view('formsuccess');
		}
	}
}
?>
```

Try it!

To try your form, visit your site using a URL similar to this one:

```
example.com/index.php/form/
```

If you submit the form you should simply see the form reload. That's because you haven't set up any validation rules yet.

Since you haven't told the Form Validation class to validate anything yet, it returns FALSE (boolean false) by default. The run() function only returns TRUE if it has successfully applied your rules without any of them failing.

Explanation

You'll notice several things about the above pages:

The **form** (myform.php) is a standard web form with a couple exceptions:

1. It uses a **form helper** to create the form opening. Technically, this isn't necessary. You could create the form using standard HTML. However, the benefit of using the helper is that it generates the action URL for you, based on the URL in your config file. This makes your application more portable in the event your URLs change.
2. At the top of the form you'll notice the following function call:

   ```
   <?php echo validation_errors(); ?>
   ```

   This function will return any error messages sent back by the validator. If there are no messages it returns an empty string.

The **controller** (form.php) has one function: **index()**. This function initializes the validation class and loads the **form helper** and **URL helper** used by your view files. It also **runs** the validation routine. Based on whether the validation was successful it either presents the form or the success page.

Setting Validation Rules

CodeIgniter lets you set as many validation rules as you need for a given field, cascading them in order, and it even lets you prep and pre-process the field data at the same time. To set validation rules you will use the **set_rules()** function:

```
$this->form_validation->set_rules();
```

The above function takes **three** parameters as input:

1. The field name - the exact name you've given the form field.
2. A "human" name for this field, which will be inserted into the error message. For example, if your field is named "user" you might give it a human name of "Username". **Note:** If you would like the field name to be stored in a language file, please see Translating Field Names.
3. The validation rules for this form field.

Here is an example. In your **controller** (form.php), add this code just below the validation initialization function:

```
$this->form_validation->set_rules('username', 'Username', 'required');
$this->form_validation->set_rules('password', 'Password', 'required');
$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
$this->form_validation->set_rules('email', 'Email', 'required');
```

Your controller should now look like this:

```
<?php

class Form extends Controller {

	function index()
	{
		$this->load->helper(array('form', 'url'));

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

		$this->form_validation->set_rules('username', 'Username', 'required');
		$this->form_validation->set_rules('password', 'Password', 'required');
		$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
		$this->form_validation->set_rules('email', 'Email', 'required');

		if ($this->form_validation->run() == FALSE)
		{
			$this->load->view('myform');
		}
		else
		{
			$this->load->view('formsuccess');
		}
	}
}
?>
```

Now submit the form with the fields blank and you should see the error messages. If you submit the form with all the fields populated you'll see your success page.

Note: The form fields are not yet being re-populated with the data when there is an error. We'll get to that shortly.

Setting Rules Using an Array

Before moving on it should be noted that the rule setting function can be passed an array if you prefer to set all your rules in one action. If you use this approach you must name your array keys as indicated:

```
$config = array(
          array(
              'field'   => 'username',
              'label'   => 'Username',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'password',
              'label'   => 'Password',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'passconf',
              'label'   => 'Password Confirmation',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'email',
              'label'   => 'Email',
              'rules'   => 'required'
            )
        );

$this->form_validation->set_rules($config);
```

Cascading Rules

CodeIgniter lets you pipe multiple rules together. Let's try it. Change your rules in the third parameter of rule setting function, like this:

```
$this->form_validation->set_rules('username', 'Username', 'required|min_length[5]|max_length[12]');
$this->form_validation->set_rules('password', 'Password', 'required|matches[passconf]');
$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
$this->form_validation->set_rules('email', 'Email', 'required|valid_email');
```

The above code sets the following rules:

1. The username field be no shorter than 5 characters and no longer than 12.
2. The password field must match the password confirmation field.
3. The email field must contain a valid email address.

Give it a try! Submit your form without the proper data and you'll see new error messages that correspond to your new rules. There are numerous rules available which you can read about in the validation reference.

Prepping Data

In addition to the validation functions like the ones we used above, you can also prep your data in various ways. For example, you can set up rules like this:

```
$this->form_validation->set_rules('username', 'Username',
'trim|required|min_length[5]|max_length[12]|xss_clean');
$this->form_validation->set_rules('password', 'Password', 'trim|required|matches[passconf]|md5');
$this->form_validation->set_rules('passconf', 'Password Confirmation', 'trim|required');
$this->form_validation->set_rules('email', 'Email', 'trim|required|valid_email');
```

In the above example, we are "trimming" the fields, converting the password to MD5, and running the username through the "xss_clean" function, which removes malicious data.

Any native PHP function that accepts one parameter can be used as a rule, like htmlspecialchars, trim, MD5, etc.

Note: You will generally want to use the prepping functions **after** the validation rules so if there is an error, the original data will be shown in the form.

Re-populating the form

Thus far we have only been dealing with errors. It's time to repopulate the form field with the submitted data. CodeIgniter offers several helper functions that permit you to do this. The one you will use most commonly is:

```
set_value('field name')
```

Open your **myform.php** view file and update the **value** in each field using the **set_value()** function:

Don't forget to include each. field name in the set_value() functions!

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<?php echo validation_errors(); ?>

<?php echo form_open('form'); ?>

<h5>Username</h5>
<input type="text" name="username" value="<?php echo set_value('username'); ?>" size="50" />

<h5>Password</h5>
<input type="text" name="password" value="<?php echo set_value('password'); ?>" size="50" />

<h5>Password Confirm</h5>
<input type="text" name="passconf" value="<?php echo set_value('passconf'); ?>" size="50" />

<h5>Email Address</h5>
<input type="text" name="email" value="<?php echo set_value('email'); ?>" size="50" />

<div><input type="submit" value="Submit" /></div>

</form>

</body>
</html>
```

Now reload your page and submit the form so that it triggers an error. Your form fields should now be re-populated

Note: The Function Reference section below contains functions that permit you to re-populate <select> menus, radio buttons, and checkboxes.

Important Note: If you use an array as the name of a form field, you must supply it as an array to the function. Example:

```
<input type="text" name="colors[]" value="<?php echo set_value('colors[]'); ?>" size="50" />
```

For more info please see the Using Arrays as Field Names section below.

Callbacks: Your own Validation Functions

The validation system supports callbacks to your own validation functions. This permits you to extend the validation class to meet your needs. For example, if you need to run a database query to see if the user is choosing a unique username, you can create a callback function that does that. Let's create a example of this.

In your controller, change the "username" rule to this:

```
$this->form_validation->set_rules('username', 'Username', 'callback_username_check');
```

Then add a new function called **username_check** to your controller. Here's how your controller should now look:

```
<?php

class Form extends Controller {

	function index()
	{
		$this->load->helper(array('form', 'url'));

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

		$this->form_validation->set_rules('username', 'Username', 'callback_username_check');
		$this->form_validation->set_rules('password', 'Password', 'required');
		$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
		$this->form_validation->set_rules('email', 'Email', 'required');

		if ($this->form_validation->run() == FALSE)
		{
			$this->load->view('myform');
		}
		else
		{
			$this->load->view('formsuccess');
		}
	}

	function username_check($str)
	{
		if ($str == 'test')
		{
			$this->form_validation->set_message('username_check', 'The %s field can not be the
word "test"');
			return FALSE;
		}
		else
		{
			return TRUE;
		}
	}

}
?>
```

Reload your form and submit it with the word "test" as the username. You can see that the form field data was passed to your callback function for you to process.

To invoke a callback just put the function name in a rule, with "callback_" as the rule prefix.

You can also process the form data that is passed to your callback and return it. If your callback returns anything other than a boolean TRUE/FALSE it is assumed that the data is your newly processed form data.

Setting Error Messages

All of the native error messages are located in the following language file: **language/english/form_validation_lang.php**

To set your own custom message you can either edit that file, or use the following function:

```
$this->form_validation->set_message('rule', 'Error Message');
```

Where **rule** corresponds to the name of a particular rule, and **Error Message** is the text you would like displayed.

If you include **%s** in your error string, it will be replaced with the "human" name you used for your field when you set your rules.

In the "callback" example above, the error message was set by passing the name of the function:

```
$this->form_validation->set_message('username_check')
```

You can also override any error message found in the language file. For example, to change the message for the "required" rule you will do this:

```
$this->form_validation->set_message('required', 'Your custom message here');
```

Translating Field Names

If you would like to store the "human" name you passed to the **set_rules()** function in a language file, and therefore make the name able to be translated, here's how:

First, prefix your "human" name with **lang:**, as in this example:

```
$this->form_validation->set_rules('first_name', 'lang:first_name', 'required');
```

Then, store the name in one of your language file arrays (without the prefix):

```
$lang['first_name'] = 'First Name';
```

Note: If you store your array item in a language file that is not loaded automatically by CI, you'll need to remember to load it in your controller using:

```
$this->lang->load('file_name');
```

See the Language Class page for more info regarding language files.

Changing the Error Delimiters

By default, the Form Validation class adds a paragraph tag (<p>) around each error message shown. You can either change these delimiters globally or individually.

1. **Changing delimiters Globally**

   To globally change the error delimiters, in your controller function, just after loading the Form

Validation class, add this:

```
$this->form_validation->set_error_delimiters('<div class="error">', '</div>');
```

In this example, we've switched to using div tags.

2. **Changing delimiters Individually**

   Each of the two error generating functions shown in this tutorial can be supplied their own delimiters as follows:

   ```
   <?php echo form_error('field name', '<div class="error">', '</div>'); ?>
   ```

   Or:

   ```
   <?php echo validation_errors('<div class="error">', '</div>'); ?>
   ```

Showing Errors Individually

If you prefer to show an error message next to each form field, rather than as a list, you can use the **form_error()** function.

Try it! Change your form so that it looks like this:

```
<h5>Username</h5>
<?php echo form_error('username'); ?>
<input type="text" name="username" value="<?php echo set_value('username'); ?>" size="50" />

<h5>Password</h5>
<?php echo form_error('password'); ?>
<input type="text" name="password" value="<?php echo set_value('password'); ?>" size="50" />

<h5>Password Confirm</h5>
<?php echo form_error('passconf'); ?>
<input type="text" name="passconf" value="<?php echo set_value('passconf'); ?>" size="50" />

<h5>Email Address</h5>
<?php echo form_error('email'); ?>
<input type="text" name="email" value="<?php echo set_value('email'); ?>" size="50" />
```

If there are no errors, nothing will be shown. If there is an error, the message will appear.

Important Note: If you use an array as the name of a form field, you must supply it as an array to the function. Example:

```
<?php echo form_error('options[size]'); ?>
<input type="text" name="options[size]" value="<?php echo set_value("options[size]"); ?>" size="50" />
```

For more info please see the Using Arrays as Field Names section below.

Saving Sets of Validation Rules to a Config File

A nice feature of the Form Validation class is that it permits you to store all your validation rules for your entire application in a config file. You can organize these rules into "groups". These groups can either be loaded automatically when a matching controller/function is called, or you can manually call each set as needed.

How to save your rules

To store your validation rules, simply create a file named **form_validation.php** in your **application/config/** folder. In that file you will place an array named **$config** with your rules. As shown earlier, the validation array will have this prototype:

```
$config = array(
          array(
              'field'   => 'username',
              'label'   => 'Username',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'password',
              'label'   => 'Password',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'passconf',
              'label'   => 'Password Confirmation',
              'rules'   => 'required'
            ),
          array(
              'field'   => 'email',
              'label'   => 'Email',
              'rules'   => 'required'
            )
        );
```

Your validation rule file will be loaded automatically and used when you call the run() function.

Please note that you MUST name your array $config.

Creating Sets of Rules

In order to organize your rules into "sets" requires that you place them into "sub arrays". Consider the following example, showing two sets of rules. We've arbitrarily called these two rules "signup" and "email". You can name your rules anything you want:

```
$config = array(
           'signup' => array(
                         array(
                              'field' => 'username',
                              'label' => 'Username',
                              'rules' => 'required'
                           ),
                         array(
                              'field' => 'password',
                              'label' => 'Password',
                              'rules' => 'required'
                          ),
                     array(
                              'field' => 'passconf',
                              'label' => 'PasswordConfirmation',
                              'rules' => 'required'
                          ),
                     array(
                              'field' => 'email',
                              'label' => 'Email',
                              'rules' => 'required'
                          )
                     ),
          'email' => array(
                     array(
                              'field' => 'emailaddress',
                              'label' => 'EmailAddress',
                              'rules' => 'required|valid_email'
                          ),
                     array(
                              'field' => 'name',
                              'label' => 'Name',
                              'rules' => 'required|alpha'
                          ),
                     array(
                              'field' => 'title',
                              'label' => 'Title',
                              'rules' => 'required'
                          ),
                     array(
                              'field' => 'message',
                              'label' => 'MessageBody',
                              'rules' => 'required'
                          )
                     )
        );
```

Calling a Specific Rule Group

In order to call a specific group you will pass its name to the **run()** function. For example, to call the **signup** rule you will do this:

```
if ($this->form_validation->run('signup') == FALSE)
{
  $this->load->view('myform');
}
else
{
  $this->load->view('formsuccess');
}
```

Associating a Controller Function with a Rule Group

An alternate (and more automatic) method of calling a rule group is to name it according to the controller class/function you intend to use it with. For example, let's say you have a controller named **Member** and a function named **signup**. Here's what your class might look like:

```
<?php

class Member extends Controller {
  function signup()
  {
    $this->load->library('form_validation');

    if ($this->form_validation->run() == FALSE)
    {
      $this->load->view('myform');
    }
    else
    {
      $this->load->view('formsuccess');
    }
  }
}
?>
```

In your validation config file, you will name your rule group **member/signup**:

```
$config = array(
       'member/signup' => array(
                        array(
                              'field' => 'username',
                              'label' => 'Username',
                              'rules' => 'required'
                           ),
                        array(
                              'field' => 'password',
                              'label' => 'Password',
                              'rules' => 'required'
                           ),
                        array(
                              'field' => 'passconf',
                              'label' => 'PasswordConfirmation',
                              'rules' => 'required'
                           ),
                        array(
                              'field' => 'email',
                              'label' => 'Email',
                              'rules' => 'required'
                           )
                        )
          );
```

When a rule group is named identically to a controller class/function it will be used automatically when the run() function is invoked from that class/function.

Using Arrays as Field Names

The Form Validation class supports the use of arrays as field names. Consider this example:

```
<input type="text" name="options[]" value="" size="50" />
```

If you do use an array as a field name, you must use the EXACT array name in the Helper Functions that require the field name, and as your Validation Rule field name.

For example, to set a rule for the above field you would use:

```
$this->form_validation->set_rules('options[]', 'Options', 'required');
```

Or, to show an error for the above field you would use:

```
<?php echo form_error('options[]'); ?>
```

Or to re-populate the field you would use:

```
<input type="text" name="options[]" value="<?php echo set_value('options[]'); ?>" size="50" />
```

You can use multidimensional arrays as field names as well. For example:

```
<input type="text" name="options[size]" value="" size="50" />
```

Or even:

```
<input type="text" name="sports[nba][basketball]" value="" size="50" />
```

As with our first example, you must use the exact array name in the helper functions:

```
<?php echo form_error('sports[nba][basketball]'); ?>
```

If you are using checkboxes (or other fields) that have multiple options, don't forget to leave an empty bracket after each option, so that all selections will be added to the POST array:

```
<input type="checkbox" name="options[]" value="red" />
<input type="checkbox" name="options[]" value="blue" />
<input type="checkbox" name="options[]" value="green" />
```

Or if you use a multidimensional array:

```
<input type="checkbox" name="options[color][]" value="red" />
<input type="checkbox" name="options[color][]" value="blue" />
<input type="checkbox" name="options[color][]" value="green" />
```

When you use a helper function you'll include the bracket as well:

```
<?php echo form_error('options[color][]'); ?>
```

Rule Reference

The following is a list of all the native rules that are available to use:

Rule	Parameter	Description	Example
required	No	Returns FALSE if the form element is empty.	

matches	Yes	Returns FALSE if the form element does not match the one in the parameter.	matches[form_item]
min_length	Yes	Returns FALSE if the form element is shorter then the parameter value.	min_length[6]
max_length	Yes	Returns FALSE if the form element is longer then the parameter value.	max_length[12]
exact_length	Yes	Returns FALSE if the form element is not exactly the parameter value.	exact_length[8]
alpha	No	Returns FALSE if the form element contains anything other than alphabetical characters.	
alpha_numeric	No	Returns FALSE if the form element contains anything other than alpha-numeric characters.	
alpha_dash	No	Returns FALSE if the form element contains anything other than alpha-numeric characters, underscores or dashes.	
numeric	No	Returns FALSE if the form element contains anything other than numeric characters.	
integer	No	Returns FALSE if the form element contains anything other than an integer.	
is_natural	No	Returns FALSE if the form element contains anything other than a natural number: 0, 1, 2, 3, etc.	
is_natural_no_zero	No	Returns FALSE if the form element contains anything other than a natural number, but not zero: 1, 2, 3, etc.	
valid_email	No	Returns FALSE if the form element does not contain a valid email address.	
valid_emails	No	Returns FALSE if any value provided in a comma separated list is not a valid email.	
valid_ip	No	Returns FALSE if the supplied IP is not valid.	
valid_base64	No	Returns FALSE if the supplied string contains anything other than valid Base64 characters.	

Note: These rules can also be called as discrete functions. For example:

```
$this->form_validation->required($string);
```

Note: You can also use any native PHP functions that permit one parameter.

Prepping Reference

The following is a list of all the prepping functions that are available to use:

Name	Parameter	Description
xss_clean	No	Runs the data through the XSS filtering function, described in the Input Class page.
prep_for_form	No	Converts special characters so that HTML data can be shown in a form field without breaking it.

prep_url	No	Adds "http://" to URLs if missing.
strip_image_tags	No	Strips the HTML from image tags leaving the raw URL.
encode_php_tags	No	Converts PHP tags to entities.

Note: You can also use any native PHP functions that permit one parameter, like **trim**, **htmlspecialchars**, **urldecode**, etc.

Function Reference

The following functions are intended for use in your controller functions.

$this->form_validation->set_rules();

Permits you to set validation rules, as described in the tutorial sections above:

- Setting Validation Rules
- Saving Groups of Validation Rules to a Config File

$this->form_validation->run();

Runs the validation routines. Returns boolean TRUE on success and FALSE on failure. You can optionally pass the name of the validation group via the function, as described in: Saving Groups of Validation Rules to a Config File.

$this->form_validation->set_message();

Permits you to set custom error messages. See Setting Error Messages above.

Helper Reference

The following helper functions are available for use in the view files containing your forms. Note that these are procedural functions, so they **do not** require you to prepend them with $this->form_validation.

form_error()

Shows an individual error message associated with the field name supplied to the function. Example:

```
<?php echo form_error('username'); ?>
```

The error delimiters can be optionally specified. See the Changing the Error Delimiters section above.

validation_errors()

Shows all error messages as a string: Example:

```
<?php echo validation_errors(); ?>
```

The error delimiters can be optionally specified. See the Changing the Error Delimiters section above.

set_value()

Permits you to set the value of an input form or textarea. You must supply the field name via the first parameter of the function. The second (optional) parameter allows you to set a default value for the form. Example:

```
<input type="text" name="quantity" value="<?php echo set_value('quantity', '0'); ?>" size="50" />
```

The above form will show "0" when loaded for the first time.

set_select()

If you use a **<select>** menu, this function permits you to display the menu item that was selected. The first parameter must contain the name of the select menu, the second parameter must contain the value of each item, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE).

Example:

```
<select name="myselect">
<option value="one" <?php echo set_select('myselect', 'one', TRUE); ?> >One</option>
<option value="two" <?php echo set_select('myselect', 'two'); ?> >Two</option>
<option value="three" <?php echo set_select('myselect', 'three'); ?> >Three</option>
</select>
```

set_checkbox()

Permits you to display a checkbox in the state it was submitted. The first parameter must contain the name of the checkbox, the second parameter must contain its value, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE). Example:

```
<input type="checkbox" name="mycheck[]" value="1" <?php echo set_checkbox('mycheck[]', '1'); ?> />
<input type="checkbox" name="mycheck[]" value="2" <?php echo set_checkbox('mycheck[]', '2'); ?> />
```

set_radio()

Permits you to display radio buttons in the state they were submitted. This function is identical to the **set_checkbox()** function above.

```
<input type="radio" name="myradio" value="1" <?php echo set_radio('myradio', '1', TRUE); ?> />
<input type="radio" name="myradio" value="2" <?php echo set_radio('myradio', '2'); ?> />
```

FTP Class

CodeIgniter's FTP Class permits files to be transfered to a remote server. Remote files can also be moved, renamed, and deleted. The FTP class also includes a "mirroring" function that permits an entire local directory to be recreated remotely via FTP.

Note: SFTP and SSL FTP protocols are not supported, only standard FTP.

Initializing the Class

Like most other classes in CodeIgniter, the FTP class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the FTP object will be available using: **$this->ftp**

Usage Examples

In this example a connection is opened to the FTP server, and a local file is read and uploaded in ASCII mode. The file permissions are set to 755. Note: Setting permissions requires PHP 5.

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

$config['hostname'] = 'ftp.example.com';
$config['username'] = 'your-username';
$config['password'] = 'your-password';
$config['debug'] = TRUE;

$this->ftp->connect($config);

$this->ftp->upload('/local/path/to/myfile.html', '/public_html/myfile.html', 'ascii', 0775);

$this->ftp->close();
```

In this example a list of files is retrieved from the server.

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

$config['hostname'] = 'ftp.example.com';
$config['username'] = 'your-username';
$config['password'] = 'your-password';
$config['debug'] = TRUE;

$this->ftp->connect($config);

$list = $this->ftp->list_files('/public_html/');

print_r($list);
```

```
$this->ftp->close();
```

In this example a local directory is mirrored on the server.

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

$config['hostname'] = 'ftp.example.com';
$config['username'] = 'your-username';
$config['password'] = 'your-password';
$config['debug'] = TRUE;

$this->ftp->connect($config);

$this->ftp->mirror('/path/to/myfolder/', '/public_html/myfolder/');

$this->ftp->close();
```

Function Reference

$this->ftp->connect()

Connects and logs into to the FTP server. Connection preferences are set by passing an array to the function, or you can store them in a config file.

Here is an example showing how you set preferences manually:

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

$config['hostname'] = 'ftp.example.com';
$config['username'] = 'your-username';
$config['password'] = 'your-password';
$config['port']     = 21;
$config['passive']  = FALSE;
$config['debug']    = TRUE;

$this->ftp->connect($config);
```

Setting FTP Preferences in a Config File

If you prefer you can store your FTP preferences in a config file. Simply create a new file called the **ftp.php**, add the **$config** array in that file. Then save the file at **config/ftp.php** and it will be used automatically.

Available connection options:

- **hostname** - the FTP hostname. Usually something like: **ftp.example.com**
- **username** - the FTP username.
- **password** - the FTP password.
- **port** - The port number. Set to **21** by default.
- **debug** - **TRUE/FALSE** (boolean). Whether to enable debugging to display error messages.
- **passive** - **TRUE/FALSE** (boolean). Whether to use passive mode. Passive is set automatically

by default.

$this->ftp->upload()

Uploads a file to your server. You must supply the local path and the remote path, and you can optionally set the mode and permissions. Example:

```
$this->ftp->upload('/local/path/to/myfile.html', '/public_html/myfile.html', 'ascii', 0775);
```

Mode options are: **ascii**, **binary**, and **auto** (the default). If **auto** is used it will base the mode on the file extension of the source file.

Permissions are available if you are running PHP 5 and can be passed as an **octal** value in the fourth parameter.

$this->ftp->rename()

Permits you to rename a file. Supply the source file name/path and the new file name/path.

```
// Renames green.html to blue.html
$this->ftp->rename('/public_html/foo/green.html', '/public_html/foo/blue.html');
```

$this->ftp->move()

Lets you move a file. Supply the source and destination paths:

```
// Moves blog.html from "joe" to "fred"
$this->ftp->move('/public_html/joe/blog.html', '/public_html/fred/blog.html');
```

Note: if the destination file name is different the file will be renamed.

$this->ftp->delete_file()

Lets you delete a file. Supply the source path with the file name.

```
$this->ftp->delete_file('/public_html/joe/blog.html');
```

$this->ftp->delete_dir()

Lets you delete a directory and everything it contains. Supply the source path to the directory with a trailing slash.

Important Be VERY careful with this function. It will recursively delete **everything** within the supplied path, including sub-folders and all files. Make absolutely sure your path is correct. Try using the **list_files()** function first to verify that your path is correct.

```
$this->ftp->delete_dir('/public_html/path/to/folder/');
```

$this->ftp->list_files()

Permits you to retrieve a list of files on your server returned as an **array**. You must supply the path to the desired directory.

```
$list = $this->ftp->list_files('/public_html/');

print_r($list);
```

$this->ftp->mirror()

Recursively reads a local folder and everything it contains (including sub-folders) and creates a mirror via FTP based on it. Whatever the directory structure of the original file path will be recreated on the server. You must supply a source path and a destination path:

```
$this->ftp->mirror('/path/to/myfolder/', '/public_html/myfolder/');
```

$this->ftp->mkdir()

Lets you create a directory on your server. Supply the path ending in the folder name you wish to create, with a trailing slash. Permissions can be set by passed an **octal** value in the second parameter (if you are running PHP 5).

```
// Creates a folder named "bar"
$this->ftp->mkdir('/public_html/foo/bar/', DIR_WRITE_MODE);
```

$this->ftp->chmod()

Permits you to set file permissions. Supply the path to the file or folder you wish to alter permissions on:

```
// Chmod "bar" to 777
$this->ftp->chmod('/public_html/foo/bar/', DIR_WRITE_MODE);
```

$this->ftp->close();

Closes the connection to your server. It's recommended that you use this when you are finished uploading.

HTML Table Class

The Table Class provides functions that enable you to auto-generate HTML tables from arrays or database result sets.

Initializing the Class

Like most other classes in CodeIgniter, the Table class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Table library object will be available using: **$this->table**

Examples

Here is an example showing how you can create a table from a multi-dimensional array. Note that the first array index will become the table heading (or you can set your own headings using the **set_heading()** function described in the function reference below).

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

$data = array(
        array('Name', 'Color', 'Size'),
        array('Fred', 'Blue', 'Small'),
        array('Mary', 'Red', 'Large'),
        array('John', 'Green', 'Medium')
        );

echo $this->table->generate($data);
```

Here is an example of a table created from a database query result. The table class will automatically generate the headings based on the table names (or you can set your own headings using the **set_heading()** function described in the function reference below).

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

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

echo $this->table->generate($query);
```

Here is an example showing how you might create a table using discrete parameters:

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

$this->table->set_heading('Name', 'Color', 'Size');

$this->table->add_row('Fred', 'Blue', 'Small');
$this->table->add_row('Mary', 'Red', 'Large');
```

```
$this->table->add_row('John', 'Green', 'Medium');

echo $this->table->generate();
```

Here is the same example, except instead of individual parameters, arrays are used:

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

$this->table->set_heading(array('Name', 'Color', 'Size'));

$this->table->add_row(array('Fred', 'Blue', 'Small'));
$this->table->add_row(array('Mary', 'Red', 'Large'));
$this->table->add_row(array('John', 'Green', 'Medium'));

echo $this->table->generate();
```

Changing the Look of Your Table

The Table Class permits you to set a table template with which you can specify the design of your layout. Here is the template prototype:

```
$tmpl = array (
            'table_open'          => '<table border="0" cellpadding="4" cellspacing="0">',

            'heading_row_start'   => '<tr>',
            'heading_row_end'     => '</tr>',
            'heading_cell_start'  => '<th>',
            'heading_cell_end'    => '</th>',

            'row_start'           => '<tr>',
            'row_end'             => '</tr>',
            'cell_start'          => '<td>',
            'cell_end'            => '</td>',

            'row_alt_start'       => '<tr>',
            'row_alt_end'         => '</tr>',
            'cell_alt_start'      => '<td>',
            'cell_alt_end'        => '</td>',

            'table_close'         => '</table>'
        );

$this->table->set_template($tmpl);
```

Note: You'll notice there are two sets of "row" blocks in the template. These permit you to create alternating row colors or design elements that alternate with each iteration of the row data.

You are NOT required to submit a complete template. If you only need to change parts of the layout you can simply submit those elements. In this example, only the table opening tag is being changed:

```
$tmpl = array ( 'table_open'  => '<table border="1" cellpadding="2" cellspacing="1" class="mytable">' );

$this->table->set_template($tmpl);
```

Function Reference

$this->table->generate()

Returns a string containing the generated table. Accepts an optional parameter which can be an array or a database result object.

$this->table->set_caption()

Permits you to add a caption to the table.

```
$this->table->set_caption('Colors');
```

$this->table->set_heading()

Permits you to set the table heading. You can submit an array or discrete params:

```
$this->table->set_heading('Name', 'Color', 'Size');
```

```
$this->table->set_heading(array('Name', 'Color', 'Size'));
```

$this->table->add_row()

Permits you to add a row to your table. You can submit an array or discrete params:

```
$this->table->add_row('Blue', 'Red', 'Green');
```

```
$this->table->add_row(array('Blue', 'Red', 'Green'));
```

$this->table->make_columns()

This function takes a one-dimensional array as input and creates a multi-dimensional array with a depth equal to the number of columns desired. This allows a single array with many elements to be displayed in a table that has a fixed column count. Consider this example:

```
$list = array('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten', 'eleven', 'twelve');

$new_list = $this->table->make_columns($list, 3);

$this->table->generate($new_list);
```

```
// Generates a table with this prototype

<table border="0" cellpadding="4" cellspacing="0">
<tr>
<td>one</td><td>two</td><td>three</td>
</tr><tr>
<td>four</td><td>five</td><td>six</td>
</tr><tr>
<td>seven</td><td>eight</td><td>nine</td>
</tr><tr>
<td>ten</td><td>eleven</td><td>twelve</td></tr>
</table>
```

$this->table->set_template()

Permits you to set your template. You can submit a full or partial template.

```
$tmpl = array ( 'table_open'  => '<table border="1" cellpadding="2" cellspacing="1" class="mytable">' );

$this->table->set_template($tmpl);
```

$this->table->set_empty()

Let's you set a default value for use in any table cells that are empty. You might, for example, set a non-breaking space:

```
$this->table->set_empty("&nbsp;");
```

$this->table->clear()

Lets you clear the table heading and row data. If you need to show multiple tables with different data you should to call this function after each table has been generated to empty the previous table information. Example:

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

$this->table->set_heading('Name', 'Color', 'Size');
$this->table->add_row('Fred', 'Blue', 'Small');
$this->table->add_row('Mary', 'Red', 'Large');
$this->table->add_row('John', 'Green', 'Medium');

echo $this->table->generate();

$this->table->clear();

$this->table->set_heading('Name', 'Day', 'Delivery');
$this->table->add_row('Fred', 'Wednesday', 'Express');
$this->table->add_row('Mary', 'Monday', 'Air');
$this->table->add_row('John', 'Saturday', 'Overnight');
```

```
echo $this->table->generate();
```

Image Manipulation Class

CodeIgniter's Image Manipulation class lets you perform the following actions:

- Image Resizing
- Thumbnail Creation
- Image Cropping
- Image Rotating
- Image Watermarking

All three major image libraries are supported: GD/GD2, NetPBM, and ImageMagick

Note: Watermarking is only available using the GD/GD2 library. In addition, even though other libraries are supported, GD is required in order for the script to calculate the image properties. The image processing, however, will be performed with the library you specify.

Initializing the Class

Like most other classes in CodeIgniter, the image class is initialized in your controller using the **$this->load->library** function:

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

Once the library is loaded it will be ready for use. The image library object you will use to call all functions is: **$this->image_lib**

Processing an Image

Regardless of the type of processing you would like to perform (resizing, cropping, rotation, or watermarking), the general process is identical. You will set some preferences corresponding to the action you intend to perform, then call one of four available processing functions. For example, to create an image thumbnail you'll do this:

```
$config['image_library'] = 'gd2';
$config['source_image'] = '/path/to/image/mypic.jpg';
$config['create_thumb'] = TRUE;
$config['maintain_ratio'] = TRUE;
$config['width'] = 75;
$config['height'] = 50;

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

$this->image_lib->resize();
```

The above code tells the **image_resize** function to look for an image called *mypic.jpg* located in the **source_image** folder, then create a thumbnail that is 75 X 50 pixels using the GD2 **image_library**. Since the **maintain_ratio** option is enabled, the thumb will be as close to the target **width** and **height** as possible while preserving the original aspect ratio. The thumbnail will be called *mypic_thumb.jpg*

Note: In order for the image class to be allowed to do any processing, the folder containing the image files must have write permissions.

Processing Functions

There are four available processing functions:

- $this->image_lib->resize()
- $this->image_lib->crop()
- $this->image_lib->rotate()
- $this->image_lib->watermark()
- $this->image_lib->clear()

These functions return boolean TRUE upon success and FALSE for failure. If they fail you can retrieve the error message using this function:

```
echo $this->image_lib->display_errors();
```

A good practice is use the processing function conditionally, showing an error upon failure, like this:

```
if ( ! $this->image_lib->resize())
{
    echo $this->image_lib->display_errors();
}
```

Note: You can optionally specify the HTML formatting to be applied to the errors, by submitting the opening/closing tags in the function, like this:

```
$this->image_lib->display_errors('<p>', '</p>');
```

Preferences

The preferences described below allow you to tailor the image processing to suit your needs.

Note that not all preferences are available for every function. For example, the x/y axis preferences are only available for image cropping. Likewise, the width and height preferences have no effect on cropping. The "availability" column indicates which functions support a given preference.

Availability Legend:

- **R** - Image Resizing
- **C** - Image Cropping
- **X** - Image Rotation

W - Image Watermarking

Preference	Default Value	Options	Description	Availability
image_library	GD2	GD, GD2, ImageMagick, NetPBM	Sets the image library to be used.	R, C, X, W
library_path	None	None	Sets the server path to your ImageMagick or NetPBM library. If you use either of those libraries you must supply the path.	R, C, X
source_image	None	None	Sets the source image name/path. The path must be a relative or absolute server path, not a URL.	R, C, S, W
dynamic_output	FALSE	TRUE/FALSE (boolean)	Determines whether the new image file should be written to disk or generated dynamically. Note: If you choose the dynamic setting, only one image can be shown at a time, and it can't be positioned on the page. It simply outputs the raw image dynamically to your browser, along with image headers.	R, C, X, W
quality	90%	1 - 100%	Sets the quality of the image. The higher the quality the larger the file size.	R, C, X, W
new_image	None	None	Sets the destination image name/path. You'll use this preference when creating an image copy. The path must be a relative or absolute server path, not a URL.	R, C, X, W
width	None	None	Sets the width you would like the image set to.	R, C
height	None	None	Sets the height you would like the image set to.	R, C
create_thumb	FALSE	TRUE/FALSE (boolean)	Tells the image processing function to create a thumb.	R
thumb_marker	_thumb	None	Specifies the thumbnail indicator. It will be inserted just before the file extension, so mypic.jpg would become mypic_thumb.jpg	R
maintain_ratio	TRUE	TRUE/FALSE (boolean)	Specifies whether to maintain the original aspect ratio when resizing or use hard values.	R, C
master_dim	auto	auto, width, height	Specifies what to use as the master axis when resizing or creating thumbs. For example, let's say you want to resize an image to 100 X 75 pixels. If the source image size does not allow perfect resizing to those dimensions, this setting determines which axis should be used as the hard value. "auto" sets the axis automatically based on whether the image is taller then wider, or vice versa.	R
rotation_angle	None	90, 180, 270, vrt, hor	Specifies the angle of rotation when rotating images. Note that PHP rotates counter-clockwise, so a 90 degree rotation to the right must be specified as 270.	X
x_axis	None	None	Sets the X coordinate in pixels for image cropping. For example, a setting of 30 will crop an image 30 pixels from the left.	C

y_axis	None	None	Sets the Y coordinate in pixels for image cropping. For example, a setting of 30 will crop an image 30 pixels from the top.	C

Setting preferences in a config file

If you prefer not to set preferences using the above method, you can instead put them into a config file. Simply create a new file called **image_lib.php**, add the **$config** array in that file. Then save the file in: **config/image_lib.php** and it will be used automatically. You will NOT need to use the **$this->image_lib->initialize** function if you save your preferences in a config file.

$this->image_lib->resize()

The image resizing function lets you resize the original image, create a copy (with or without resizing), or create a thumbnail image.

For practical purposes there is no difference between creating a copy and creating a thumbnail except a thumb will have the thumbnail marker as part of the name (ie, mypic_thumb.jpg).

All preferences listed in the table above are available for this function except these three: rotation_angle, x_axis, and y_axis.

Creating a Thumbnail

The resizing function will create a thumbnail file (and preserve the original) if you set this preference to TRUE:

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

This single preference determines whether a thumbnail is created or not.

Creating a Copy

The resizing function will create a copy of the image file (and preserve the original) if you set a path and/or a new filename using this preference:

```
$config['new_image'] = '/path/to/new_image.jpg';
```

Notes regarding this preference:

- If only the new image name is specified it will be placed in the same folder as the original
- If only the path is specified, the new image will be placed in the destination with the same name as the original.
- If both the path and image name are specified it will placed in its own destination and given the new name.

Resizing the Original Image

If neither of the two preferences listed above (create_thumb, and new_image) are used, the resizing function will instead target the original image for processing.

$this->image_lib->crop()

The cropping function works nearly identically to the resizing function except it requires that you set preferences for the X and Y axis (in pixels) specifying where to crop, like this:

```
$config['x_axis'] = '100';
$config['y_axis'] = '40';
```

All preferences listed in the table above are available for this function except these: rotation_angle, width, height, create_thumb, new_image.

Here's an example showing how you might crop an image:

```
$config['image_library'] = 'imagemagick';
$config['library_path'] = '/usr/X11R6/bin/';
$config['source_image'] = '/path/to/image/mypic.jpg';
$config['x_axis'] = '100';
$config['y_axis'] = '60';

$this->image_lib->initialize($config);

if ( ! $this->image_lib->crop())
{
    echo $this->image_lib->display_errors();
}
```

Note: Without a visual interface it is difficult to crop images, so this function is not very useful unless you intend to build such an interface. That's exactly what we did using for the photo gallery module in ExpressionEngine, the CMS we develop. We added a JavaScript UI that lets the cropping area be selected.

$this->image_lib->rotate()

The image rotation function requires that the angle of rotation be set via its preference:

```
$config['rotation_angle'] = '90';
```

There are 5 rotation options:

1. 90 - rotates counter-clockwise by 90 degrees.
2. 180 - rotates counter-clockwise by 180 degrees.
3. 270 - rotates counter-clockwise by 270 degrees.
4. hor - flips the image horizontally.
5. vrt - flips the image vertically.

Here's an example showing how you might rotate an image:

```
$config['image_library'] = 'netpbm';
$config['library_path'] = '/usr/bin/';
$config['source_image'] = '/path/to/image/mypic.jpg';
```

```
$config['rotation_angle'] = 'hor';

$this->image_lib->initialize($config);

if ( ! $this->image_lib->rotate())
{
   echo $this->image_lib->display_errors();
}
```

$this->image_lib->clear()

The clear function resets all of the values used when processing an image. You will want to call this if you are processing images in a loop.

```
$this->image_lib->clear();
```

Image Watermarking

The Watermarking feature requires the GD/GD2 library.

Two Types of Watermarking

There are two types of watermarking that you can use:

- **Text**: The watermark message will be generating using text, either with a True Type font that you specify, or using the native text output that the GD library supports. If you use the True Type version your GD installation must be compiled with True Type support (most are, but not all).
- **Overlay**: The watermark message will be generated by overlaying an image (usually a transparent PNG or GIF) containing your watermark over the source image.

Watermarking an Image

Just as with the other functions (resizing, cropping, and rotating) the general process for watermarking involves setting the preferences corresponding to the action you intend to perform, then calling the watermark function. Here is an example:

```
$config['source_image'] = '/path/to/image/mypic.jpg';
$config['wm_text'] = 'Copyright 2006 - John Doe';
$config['wm_type'] = 'text';
$config['wm_font_path'] = './system/fonts/texb.ttf';
$config['wm_font_size'] = '16';
$config['wm_font_color'] = 'ffffff';
$config['wm_vrt_alignment'] = 'bottom';
$config['wm_hor_alignment'] = 'center';
$config['wm_padding'] = '20';

$this->image_lib->initialize($config);
```

```
$this->image_lib->watermark();
```

The above example will use a 16 pixel True Type font to create the text "Copyright 2006 - John Doe". The watermark will be positioned at the bottom/center of the image, 20 pixels from the bottom of the image.

Note: In order for the image class to be allowed to do any processing, the image file must have "write" file permissions. For example, 777.

Watermarking Preferences

This table shown the preferences that are available for both types of watermarking (text or overlay)

Preference	Default Value	Options	Description
wm_type	text	text, overlay	Sets the type of watermarking that should be used.
source_image	None	None	Sets the source image name/path. The path must be a relative or absolute server path, not a URL.
dynamic_output	FALSE	TRUE/FALSE (boolean)	Determines whether the new image file should be written to disk or generated dynamically. Note: If you choose the dynamic setting, only one image can be shown at a time, and it can't be positioned on the page. It simply outputs the raw image dynamically to your browser, along with image headers.
quality	90%	1 - 100%	Sets the quality of the image. The higher the quality the larger the file size.
padding	None	A number	The amount of padding, set in pixels, that will be applied to the watermark to set it away from the edge of your images.
wm_vrt_alignment	bottom	top, middle, bottom	Sets the vertical alignment for the watermark image.
wm_hor_alignment	center	left, center, right	Sets the horizontal alignment for the watermark image.
wm_hor_offset	None	None	You may specify a horizontal offset (in pixels) to apply to the watermark position. The offset normally moves the watermark to the right, except if you have your alignment set to "right" then your offset value will move the watermark toward the left of the image.
wm_vrt_offset	None	None	You may specify a vertical offset (in pixels) to apply to the watermark position. The offset normally moves the watermark down, except if you have your alignment set to "bottom" then your offset value will move the watermark toward the top of the image.

Text Preferences

This table shown the preferences that are available for the text type of watermarking.

Preference	Default Value	Options	Description

wm_text	None	None	The text you would like shown as the watermark. Typically this will be a copyright notice.
wm_font_path	None	None	The server path to the True Type Font you would like to use. If you do not use this option, the native GD font will be used.
wm_font_size	16	None	The size of the text. Note: If you are not using the True Type option above, the number is set using a range of 1 - 5. Otherwise, you can use any valid pixel size for the font you're using.
wm_font_color	ffffff	None	The font color, specified in hex. Note, you must use the full 6 character hex value (ie, 993300), rather than the three character abbreviated version (ie fff).
wm_shadow_color	None	None	The color of the drop shadow, specified in hex. If you leave this blank a drop shadow will not be used. Note, you must use the full 6 character hex value (ie, 993300), rather than the three character abbreviated version (ie fff).
wm_shadow_distance	3	None	The distance (in pixels) from the font that the drop shadow should appear.

Overlay Preferences

This table shown the preferences that are available for the overlay type of watermarking.

Preference	Default Value	Options	Description
wm_overlay_path	None	None	The server path to the image you wish to use as your watermark. Required only if you are using the overlay method.
wm_opacity	50	1 - 100	Image opacity. You may specify the opacity (i.e. transparency) of your watermark image. This allows the watermark to be faint and not completely obscure the details from the original image behind it. A 50% opacity is typical.
wm_x_transp	4	A number	If your watermark image is a PNG or GIF image, you may specify a color on the image to be "transparent". This setting (along with the next) will allow you to specify that color. This works by specifying the "X" and "Y" coordinate pixel (measured from the upper left) within the image that corresponds to a pixel representative of the color you want to be transparent.
wm_y_transp	4	A number	Along with the previous setting, this allows you to specify the coordinate to a pixel representative of the color you want to be transparent.

Input Class

The Input Class serves two purposes:

1. It pre-processes global input data for security.
2. It provides some helper functions for fetching input data and pre-processing it.

Note: This class is initialized automatically by the system so there is no need to do it manually.

Security Filtering

The security filtering function is called automatically when a new controller is invoked. It does the following:

- Destroys the global GET array. Since CodeIgniter does not utilize GET strings, there is no reason to allow it.
- Destroys all global variables in the event register_globals is turned on.
- Filters the POST/COOKIE array keys, permitting only alpha-numeric (and a few other) characters.
- Provides XSS (Cross-site Scripting Hacks) filtering. This can be enabled globally, or upon request.
- Standardizes newline characters to \n

XSS Filtering

CodeIgniter comes with a Cross Site Scripting Hack prevention filter which can either run automatically to filter all POST and COOKIE data that is encountered, or you can run it on a per item basis. By default it does **not** run globally since it requires a bit of processing overhead, and since you may not need it in all cases.

The XSS filter looks for commonly used techniques to trigger Javascript or other types of code that attempt to hijack cookies or do other malicious things. If anything disallowed is encountered it is rendered safe by converting the data to character entities.

Note: This function should only be used to deal with data upon submission. It's not something that should be used for general runtime processing since it requires a fair amount of processing overhead.

To filter data through the XSS filter use this function:

$this->input->xss_clean()

Here is an usage example:

```
$data = $this->input->xss_clean($data);
```

If you want the filter to run automatically every time it encounters POST or COOKIE data you can enable it by opening your **application/config/config.php** file and setting this:

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

Note: If you use the form validation class, it gives you the option of XSS filtering as well.

An optional second parameter, **is_image**, allows this function to be used to test images for potential XSS attacks, useful for file upload security. When this second parameter is set to **TRUE**, instead of returning an altered string, the function returns TRUE if the image is safe, and FALSE if it contained potentially malicious information that a browser may attempt to execute.

```
if ($this->input->xss_clean($file, TRUE) === FALSE)
{
    // file failed the XSS test
}
```

Using POST, COOKIE, or SERVER Data

CodeIgniter comes with three helper functions that let you fetch POST, COOKIE or SERVER items. The main advantage of using the provided functions rather than fetching an item directly ($_POST['something']) is that the functions will check to see if the item is set and return false (boolean) if not. This lets you conveniently use data without having to test whether an item exists first. In other words, normally you might do something like this:

```
if ( ! isset($_POST['something']))
{
    $something = FALSE;
}
else
{
    $something = $_POST['something'];
}
```

With CodeIgniter's built in functions you can simply do this:

```
$something = $this->input->post('something');
```

The three functions are:

- $this->input->post()
- $this->input->cookie()
- $this->input->server()

$this->input->post()

The first parameter will contain the name of the POST item you are looking for:

```
$this->input->post('some_data');
```

The function returns FALSE (boolean) if the item you are attempting to retrieve does not exist.

The second optional parameter lets you run the data through the XSS filter. It's enabled by setting the second parameter to boolean TRUE;

```
$this->input->post('some_data', TRUE);
```

$this->input->get()

This function is identical to the post function, only it fetches get data:

```
$this->input->get('some_data', TRUE);
```

$this->input->get_post()

This function will search through both the post and get streams for data, looking first in post, and then in get:

```
$this->input->get_post('some_data', TRUE);
```

$this->input->cookie()

This function is identical to the post function, only it fetches cookie data:

```
$this->input->cookie('some_data', TRUE);
```

$this->input->server()

This function is identical to the above functions, only it fetches server data:

```
$this->input->server('some_data');
```

$this->input->ip_address()

Returns the IP address for the current user. If the IP address is not valid, the function will return an IP of: 0.0.0.0

```
echo $this->input->ip_address();
```

$this->input->valid_ip($ip)

Takes an IP address as input and returns TRUE or FALSE (boolean) if it is valid or not. Note: The $this->input->ip_address() function above validates the IP automatically.

```
if ( ! $this->input->valid_ip($ip))
{
    echo 'Not Valid';
}
else
{
    echo 'Valid';
}
```

$this->input->user_agent()

Returns the user agent (web browser) being used by the current user. Returns FALSE if it's not available.

```
echo $this->input->user_agent();
```

Loader Class

Loader, as the name suggests, is used to load elements. These elements can be libraries (classes) View files, Helpers, Plugins, or your own files.

Note: This class is initialized automatically by the system so there is no need to do it manually.

The following functions are available in this class:

$this->load->library('class_name', $config, 'object name')

This function is used to load core classes. Where **class_name** is the name of the class you want to load. Note: We use the terms "class" and "library" interchangeably.

For example, if you would like to send email with CodeIgniter, the first step is to load the email class within your controller:

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

Once loaded, the library will be ready for use, using **$this->email->*some_function*()**.

Library files can be stored in subdirectories within the main "libraries" folder, or within your personal **application/libraries** folder. To load a file located in a subdirectory, simply include the path, relative to the "libraries" folder. For example, if you have file located at:

```
libraries/flavors/chocolate.php
```

You will load it using:

```
$this->load->library('flavors/chocolate');
```

You may nest the file in as many subdirectories as you want.

Setting options

The second (optional) parameter allows you to optionally pass configuration setting. You will typically pass these as an array:

```
$config = array (
              'mailtype' => 'html',
              'charset'  => 'utf-8,
              'priority' => '1'
            );

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

Config options can usually also be set via a config file. Each library is explained in detail in its own page, so please read the information regarding each one you would like to use.

Assigning a Library to a different object name

If the third (optional) parameter is blank, the library will usually be assigned to an object with the same name as the library. For example, if the library is named **Session**, it will be assigned to a variable named **$this->session**.

If you prefer to set your own class names you can pass its value to the third parameter:

```
$this->load->library('session', '', 'my_session');

// Session class is now accessed using:

$this->my_session
```

$this->load->view('file_name', $data, true/false)

This function is used to load your View files. If you haven't read the Views section of the user guide it is recommended that you do since it shows you how this function is typically used.

The first parameter is required. It is the name of the view file you would like to load. Note: The .php file extension does not need to be specified unless you use something other than **.php**.

The second **optional** parameter can take an associative array or an object as input, which it runs through the PHP extract function to convert to variables that can be used in your view files. Again, read the Views page to learn how this might be useful.

The third **optional** parameter lets you change the behavior of the function so that it returns data as a string rather than sending it to your browser. This can be useful if you want to process the data in some way. If you set the parameter to **true** (boolean) it will return data. The default behavior is **false**, which sends it to your browser. Remember to assign it to a variable if you want the data returned:

```
$string = $this->load->view('myfile', '', true);
```

$this->load->model('Model_name');

```
$this->load->model('Model_name');
```

If your model is located in a sub-folder, include the relative path from your models folder. For example, if you have a model located at application/models/blog/queries.php you'll load it using:

```
$this->load->model('blog/queries');
```

If you would like your model assigned to a different object name you can specify it via the second parameter of the loading function:

```
$this->load->model('Model_name', 'fubar');

$this->fubar->function();
```

$this->load->database('options', true/false)

This function lets you load the database class. The two parameters are **optional**. Please see the database section for more info.

$this->load->scaffolding('table_name')

This function lets you enable scaffolding. Please see the scaffolding section for more info.

$this->load->vars($array)

This function takes an associative array as input and generates variables using the PHP extract function. This function produces the same result as using the second parameter of the **$this->load->view()** function above. The reason you might want to use this function independently is if you would like to set some global variables in the constructor of your controller and have them become available in any view file loaded from any function. You can have multiple calls to this function. The data get cached and merged into one array for conversion to variables.

$this->load->helper('file_name')

This function loads helper files, where **file_name** is the name of the file, without the **_helper.php** extension.

$this->load->plugin('file_name')

This function loads plugins files, where **file_name** is the name of the file, without the **_plugin.php** extension.

$this->load->file('filepath/filename', true/false)

This is a generic file loading function. Supply the filepath and name in the first parameter and it will open and read the file. By default the data is sent to your browser, just like a View file, but if you set the second parameter to **true** (boolean) it will instead return the data as a string.

$this->load->lang('file_name')

This function is an alias of the language loading function: $this->lang->load()

$this->load->config('file_name')

This function is an alias of the config file loading function: $this->config->load()

Language Class

The Language Class provides functions to retrieve language files and lines of text for purposes of internationalization.

In your CodeIgniter system folder you'll find one called **language** containing sets of language files. You can create your own language files as needed in order to display error and other messages in other languages.

Language files are typically stored in your **system/language** directory. Alternately you can create a folder called **language** inside your **application** folder and store them there. CodeIgniter will look first in your **system/application/language** directory. If the directory does not exist or the specified language is not located there CI will instead look in your global **system/language** folder.

Note: Each language should be stored in its own folder. For example, the English files are located at: **system/language/english**

Creating Language Files

Language files must be named with **_lang.php** as the file extension. For example, let's say you want to create a file containing error messages. You might name it: **error_lang.php**

Within the file you will assign each line of text to an array called **$lang** with this prototype:

```
$lang['language_key'] = "The actual message to be shown";
```

Note: It's a good practice to use a common prefix for all messages in a given file to avoid collisions with similarly named items in other files. For example, if you are creating error messages you might prefix them with **error_**

```
$lang['error_email_missing'] = "You must submit an email address";
$lang['error_url_missing'] = "You must submit a URL";
$lang['error_username_missing'] = "You must submit a username";
```

Loading A Language File

In order to fetch a line from a particular file you must load the file first. Loading a language file is done with the following code:

```
$this->lang->load('filename', 'language');
```

Where **filename** is the name of the file you wish to load (without the file extension), and **language** is the language set containing it (ie, english). If the second parameter is missing, the default language set in your **application/config/config.php** file will be used.

Fetching a Line of Text

Once your desired language file is loaded you can access any line of text using this function:

```
$this->lang->line('language_key');
```

Where **language_key** is the array key corresponding to the line you wish to show.

Note: This function simply returns the line. It does not echo it for you.

Using language lines as form labels

This feature has been deprecated from the language library and moved to the **lang()** function of the Language helper.

Auto-loading Languages

If you find that you need a particular language globally throughout your application, you can tell CodeIgniter to auto-load it during system initialization. This is done by opening the application/config/autoload.php file and adding the language(s) to the autoload array.

Output Class

The Output class is a small class with one main function: To send the finalized web page to the requesting browser. It is also responsible for caching your web pages, if you use that feature.

Note: This class is initialized automatically by the system so there is no need to do it manually.

Under normal circumstances you won't even notice the Output class since it works transparently without your intervention. For example, when you use the Loader class to load a view file, it's automatically passed to the Output class, which will be called automatically by CodeIgniter at the end of system execution. It is possible, however, for you to manually intervene with the output if you need to, using either of the two following functions:

$this->output->set_output();

Permits you to manually set the final output string. Usage example:

```
$this->output->set_output($data);
```

Important: If you do set your output manually, it must be the last thing done in the function you call it from. For example, if you build a page in one of your controller functions, don't set the output until the end.

$this->output->get_output();

Permits you to manually retrieve any output that has been sent for storage in the output class. Usage example:

```
$string = $this->output->get_output();
```

Note that data will only be retrievable from this function if it has been previously sent to the output class by one of the CodeIgniter functions like **$this->load->view()**.

$this->output->set_header();

Permits you to manually set server headers, which the output class will send for you when outputting the final rendered display. Example:

```
$this->output->set_header("HTTP/1.0 200 OK");
$this->output->set_header("HTTP/1.1 200 OK");
$this->output->set_header('Last-Modified: '.gmdate('D, d M Y H:i:s', $last_update).' GMT');
$this->output->set_header("Cache-Control: no-store, no-cache, must-revalidate");
$this->output->set_header("Cache-Control: post-check=0, pre-check=0");
$this->output->set_header("Pragma: no-cache");
```

$this->output->set_status_header(code, 'text');

Permits you to manually set a server status header. Example:

```
$this->output->set_status_header('401');
// Sets the header as: Unauthorized
```

See here for a full list of headers.

$this->output->enable_profiler();

Permits you to enable/disable the Profiler, which will display benchmark and other data at the bottom of your pages for debugging and optimization purposes.

To enable the profiler place the following function anywhere within your Controller functions:

```
$this->output->enable_profiler(TRUE);
```

When enabled a report will be generated and inserted at the bottom of your pages.

To disable the profiler you will use:

```
$this->output->enable_profiler(FALSE);
```

$this->output->cache();

The CodeIgniter output library also controls caching. For more information, please see the caching documentation.

Pagination Class

CodeIgniter's Pagination class is very easy to use, and it is 100% customizable, either dynamically or via stored preferences.

If you are not familiar with the term "pagination", it refers to links that allows you to navigate from page to page, like this:

```
« First  < 1 2 3 4 5 >  Last »
```

Example

Here is a simple example showing how to create pagination in one of your controller functions:

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

$config['base_url'] = 'http://example.com/index.php/test/page/';
$config['total_rows'] = '200';
$config['per_page'] = '20';

$this->pagination->initialize($config);

echo $this->pagination->create_links();
```

Notes:

The **$config** array contains your configuration variables. It is passed to the **$this->pagination->initialize** function as shown above. Although there are some twenty items you can configure, at minimum you need the three shown. Here is a description of what those items represent:

- **base_url** This is the full URL to the controller class/function containing your pagination. In the example above, it is pointing to a controller called "Test" and a function called "page". Keep in mind that you can re-route your URI if you need a different structure.
- **total_rows** This number represents the total rows in the result set you are creating pagination for. Typically this number will be the total rows that your database query returned.
- **per_page** The number of items you intend to show per page. In the above example, you would be showing 20 items per page.

The **create_links()** function returns an empty string when there is no pagination to show.

Setting preferences in a config file

If you prefer not to set preferences using the above method, you can instead put them into a config file. Simply create a new file called **pagination.php**, add the **$config** array in that file. Then save the file in: **config/pagination.php** and it will be used automatically. You will NOT need to use the **$this->pagination->initialize** function if you save your preferences in a config file.

Customizing the Pagination

The following is a list of all the preferences you can pass to the initialization function to tailor the display.

$config['uri_segment'] = 3;

The pagination function automatically determines which segment of your URI contains the page number. If you need something different you can specify it.

$config['num_links'] = 2;

The number of "digit" links you would like before and after the selected page number. For example, the number 2 will place two digits on either side, as in the example links at the very top of this page.

$config['page_query_string'] = TRUE

By default, the pagination library assume you are using URI Segments, and constructs your links something like

```
http://example.com/index.php/test/page/20
```

If you have $config['enable_query_strings'] set to TRUE your links will automatically be re-written using Query Strings. This option can also be explictly set. Using $config['page_query_string'] set to TRUE, the pagination link will become.

```
http://example.com/index.php?c=test&m=page&per_page=20
```

Note that "per_page" is the default query string passed, however can be configured using $config['query_string_segment'] = 'your_string'

Adding Enclosing Markup

If you would like to surround the entire pagination with some markup you can do it with these two prefs:

$config['full_tag_open'] = '<p>';

The opening tag placed on the left side of the entire result.

$config['full_tag_close'] = '</p>';

The closing tag placed on the right side of the entire result.

Customizing the First Link

$config['first_link'] = 'First';

The text you would like shown in the "first" link on the left.

$config['first_tag_open'] = '<div>';

The opening tag for the "first" link.

$config['first_tag_close'] = '</div>';

The closing tag for the "first" link.

Customizing the Last Link

$config['last_link'] = 'Last';

The text you would like shown in the "last" link on the right.

$config['last_tag_open'] = '<div>';

The opening tag for the "last" link.

$config['last_tag_close'] = '</div>';

The closing tag for the "last" link.

Customizing the "Next" Link

$config['next_link'] = '&gt;';

The text you would like shown in the "next" page link.

$config['next_tag_open'] = '<div>';

The opening tag for the "next" link.

$config['next_tag_close'] = '</div>';

The closing tag for the "next" link.

Customizing the "Previous" Link

$config['prev_link'] = '&lt;';

The text you would like shown in the "previous" page link.

$config['prev_tag_open'] = '<div>';

The opening tag for the "previous" link.

$config['prev_tag_close'] = '</div>';

The closing tag for the "previous" link.

Customizing the "Current Page" Link

$config['cur_tag_open'] = '<b>';

The opening tag for the "current" link.

$config['cur_tag_close'] = '</b>';

The closing tag for the "current" link.

Customizing the "Digit" Link

$config['num_tag_open'] = '<div>';

The opening tag for the "digit" link.

$config['num_tag_close'] = '</div>';

The closing tag for the "digit" link.

Session Class

The Session class permits you maintain a user's "state" and track their activity while they browse your site. The Session class stores session information for each user as serialized (and optionally encrypted) data in a cookie. It can also store the session data in a database table for added security, as this permits the session ID in the user's cookie to be matched against the stored session ID. By default only the cookie is saved. If you choose to use the database option you'll need to create the session table as indicated below.

Note: The Session class does **not** utilize native PHP sessions. It generates its own session data, offering more flexibility for developers.

Initializing a Session

Sessions will typically run globally with each page load, so the session class must either be initialized in your controller constructors, or it can be auto-loaded by the system. For the most part the session class will run unattended in the background, so simply initializing the class will cause it to read, create, and update sessions.

To initialize the Session class manually in your controller constructor, use the **$this->load->library** function:

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

Once loaded, the Sessions library object will be available using: **$this->session**

How do Sessions work?

When a page is loaded, the session class will check to see if valid session data exists in the user's session cookie. If sessions data does **not** exist (or if it has expired) a new session will be created and saved in the cookie. If a session does exist, its information will be updated and the cookie will be updated. With each update, the session_id will be regenerated.

It's important for you to understand that once initialized, the Session class runs automatically. There is nothing you need to do to cause the above behavior to happen. You can, as you'll see below, work with session data or even add your own data to a user's session, but the process of reading, writing, and updating a session is automatic.

What is Session Data?

A *session*, as far as CodeIgniter is concerned, is simply an array containing the following information:

- The user's unique Session ID (this is a statistically random string with very strong entropy, hashed with MD5 for portability, and regenerated (by default) every five minutes)
- The user's IP Address
- The user's User Agent data (the first 50 characters of the browser data string)
- The "last activity" time stamp.

The above data is stored in a cookie as a serialized array with this prototype:

```
[array]
(
    'session_id'    => random hash,
    'ip_address'    => 'string - user IP address',
    'user_agent'    => 'string - user agent data',
    'last_activity' => timestamp
)
```

If you have the encryption option enabled, the serialized array will be encrypted before being stored in the cookie, making the data highly secure and impervious to being read or altered by someone. More info regarding encryption can be found here, although the Session class will take care of initializing and encrypting the data automatically.

Note: Session cookies are only updated every five minutes by default to reduce processor load. If you repeatedly reload a page you'll notice that the "last activity" time only updates if five minutes or more has passed since the last time the cookie was written. This time is configurable by changing the $config['time_to_update'] line in your system/config/config.php file.

Retrieving Session Data

Any piece of information from the session array is available using the following function:

```
$this->session->userdata('item');
```

Where **item** is the array index corresponding to the item you wish to fetch. For example, to fetch the session ID you will do this:

```
$session_id = $this->session->userdata('session_id');
```

Note: The function returns FALSE (boolean) if the item you are trying to access does not exist.

Adding Custom Session Data

A useful aspect of the session array is that you can add your own data to it and it will be stored in the user's cookie. Why would you want to do this? Here's one example:

Let's say a particular user logs into your site. Once authenticated, you could add their username and email address to the session cookie, making that data globally available to you without having to run a database query when you need it.

To add your data to the session array involves passing an array containing your new data to this function:

```
$this->session->set_userdata($array);
```

Where **$array** is an associative array containing your new data. Here's an example:

```
$newdata = array(
               'username'  => 'johndoe',
               'email'     => 'johndoe@some-site.com',
               'logged_in' => TRUE
           );

$this->session->set_userdata($newdata);
```

If you want to add userdata one value at a time, set_userdata() also supports this syntax.

```
$this->session->set_userdata('some_name', 'some_value');
```

Note: Cookies can only hold 4KB of data, so be careful not to exceed the capacity. The encryption process in particular produces a longer data string than the original so keep careful track of how much data you are storing.

Removing Session Data

Just as set_userdata() can be used to add information into a session, unset_userdata() can be used to remove it, by passing the session key. For example, if you wanted to remove 'some_name' from your session information:

```
$this->session->unset_userdata('some_name');
```

This function can also be passed an associative array of items to unset.

```
$array_items = array('username' => '', 'email' => '');

$this->session->unset_userdata($array_items);
```

Flashdata

CodeIgniter supports "flashdata", or session data that will only be available for the next server request, and are then automatically cleared. These can be very useful, and are typically used for informational or status messages (for example: "record 2 deleted").

Note: Flash variables are prefaced with "flash_" so avoid this prefix in your own session names.

To add flashdata:

```
$this->session->set_flashdata('item', 'value');
```

You can also pass an array to set_flashdata(), in the same manner as set_userdata().

To read a flashdata variable:

```
$this->session->flashdata('item');
```

If you find that you need to preserve a flashdata variable through an additional request, you can do so using the keep_flashdata() function.

```
$this->session->keep_flashdata('item');
```

Saving Session Data to a Database

While the session data array stored in the user's cookie contains a Session ID, unless you store session data in a database there is no way to validate it. For some applications that require little or no security, session ID validation may not be needed, but if your application requires security, validation is mandatory.

When session data is available in a database, every time a valid session is found in the user's cookie, a database query is performed to match it. If the session ID does not match, the session is destroyed. Session IDs can never be updated, they can only be generated when a new session is created.

In order to store sessions, you must first create a database table for this purpose. Here is the basic prototype (for MySQL) required by the session class:

```
CREATE TABLE IF NOT EXISTS  `ci_sessions` (
session_id varchar(40) DEFAULT '0' NOT NULL,
ip_address varchar(16) DEFAULT '0' NOT NULL,
user_agent varchar(50) NOT NULL,
last_activity int(10) unsigned DEFAULT 0 NOT NULL,
user_data text NOT NULL,
PRIMARY KEY (session_id)
);
```

Note: By default the table is called **ci_sessions**, but you can name it anything you want as long as you update the **application/config/config.php** file so that it contains the name you have chosen. Once you have created your database table you can enable the database option in your config.php file as follows:

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

Once enabled, the Session class will store session data in the DB.

Make sure you've specified the table name in your config file as well:

```
$config['sess_table_name'] = 'ci_sessions';
```

Note: The Session class has built-in garbage collection which clears out expired sessions so you do not need to write your own routine to do it.

Destroying a Session

To clear the current session:

```
$this->session->sess_destroy();
```

Note: This function should be the last one called, and even flash variables will no longer be available. If you only want some items destroyed and not all, use **unset_userdata()**.

Session Preferences

You'll find the following Session related preferences in your **application/config/config.php** file:

Preference	Default	Options	Description
sess_cookie_name	ci_session	None	The name you want the session cookie saved as.
sess_expiration	7200	None	The number of seconds you would like the session to last. The default value is 2 hours (7200 seconds). If you would like a non-expiring session set the value to zero: 0
sess_encrypt_cookie	FALSE	TRUE/FALSE (boolean)	Whether to encrypt the session data.
sess_use_database	FALSE	TRUE/FALSE (boolean)	Whether to save the session data to a database. You must create the table before enabling this option.
sess_table_name	ci_sessions	Any valid SQL table name	The name of the session database table.
sess_time_to_update	300	Time in seconds	This options controls how often the session class will regenerate itself and create a new session id.
sess_match_ip	FALSE	TRUE/FALSE (boolean)	Whether to match the user's IP address when reading the session data. Note that some ISPs dynamically changes the IP, so if you want a non-expiring session you will likely set this to FALSE.
sess_match_useragent	TRUE	TRUE/FALSE (boolean)	Whether to match the User Agent when reading the session data.

Trackback Class

The Trackback Class provides functions that enable you to send and receive Trackback data.

If you are not familiar with Trackbacks you'll find more information here.

Initializing the Class

Like most other classes in CodeIgniter, the Trackback class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Trackback library object will be available using: **$this->trackback**

Sending Trackbacks

A Trackback can be sent from any of your controller functions using code similar to this example:

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

$tb_data = array(
            'ping_url'  => 'http://example.com/trackback/456',
            'url'       => 'http://www.my-example.com/blog/entry/123',
            'title'     => 'The Title of My Entry',
            'excerpt'   => 'The entry content.',
            'blog_name' => 'My Blog Name',
            'charset'   => 'utf-8'
            );

if ( ! $this->trackback->send($tb_data))
{
    echo $this->trackback->display_errors();
}
else
{
    echo 'Trackback was sent!';
}
```

Description of array data:

- **ping_url** - The URL of the site you are sending the Trackback to. You can send Trackbacks to multiple URLs by separating each URL with a comma.
- **url** - The URL to YOUR site where the weblog entry can be seen.
- **title** - The title of your weblog entry.
- **excerpt** - The content of your weblog entry. Note: the Trackback class will automatically send only the first 500 characters of your entry. It will also strip all HTML.
- **blog_name** - The name of your weblog.
- **charset** - The character encoding your weblog is written in. If omitted, UTF-8 will be used.

The Trackback sending function returns TRUE/FALSE (boolean) on success or failure. If it fails, you can retrieve the error message using:

```
$this->trackback->display_errors();
```

Receiving Trackbacks

Before you can receive Trackbacks you must create a weblog. If you don't have a blog yet there's no point in continuing.

Receiving Trackbacks is a little more complex than sending them, only because you will need a database table in which to store them, and you will need to validate the incoming trackback data. You are encouraged to implement a thorough validation process to guard against spam and duplicate data. You may also want to limit the number of Trackbacks you allow from a particular IP within a given span of time to further curtail spam. The process of receiving a Trackback is quite simple; the validation is what takes most of the effort.

Your Ping URL

In order to accept Trackbacks you must display a Trackback URL next to each one of your weblog entries. This will be the URL that people will use to send you Trackbacks (we will refer to this as your "Ping URL").

Your Ping URL must point to a controller function where your Trackback receiving code is located, and the URL must contain the ID number for each particular entry, so that when the Trackback is received you'll be able to associate it with a particular entry.

For example, if your controller class is called **Trackback**, and the receiving function is called **receive**, your Ping URLs will look something like this:

```
http://example.com/index.php/trackback/receive/entry_id
```

Where **entry_id** represents the individual ID number for each of your entries.

Creating a Trackback Table

Before you can receive Trackbacks you must create a table in which to store them. Here is a basic prototype for such a table:

```
CREATE TABLE trackbacks (
 tb_id int(10) unsigned NOT NULL auto_increment,
 entry_id int(10) unsigned NOT NULL default 0,
 url varchar(200) NOT NULL,
 title varchar(100) NOT NULL,
 excerpt text NOT NULL,
 blog_name varchar(100) NOT NULL,
 tb_date int(10) NOT NULL,
 ip_address varchar(16) NOT NULL,
 PRIMARY KEY `tb_id` (`tb_id`),
 KEY `entry_id` (`entry_id`)
);
```

The Trackback specification only requires four pieces of information to be sent in a Trackback (url, title, excerpt, blog_name), but to make the data more useful we've added a few more fields in the above table schema (date, IP address, etc.).

Processing a Trackback

Here is an example showing how you will receive and process a Trackback. The following code is intended for use within the controller function where you expect to receive Trackbacks.

```
$this->load->library('trackback');
$this->load->database();

if ($this->uri->segment(3) == FALSE)
{
   $this->trackback->send_error("Unable to determine the entry ID");
}

if ( ! $this->trackback->receive())
{
   $this->trackback->send_error("The Trackback did not contain valid data");
}

$data = array(
            'tb_id'      => '',
            'entry_id'   => $this->uri->segment(3),
            'url'        => $this->trackback->data('url'),
            'title'      => $this->trackback->data('title'),
            'excerpt'    => $this->trackback->data('excerpt'),
            'blog_name'  => $this->trackback->data('blog_name'),
            'tb_date'    => time(),
            'ip_address' => $this->input->ip_address()
            );

$sql = $this->db->insert_string('trackbacks', $data);
$this->db->query($sql);

$this->trackback->send_success();
```

Notes:

The entry ID number is expected in the third segment of your URL. This is based on the URI example we gave earlier:

```
http://example.com/index.php/trackback/receive/entry_id
```

Notice the entry_id is in the third URI segment, which you can retrieve using:

```
$this->uri->segment(3);
```

In our Trackback receiving code above, if the third segment is missing, we will issue an error. Without a valid entry ID, there's no reason to continue.

The **$this->trackback->receive()** function is simply a validation function that looks at the incoming data and makes sure it contains the four pieces of data that are required (url, title, excerpt, blog_name). It returns TRUE on success and FALSE on failure. If it fails you will issue an error message.

The incoming Trackback data can be retrieved using this function:

```
$this->trackback->data('item')
```

Where **item** represents one of these four pieces of info: url, title, excerpt, or blog_name

If the Trackback data is successfully received, you will issue a success message using:

```
$this->trackback->send_success();
```

Note: The above code contains no data validation, which you are encouraged to add.

Template Parser Class

The Template Parser Class enables you to parse pseudo-variables contained within your view files. It can parse simple variables or variable tag pairs. If you've never used a template engine, pseudo-variables look like this:

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

<h3>{blog_heading}</h3>

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

These variables are not actual PHP variables, but rather plain text representations that allow you to eliminate PHP from your templates (view files).

Note: CodeIgniter does **not** require you to use this class since using pure PHP in your view pages lets them run a little faster. However, some developers prefer to use a template engine if they work with designers who they feel would find some confusion working with PHP.

Also Note: The Template Parser Class is **not** a full-blown template parsing solution. We've kept it very lean on purpose in order to maintain maximum performance.

Initializing the Class

Like most other classes in CodeIgniter, the Parser class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Parser library object will be available using: **$this->parser**

The following functions are available in this library:

$this->parser->parse()

This method accepts a template name and data array as input, and it generates a parsed version. Example:

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

```
$data = array(
        'blog_title' => 'My Blog Title',
        'blog_heading' => 'My Blog Heading'
        );

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

The first parameter contains the name of the view file (in this example the file would be called blog_template.php), and the second parameter contains an associative array of data to be replaced in the template. In the above example, the template would contain two variables: {blog_title} and {blog_heading}

There is no need to "echo" or do something with the data returned by **$this->parser->parse()**. It is automatically passed to the output class to be sent to the browser. However, if you do want the data returned instead of sent to the output class you can pass TRUE (boolean) to the third parameter:

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

Variable Pairs

The above example code allows simple variables to be replaced. What if you would like an entire block of variables to be repeated, with each iteration containing new values? Consider the template example we showed at the top of the page:

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

<h3>{blog_heading}</h3>

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

In the above code you'll notice a pair of variables: **{blog_entries}** data... **{/blog_entries}**. In a case like this, the entire chunk of data between these pairs would be repeated multiple times, corresponding to the number of rows in a result.

Parsing variable pairs is done using the identical code shown above to parse single variables, except, you will add a multi-dimensional array corresponding to your variable pair data. Consider this example:

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

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

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

If your "pair" data is coming from a database result, which is already a multi-dimensional array, you can simply use the database result_array() function:

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

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

$data = array(
          'blog_title'   => 'My Blog Title',
          'blog_heading' => 'My Blog Heading',
          'blog_entries' => $query->result_array()
        );

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

Typography Class

The Typography Class provides functions that help you format text.

Initializing the Class

Like most other classes in CodeIgniter, the Typography class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Typography library object will be available using: **$this->typography**

auto_typography()

Formats text so that it is semantically and typographically correct HTML. Takes a string as input and returns it with the following formatting:

- Surrounds paragraphs within <p></p> (looks for double line breaks to identify paragraphs).
- Single line breaks are converted to <br />, except those that appear within <pre> tags.
- Block level elements, like <div> tags, are not wrapped within paragraphs, but their contained text is if it contains paragraphs.
- Quotes are converted to correctly facing curly quote entities, except those that appear within tags.
- Apostrophes are converted to curly apostrophe entities.
- Double dashes (either like -- this or like--this) are converted to em—dashes.
- Three consecutive periods either preceding or following a word are converted to ellipsis…
- Double spaces following sentences are converted to non-breaking spaces to mimic double spacing.

Usage example:

```
$string = $this->typography->auto_typography($string);
```

Parameters

There is one optional parameters that determines whether the parser should reduce more then two consecutive line breaks down to two. Use boolean **TRUE** or **FALSE**.

By default the parser does not reduce line breaks. In other words, if no parameters are submitted, it is the same as doing this:

```
$string = $this->typography->auto_typography($string, FALSE);
```

Note: Typographic formatting can be processor intensive, particularly if you have a lot of content being formatted. If you choose to use this function you may want to consider caching your pages.

format_characters()

This function is similar to the **auto_typography** function above, except that it only does character conversion:

- Quotes are converted to correctly facing curly quote entities, except those that appear within tags.
- Apostrophes are converted to curly apostrophe entities.
- Double dashes (either like -- this or like--this) are converted to em—dashes.
- Three consecutive periods either preceding or following a word are converted to ellipsis…
- Double spaces following sentences are converted to non-breaking spaces to mimic double spacing.

Usage example:

```
$string = $this->typography->format_characters($string);
```

nl2br_except_pre()

Converts newlines to <br /> tags unless they appear within <pre> tags. This function is identical to the native PHP **nl2br()** function, except that it ignores <pre> tags.

Usage example:

```
$string = $this->typography->nl2br_except_pre($string);
```

protect_braced_quotes

When using the Typography library in conjunction with the Template Parser library it can often be desirable to protect single and double quotes within curly braces. To enable this, set the **protect_braced_quotes** class property to **TRUE**.

Usage example:

```
$this->load->library('typography');
$this->typography->protect_braced_quotes = TRUE;
```

Unit Testing Class

Unit testing is an approach to software development in which tests are written for each function in your application. If you are not familiar with the concept you might do a little googling on the subject.

CodeIgniter's Unit Test class is quite simple, consisting of an evaluation function and two result functions. It's not intended to be a full-blown test suite but rather a simple mechanism to evaluate your code to determine if it is producing the correct data type and result.

Initializing the Class

Like most other classes in CodeIgniter, the Unit Test class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Unit Test object will be available using: **$this->unit**

Running Tests

Running a test involves supplying a test and an expected result to the following function:

$this->unit->run(test, expected result, 'test name');

Where **test** is the result of the code you wish to test, **expected result** is the data type you expect, and **test name** is an optional name you can give your test. Example:

```
$test = 1 + 1;

$expected_result = 2;

$test_name = 'Adds one plus one';

$this->unit->run($test, $expected_result, $test_name);
```

The expected result you supply can either be a literal match, or a data type match. Here's an example of a literal:

```
$this->unit->run('Foo', 'Foo');
```

Here is an example of a data type match:

```
$this->unit->run('Foo', 'is_string');
```

Notice the use of "is_string" in the second parameter? This tells the function to evaluate whether your test is producing a string as the result. Here is a list of allowed comparison types:

- is_string
- is_bool
- is_true
- is_false
- is_int
- is_numeric
- is_float
- is_double
- is_array
- is_null

Generating Reports

You can either display results after each test, or your can run several tests and generate a report at the end. To show a report directly simply echo or return the **run** function:

```
echo $this->unit->run($test, $expected_result);
```

To run a full report of all tests, use this:

```
echo $this->unit->report();
```

The report will be formatted in an HTML table for viewing. If you prefer the raw data you can retrieve an array using:

```
echo $this->unit->result();
```

Strict Mode

By default the unit test class evaluates literal matches loosely. Consider this example:

```
$this->unit->run(1, TRUE);
```

The test is evaluating an integer, but the expected result is a boolean. PHP, however, due to it's loose data-typing will evaluate the above code as TRUE using a normal equality test:

```
if (1 == TRUE) echo 'This evaluates as true';
```

If you prefer, you can put the unit test class in to strict mode, which will compare the data type as well as the value:

```
if (1 === TRUE) echo 'This evaluates as FALSE';
```

To enable strict mode use this:

```
$this->unit->use_strict(TRUE);
```

Enabling/Disabling Unit Testing

If you would like to leave some testing in place in your scripts, but not have it run unless you need it, you can disable unit testing using:

```
$this->unit->active(FALSE)
```

Creating a Template

If you would like your test results formatted differently then the default you can set your own template. Here is an example of a simple template. Note the required pseudo-variables:

```
$str = '
<table border="0" cellpadding="4" cellspacing="1">
    {rows}
        <tr>
        <td>{item}</td>
        <td>{result}</td>
        </tr>
    {/rows}
</table>';

$this->unit->set_template($str);
```

Note: Your template must be declared **before** running the unit test process.

URI Class

The URI Class provides functions that help you retrieve information from your URI strings. If you use URI routing, you can also retrieve information about the re-routed segments.

Note: This class is initialized automatically by the system so there is no need to do it manually.

$this->uri->segment(n)

Permits you to retrieve a specific segment. Where **n** is the segment number you wish to retrieve. Segments are numbered from left to right. For example, if your full URL is this:

```
http://example.com/index.php/news/local/metro/crime_is_up
```

The segment numbers would be this:

1. news
2. local
3. metro
4. crime_is_up

By default the function returns FALSE (boolean) if the segment does not exist. There is an optional second parameter that permits you to set your own default value if the segment is missing. For example, this would tell the function to return the number zero in the event of failure:

```
$product_id = $this->uri->segment(3, 0);
```

It helps avoid having to write code like this:

```
if ($this->uri->segment(3) === FALSE)
{
    $product_id = 0;
}
else
{
    $product_id = $this->uri->segment(3);
}
```

$this->uri->rsegment(n)

This function is identical to the previous one, except that it lets you retrieve a specific segment from your re-routed URI in the event you are using CodeIgniter's URI Routing feature.

$this->uri->slash_segment(n)

This function is almost identical to **$this->uri->segment()**, except it adds a trailing and/or leading slash based on the second parameter. If the parameter is not used, a trailing slash added. Examples:

```
$this->uri->slash_segment(3);
$this->uri->slash_segment(3, 'leading');
$this->uri->slash_segment(3, 'both');
```

Returns:

1. segment/
2. /segment
3. /segment/

$this->uri->slash_rsegment(n)

This function is identical to the previous one, except that it lets you add slashes a specific segment from your re-routed URI in the event you are using CodeIgniter's URI Routing feature.

$this->uri->uri_to_assoc(n)

This function lets you turn URI segments into and associative array of key/value pairs. Consider this URI:

```
index.php/user/search/name/joe/location/UK/gender/male
```

Using this function you can turn the URI into an associative array with this prototype:

```
[array]
(
    'name' => 'joe'
    'location' => 'UK'
    'gender' => 'male'
)
```

The first parameter of the function lets you set an offset. By default it is set to **3** since your URI will normally contain a controller/function in the first and second segments. Example:

```
$array = $this->uri->uri_to_assoc(3);

echo $array['name'];
```

The second parameter lets you set default key names, so that the array returned by the function will always contain expected indexes, even if missing from the URI. Example:

```
$default = array('name', 'gender', 'location', 'type', 'sort');

$array = $this->uri->uri_to_assoc(3, $default);
```

If the URI does not contain a value in your default, an array index will be set to that name, with a value of FALSE.

Lastly, if a corresponding value is not found for a given key (if there is an odd number of URI segments) the value will be set to FALSE (boolean).

$this->uri->ruri_to_assoc(n)

This function is identical to the previous one, except that it creates an associative array using the re-routed URI in the event you are using CodeIgniter's URI Routing feature.

$this->uri->assoc_to_uri()

Takes an associative array as input and generates a URI string from it. The array keys will be included in the string. Example:

```
$array = array('product' => 'shoes', 'size' => 'large', 'color' => 'red');

$str = $this->uri->assoc_to_uri($array);

// Produces: product/shoes/size/large/color/red
```

$this->uri->uri_string()

Returns a string with the complete URI. For example, if this is your full URL:

```
http://example.com/index.php/news/local/345
```

The function would return this:

```
/news/local/345
```

$this->uri->ruri_string(n)

This function is identical to the previous one, except that it returns the re-routed URI in the event you are using CodeIgniter's URI Routing feature.

$this->uri->total_segments()

Returns the total number of segments.

$this->uri->total_rsegments()

This function is identical to the previous one, except that it returns the total number of segments in your re-routed URI in the event you are using CodeIgniter's URI Routing feature.

$this->uri->segment_array()

Returns an array containing the URI segments. For example:

```
$segs = $this->uri->segment_array();

foreach ($segs as $segment)
{
   echo $segment;
   echo '<br />';
}
```

$this->uri->rsegment_array()

This function is identical to the previous one, except that it returns the array of segments in your re-routed URI in the event you are using CodeIgniter's URI Routing feature.

User Agent Class

The User Agent Class provides functions that help identify information about the browser, mobile device, or robot visiting your site. In addition you can get referrer information as well as language and supported character-set information.

Initializing the Class

Like most other classes in CodeIgniter, the User Agent class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the object will be available using: **$this->agent**

User Agent Definitions

The user agent name definitions are located in a config file located at: **application/config/user_agents.php**. You may add items to the various user agent arrays if needed.

Example

When the User Agent class is initialized it will attempt to determine whether the user agent browsing your site is a web browser, a mobile device, or a robot. It will also gather the platform information if it is available.

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

if ($this->agent->is_browser())
{
   $agent = $this->agent->browser().' '.$this->agent->version();
}
elseif ($this->agent->is_robot())
{
   $agent = $this->agent->robot();
}
elseif ($this->agent->is_mobile())
{
   $agent = $this->agent->mobile();
}
else
{
   $agent = 'Unidentified User Agent';
}

echo $agent;

echo $this->agent->platform(); // Platform info (Windows, Linux, Mac, etc.)
```

Function Reference

$this->agent->is_browser()

Returns TRUE/FALSE (boolean) if the user agent is a known web browser.

$this->agent->is_mobile()

Returns TRUE/FALSE (boolean) if the user agent is a known mobile device.

$this->agent->is_robot()

Returns TRUE/FALSE (boolean) if the user agent is a known robot.

Note: The user agent library only contains the most common robot definitions. It is not a complete list of bots. There are hundreds of them so searching for each one would not be very efficient. If you find that some bots that commonly visit your site are missing from the list you can add them to your **application/config/user_agents.php** file.

$this->agent->is_referral()

Returns TRUE/FALSE (boolean) if the user agent was referred from another site.

$this->agent->browser()

Returns a string containing the name of the web browser viewing your site.

$this->agent->version()

Returns a string containing the version number of the web browser viewing your site.

$this->agent->mobile()

Returns a string containing the name of the mobile device viewing your site.

$this->agent->robot()

Returns a string containing the name of the robot viewing your site.

$this->agent->platform()

Returns a string containing the platform viewing your site (Linux, Windows, OS X, etc.).

$this->agent->referrer()

The referrer, if the user agent was referred from another site. Typically you'll test for this as follows:

```
if ($this->agent->is_referral())
{
    echo $this->agent->referrer();
}
```

$this->agent->agent_string()

Returns a string containing the full user agent string. Typically it will be something like this:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.0.4) Gecko/20060613 Camino/1.0.2
```

$this->agent->accept_lang()

Lets you determine if the user agent accepts a particular language. Example:

```
if ($this->agent->accept_lang('en'))
{
    echo 'You accept English!';
}
```

Note: This function is not typically very reliable since some browsers do not provide language info, and even among those that do, it is not always accurate.

$this->agent->accept_charset()

Lets you determine if the user agent accepts a particular character set. Example:

```
if ($this->agent->accept_charset('utf-8'))
{
    echo 'You browser supports UTF-8!';
}
```

Note: This function is not typically very reliable since some browsers do not provide character-set info, and even among those that do, it is not always accurate.

XML-RPC and XML-RPC Server Classes

CodeIgniter's XML-RPC classes permit you to send requests to another server, or set up your own XML-RPC server to receive requests.

What is XML-RPC?

Quite simply it is a way for two computers to communicate over the internet using XML. One computer, which we will call the **client**, sends an XML-RPC **request** to another computer, which we will call the **server**. Once the server receives and processes the request it will send back a **response** to the client.

For example, using the MetaWeblog API, an XML-RPC Client (usually a desktop publishing tool) will send a request to an XML-RPC Server running on your site. This request might be a new weblog entry being sent for publication, or it could be a request for an existing entry for editing. When the XML-RPC Server receives this request it will examine it to determine which class/method should be called to process the request. Once processed, the server will then send back a response message.

For detailed specifications, you can visit the XML-RPC site.

Initializing the Class

Like most other classes in CodeIgniter, the XML-RPC and XML-RPCS classes are initialized in your controller using the **$this->load->library** function:

To load the XML-RPC class you will use:

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

Once loaded, the xml-rpc library object will be available using: **$this->xmlrpc**

To load the XML-RPC Server class you will use:

```
$this->load->library('xmlrpc');
$this->load->library('xmlrpcs');
```

Once loaded, the xml-rpcs library object will be available using: **$this->xmlrpcs**

Note: When using the XML-RPC Server class you must load BOTH the XML-RPC class and the XML-RPC Server class.

Sending XML-RPC Requests

To send a request to an XML-RPC server you must specify the following information:

- The URL of the server
- The method on the server you wish to call
- The *request* data (explained below).

Here is a basic example that sends a simple Weblogs.com ping to the Ping-o-Matic

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

$this->xmlrpc->server('http://rpc.pingomatic.com/', 80);
$this->xmlrpc->method('weblogUpdates.ping');

$request = array('My Photoblog', 'http://www.my-site.com/photoblog/');
$this->xmlrpc->request($request);

if ( ! $this->xmlrpc->send_request())
{
    echo $this->xmlrpc->display_error();
}
```

Explanation

The above code initializes the XML-RPC class, sets the server URL and method to be called (weblogUpdates.ping). The request (in this case, the title and URL of your site) is placed into an array for transportation, and compiled using the request() function. Lastly, the full request is sent. If the **send_request()** method returns false we will display the error message sent back from the XML-RPC Server.

Anatomy of a Request

An XML-RPC **request** is simply the data you are sending to the XML-RPC server. Each piece of data in a request is referred to as a **request parameter**. The above example has two parameters: The URL and title of your site. When the XML-RPC server receives your request, it will look for parameters it requires.

Request parameters must be placed into an array for transportation, and each parameter can be one of seven data types (strings, numbers, dates, etc.). If your parameters are something other than strings you will have to include the data type in the request array.

Here is an example of a simple array with three parameters:

```
$request = array('John', 'Doe', 'www.some-site.com');
$this->xmlrpc->request($request);
```

If you use data types other than strings, or if you have several different data types, you will place each parameter into its own array, with the data type in the second position:

```
$request = array (
                   array('John', 'string'),
                   array('Doe', 'string'),
                   array(FALSE, 'boolean'),
                   array(12345, 'int')
                 );
$this->xmlrpc->request($request);
```

The Data Types section below has a full list of data types.

Creating an XML-RPC Server

An XML-RPC Server acts as a traffic cop of sorts, waiting for incoming requests and redirecting them to the appropriate functions for processing.

To create your own XML-RPC server involves initializing the XML-RPC Server class in your controller where you expect the incoming request to appear, then setting up an array with mapping instructions so that incoming requests can be sent to the appropriate class and method for processing.

Here is an example to illustrate:

```
$this->load->library('xmlrpc');
$this->load->library('xmlrpcs');

$config['functions']['new_post'] = array('function' => 'My_blog.new_entry'),
$config['functions']['update_post'] = array('function' => 'My_blog.update_entry');
$config['object'] = $this;

$this->xmlrpcs->initialize($config);
$this->xmlrpcs->serve();
```

The above example contains an array specifying two method requests that the Server allows. The allowed methods are on the left side of the array. When either of those are received, they will be mapped to the class and method on the right.

The '**object**' key is a special key that you pass an instantiated class object with, which is necessary when the method you are mapping to is not part of the CodeIgniter super object.

In other words, if an XML-RPC Client sends a request for the **new_post** method, your server will load the **My_blog** class and call the **new_entry** function. If the request is for the **update_post** method, your server will load the **My_blog** class and call the **update_entry** function.

The function names in the above example are arbitrary. You'll decide what they should be called on your server, or if you are using standardized APIs, like the Blogger or MetaWeblog API, you'll use their function names.

Processing Server Requests

When the XML-RPC Server receives a request and loads the class/method for processing, it will pass an object to that method containing the data sent by the client.

Using the above example, if the **new_post** method is requested, the server will expect a class to exist with this prototype:

```
class My_blog extends Controller {

   function new_post($request)
   {

   }
}
```

The **$request** variable is an object compiled by the Server, which contains the data sent by the XML-RPC Client. Using this object you will have access to the *request parameters* enabling you to process the request. When you are done you will send a **Response** back to the Client.

Below is a real-world example, using the Blogger API. One of the methods in the Blogger API is **getUserInfo()**. Using this method, an XML-RPC Client can send the Server a username and password, in return the Server sends back information about that particular user (nickname, user ID, email address, etc.). Here is how the processing function might look:

```
class My_blog extends Controller {

   function getUserInfo($request)
   {
      $username = 'smitty';
      $password = 'secretsmittypass';

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

      $parameters = $request->output_parameters();

      if ($parameters['1'] != $username AND $parameters['2'] != $password)
      {
         return $this->xmlrpc->send_error_message('100', 'Invalid Access');
      }

      $response = array(array('nickname'  => array('Smitty','string'),
                              'userid'    => array('99','string'),
                              'url'       => array('http://yoursite.com','string'),
                              'email'     => array('jsmith@yoursite.com','string'),
                              'lastname'  => array('Smith','string'),
                              'firstname' => array('John','string')
                              ),
                         'struct');

      return $this->xmlrpc->send_response($response);
   }
}
```

Notes:

The **output_parameters()** function retrieves an indexed array corresponding to the request parameters sent by the client. In the above example, the output parameters will be the username and password.

If the username and password sent by the client were not valid, and error message is returned using **send_error_message()**.

If the operation was successful, the client will be sent back a response array containing the user's info.

Formatting a Response

Similar to *Requests*, *Responses* must be formatted as an array. However, unlike requests, a response is an array **that contains a single item**. This item can be an array with several additional arrays, but there can be only one primary array index. In other words, the basic prototype is this:

```
$response = array('Response data', 'array');
```

Responses, however, usually contain multiple pieces of information. In order to accomplish this we must put the response into its own array so that the primary array continues to contain a single piece of data. Here's an example showing how this might be accomplished:

```
$response = array (
                   array(
                         'first_name' => array('John', 'string'),
                         'last_name' => array('Doe', 'string'),
                         'member_id' => array(123435, 'int'),
                         'todo_list' => array(array('clean house', 'call mom', 'water plants'), 'array'),
                        ),
                     'struct'
                     );
```

Notice that the above array is formatted as a **struct**. This is the most common data type for responses.

As with Requests, a response can be one of the seven data types listed in the Data Types section.

Sending an Error Response

If you need to send the client an error response you will use the following:

```
return $this->xmlrpc->send_error_message('123', 'Requested data not available');
```

The first parameter is the error number while the second parameter is the error message.

Creating Your Own Client and Server

To help you understand everything we've covered thus far, let's create a couple controllers that act as XML-RPC Client and Server. You'll use the Client to send a request to the Server and receive a response.

The Client

Using a text editor, create a controller called **xmlrpc_client.php**. In it, place this code and save it to your **applications/controllers/** folder:

```
<?php

class Xmlrpc_client extends Controller {

	function index()
	{
		$this->load->helper('url');
		$server_url = site_url('xmlrpc_server');

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

		$this->xmlrpc->server($server_url, 80);
		$this->xmlrpc->method('Greetings');

		$request = array('How is it going?');
		$this->xmlrpc->request($request);

		if ( ! $this->xmlrpc->send_request())
		{
			echo $this->xmlrpc->display_error();
		}
		else
		{
			echo '<pre>';
			print_r($this->xmlrpc->display_response());
			echo '</pre>';
		}
	}
}
?>
```

Note: In the above code we are using a "url helper". You can find more information in the Helpers Functions page.

The Server

Using a text editor, create a controller called **xmlrpc_server.php**. In it, place this code and save it to your **applications/controllers/** folder:

```
<?php

class Xmlrpc_server extends Controller {

	function index()
	{
		$this->load->library('xmlrpc');
		$this->load->library('xmlrpcs');

		$config['functions']['Greetings'] = array('function' => 'Xmlrpc_server.process');

		$this->xmlrpcs->initialize($config);
		$this->xmlrpcs->serve();
	}


	function process($request)
	{
		$parameters = $request->output_parameters();

		$response = array(
					array(
						'you_said'  => $parameters['0'],
						'i_respond' => 'Not bad at all.'),
					'struct');

		return $this->xmlrpc->send_response($response);
	}
}
?>
```

Try it!

Now visit the your site using a URL similar to this:

```
example.com/index.php/xmlrpc_client/
```

You should now see the message you sent to the server, and its response back to you.

The client you created sends a message ("How's is going?") to the server, along with a request for the "Greetings" method. The Server receives the request and maps it to the "process" function, where a response is sent back.

Using Associative Arrays In a Request Parameter

If you wish to use an associative array in your method parameters you will need to use a struct datatype:

```
$request = array(
           array(
               // Param 0
               array(
                   'name'=>'John'
                   ),
                   'struct'
```

```
            ),
            array(
                // Param 1
                array(
                    'size'=>'large',
                    'shape'=>'round'
                    ),
                'struct'
            )
        );
$this->xmlrpc->request($request);
```

You can retrieve the associative array when processing the request in the Server.

```
$parameters = $request->output_parameters();
$name = $parameters['0']['name'];
$size = $parameters['1']['size'];
$size = $parameters['1']['shape'];
```

XML-RPC Function Reference

$this->xmlrpc->server()

Sets the URL and port number of the server to which a request is to be sent:

```
$this->xmlrpc->server('http://www.sometimes.com/pings.php', 80);
```

$this->xmlrpc->timeout()

Set a time out period (in seconds) after which the request will be canceled:

```
$this->xmlrpc->timeout(6);
```

$this->xmlrpc->method()

Sets the method that will be requested from the XML-RPC server:

```
$this->xmlrpc->method('method');
```

Where **method** is the name of the method.

$this->xmlrpc->request()

Takes an array of data and builds request to be sent to XML-RPC server:

```
$request = array(array('My Photoblog', 'string'), 'http://www.yoursite.com/photoblog/');
```

```
$this->xmlrpc->request($request);
```

$this->xmlrpc->send_request()

The request sending function. Returns boolean TRUE or FALSE based on success for failure, enabling it to be used conditionally.

$this->xmlrpc->set_debug(TRUE);

Enables debugging, which will display a variety of information and error data helpful during development.

$this->xmlrpc->display_error()

Returns an error message as a string if your request failed for some reason.

```
echo $this->xmlrpc->display_error();
```

$this->xmlrpc->display_response()

Returns the response from the remote server once request is received. The response will typically be an associative array.

```
$this->xmlrpc->display_response();
```

$this->xmlrpc->send_error_message()

This function lets you send an error message from your server to the client. First parameter is the error number while the second parameter is the error message.

```
return $this->xmlrpc->send_error_message('123', 'Requested data not available');
```

$this->xmlrpc->send_response()

Lets you send the response from your server to the client. An array of valid data values must be sent with this method.

```
$response = array(
            array(
                 'flerror' => array(FALSE, 'boolean'),
                 'message' => "Thanks for the ping!")
               )
            'struct');
return $this->xmlrpc->send_response($response);
```

Data Types

According to the XML-RPC spec there are seven types of values that you can send via XML-RPC:

- *int* or *i4*
- *boolean*
- *string*
- *double*
- *dateTime.iso8601*
- *base64*
- *struct* (contains array of values)
- *array* (contains array of values)

Zip Encoding Class

CodeIgniter's Zip Encoding Class classes permit you to create Zip archives. Archives can be downloaded to your desktop or saved to a directory.

Initializing the Class

Like most other classes in CodeIgniter, the Zip class is initialized in your controller using the **$this->load->library** function:

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

Once loaded, the Zip library object will be available using: **$this->zip**

Usage Example

This example demonstrates how to compress a file, save it to a folder on your server, and download it to your desktop.

```
$name = 'mydata1.txt';
$data = 'A Data String!';

$this->zip->add_data($name, $data);

// Write the zip file to a folder on your server. Name it "my_backup.zip"
$this->zip->archive('/path/to/directory/my_backup.zip');

// Download the file to your desktop. Name it "my_backup.zip"
$this->zip->download('my_backup.zip');
```

Function Reference

$this->zip->add_data()

Permits you to add data to the Zip archive. The first parameter must contain the name you would like given to the file, the second parameter must contain the file data as a string:

```
$name = 'my_bio.txt';
$data = 'I was born in an elevator...';

$this->zip->add_data($name, $data);
```

You are allowed multiple calls to this function in order to add several files to your archive. Example:

```
$name = 'mydata1.txt';
```

```
$data = 'A Data String!';
$this->zip->add_data($name, $data);

$name = 'mydata2.txt';
$data = 'Another Data String!';
$this->zip->add_data($name, $data);
```

Or you can pass multiple files using an array:

```
$data = array(
            'mydata1.txt' => 'A Data String!',
            'mydata2.txt' => 'Another Data String!'
         );

$this->zip->add_data($data);

$this->zip->download('my_backup.zip');
```

If you would like your compressed data organized into sub-folders, include the path as part of the filename:

```
$name = 'personal/my_bio.txt';
$data = 'I was born in an elevator...';

$this->zip->add_data($name, $data);
```

The above example will place **my_bio.txt** inside a folder called **personal**.

$this->zip->add_dir()

Permits you to add a directory. Usually this function is unnecessary since you can place your data into folders when using **$this->zip->add_data()**, but if you would like to create an empty folder you can do so. Example:

```
$this->zip->add_dir('myfolder'); // Creates a folder called "myfolder"
```

$this->zip->read_file()

Permits you to compress a file that already exists somewhere on your server. Supply a file path and the zip class will read it and add it to the archive:

```
$path = '/path/to/photo.jpg';

$this->zip->read_file($path);

// Download the file to your desktop. Name it "my_backup.zip"
$this->zip->download('my_backup.zip');
```

If you would like the Zip archive to maintain the directory structure of the file in it, pass **TRUE** (boolean) in the second parameter. Example:

```
$path = '/path/to/photo.jpg';

$this->zip->read_file($path, TRUE);

// Download the file to your desktop. Name it "my_backup.zip"
$this->zip->download('my_backup.zip');
```

In the above example, **photo.jpg** will be placed inside two folders: **path/to/**

$this->zip->read_dir()

Permits you to compress a folder (and its contents) that already exists somewhere on your server. Supply a file path to the directory and the zip class will recursively read it and recreate it as a Zip archive. All files contained within the supplied path will be encoded, as will any sub-folders contained within it. Example:

```
$path = '/path/to/your/directory/';

$this->zip->read_dir($path);

// Download the file to your desktop. Name it "my_backup.zip"
$this->zip->download('my_backup.zip');
```

$this->zip->archive()

Writes the Zip-encoded file to a directory on your server. Submit a valid server path ending in the file name. Make sure the directory is writable (666 or 777 is usually OK). Example:

```
$this->zip->archive('/path/to/folder/myarchive.zip'); // Creates a file named myarchive.zip
```

$this->zip->download()

Causes the Zip file to be downloaded from your server. The function must be passed the name you would like the zip file called. Example:

```
$this->zip->download('latest_stuff.zip'); // File will be named "latest_stuff.zip"
```

Note: Do not display any data in the controller in which you call this function since it sends various server headers that cause the download to happen and the file to be treated as binary.

$this->zip->get_zip()

Returns the Zip-compressed file data. Generally you will not need this function unless you want to do something unique with the data. Example:

```
$name = 'my_bio.txt';
$data = 'I was born in an elevator...';

$this->zip->add_data($name, $data);

$zip_file = $this->zip->get_zip();
```

$this->zip->clear_data()

The Zip class caches your zip data so that it doesn't need to recompile the Zip archive for each function you use above. If, however, you need to create multiple Zips, each with different data, you can clear the cache between calls. Example:

```
$name = 'my_bio.txt';
$data = 'I was born in an elevator...';

$this->zip->add_data($name, $data);
$zip_file = $this->zip->get_zip();

$this->zip->clear_data();

$name = 'photo.jpg';
$this->zip->read_file("/path/to/photo.jpg"); // Read the file's contents


$this->zip->download('myphotos.zip');
```

Array Helper

The Array Helper file contains functions that assist in working with arrays.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

element()

Lets you fetch an item from an array. The function tests whether the array index is set and whether it has a value. If a value exists it is returned. If a value does not exist it returns FALSE, or whatever you've specified as the default value via the third parameter. Example:

```
$array = array('color' => 'red', 'shape' => 'round', 'size' => '');

// returns "red"
echo element('color', $array);

// returns NULL
echo element('size', $array, NULL);
```

random_element()

Takes an array as input and returns a random element from it. Usage example:

```
$quotes = array(
        "I find that the harder I work, the more luck I seem to have. - Thomas Jefferson",
        "Don't stay in bed, unless you can make money in bed. - George Burns",
        "We didn't lose the game; we just ran out of time. - Vince Lombardi",
        "If everything seems under control, you're not going fast enough. - Mario Andretti",
        "Reality is merely an illusion, albeit a very persistent one. - Albert Einstein",
        "Chance favors the prepared mind - Louis Pasteur"
        );

echo random_element($quotes);
```

Compatibility Helper

The Compatibility Helper file contains PHP 4 implementations of some PHP 5 only native PHP functions and constants. This can be useful if you'd like to take advantage of some of these native function but your application may end up running on a PHP 4 server. In these cases, it may be advantageous to Auto-load the Compatibility Helper so you do not have to load it in each controller.

Note: There are a few compatibility functions that are in CodeIgniter's native Compat.php file. You may use those functions without loading this helper. The functions are split between that file and this Helper so that only functions required by the framework are included by default. This way, whether or not you load the additional functions in this Helper remains your choice.

Loading this Helper

This helper is loaded using the following code:

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

Available Constants

The following constants are available:

PHP_EOL

The newline character for the server's current OS, e.g. on Windows systems "\r\n", on *nix "\n".

Available Functions

The following functions are available (see linked PHP documentation for documentation):

file_put_contents() - The fourth parameter, $context, is not supported.

fputcsv()

http_build_query()

str_ireplace() - The fourth parameter, $count, is not supported, as PHP 4 would make it become required.

stripos()

Cookie Helper

The Cookie Helper file contains functions that assist in working with cookies.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

set_cookie()

Sets a cookie containing the values you specify. There are two ways to pass information to this function so that a cookie can be set: Array Method, and Discrete Parameters:

Array Method

Using this method, an associative array is passed to the first parameter:

```
$cookie = array(
               'name'   => 'The Cookie Name',
               'value'  => 'The Value',
               'expire' => '86500',
               'domain' => '.some-domain.com',
               'path'   => '/',
               'prefix' => 'myprefix_',
           );

set_cookie($cookie);
```

Notes:

Only the name and value are required. To delete a cookie set it with the expiration blank.

The expiration is set in **seconds**, which will be added to the current time. Do not include the time, but rather only the number of seconds from *now* that you wish the cookie to be valid. If the expiration is set to zero the cookie will only last as long as the browser is open.

For site-wide cookies regardless of how your site is requested, add your URL to the **domain** starting with a period, like this: .your-domain.com

The path is usually not needed since the function sets a root path.

The prefix is only needed if you need to avoid name collisions with other identically named cookies for your server.

Discrete Parameters

If you prefer, you can set the cookie by passing data using individual parameters:

```
set_cookie($name, $value, $expire, $domain, $path, $prefix);
```

get_cookie()

Lets you fetch a cookie. The first parameter will contain the name of the cookie you are looking for (including any prefixes):

```
get_cookie('some_cookie');
```

The function returns FALSE (boolean) if the item you are attempting to retrieve does not exist.

The second optional parameter lets you run the data through the XSS filter. It's enabled by setting the second parameter to boolean TRUE;

```
get_cookie('some_cookie', TRUE);
```

delete_cookie()

Lets you delete a cookie. Unless you've set a custom path or other values, only the name of the cookie is needed:

```
delete_cookie("name");
```

This function is otherwise identical to **set_cookie()**, except that it does not have the value and expiration parameters. You can submit an array of values in the first parameter or you can set discrete parameters.

```
delete_cookie($name, $domain, $path, $prefix)
```

Date Helper

The Date Helper file contains functions that help you work with dates.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

now()

Returns the current time as a Unix timestamp, referenced either to your server's local time or GMT, based on the "time reference" setting in your config file. If you do not intend to set your master time reference to GMT (which you'll typically do if you run a site that lets each user set their own timezone settings) there is no benefit to using this function over PHP's time() function.

mdate()

This function is identical to PHPs date() function, except that it lets you use MySQL style date codes, where each code letter is preceded with a percent sign: %Y %m %d etc.

The benefit of doing dates this way is that you don't have to worry about escaping any characters that are not date codes, as you would normally have to do with the date() function. Example:

```
$datestring = "Year: %Y Month: %m Day: %d - %h:%i %a";
$time = time();

echo mdate($datestring, $time);
```

If a timestamp is not included in the second parameter the current time will be used.

standard_date()

Lets you generate a date string in one of several standardized formats. Example:

```
$format = 'DATE_RFC822';
$time = time();

echo standard_date($format, $time);
```

The first parameter must contain the format, the second parameter must contain the date as a Unix timestamp.

Supported formats:

Constant	Description	Example
DATE_ATOM	Atom	2005-08-15T16:13:03+0000
DATE_COOKIE	HTTP Cookies	Sun, 14 Aug 2005 16:13:03 UTC
DATE_ISO8601	ISO-8601	2005-08-14T16:13:03+0000
DATE_RFC822	RFC 822	Sun, 14 Aug 2005 16:13:03 UTC
DATE_RFC850	RFC 850	Sunday, 14-Aug-05 16:13:03 UTC
DATE_RFC1036	RFC 1036	Sunday, 14-Aug-05 16:13:03 UTC
DATE_RFC1123	RFC 1123	Sun, 14 Aug 2005 16:13:03 UTC
DATE_RFC2822	RFC 2822	Sun, 14 Aug 2005 16:13:03 +0000
DATE_RSS	RSS	Sun, 14 Aug 2005 16:13:03 UTC
DATE_W3C	World Wide Web Consortium	2005-08-14T16:13:03+0000

local_to_gmt()

Takes a Unix timestamp as input and returns it as GMT. Example:

```
$now = time();

$gmt = local_to_gmt($now);
```

gmt_to_local()

Takes a Unix timestamp (referenced to GMT) as input, and converts it to a localized timestamp based on the timezone and Daylight Saving time submitted. Example:

```
$timestamp = '1140153693';
$timezone = 'UM8';
$daylight_saving = TRUE;

echo gmt_to_local($timestamp, $timezone, $daylight_saving);
```

Note: For a list of timezones see the reference at the bottom of this page.

mysql_to_unix()

Takes a MySQL Timestamp as input and returns it as Unix. Example:

```
$mysql = '20061124092345';

$unix = mysql_to_unix($mysql);
```

unix_to_human()

Takes a Unix timestamp as input and returns it in a human readable format with this prototype:

```
YYYY-MM-DD HH:MM:SS AM/PM
```

This can be useful if you need to display a date in a form field for submission.

The time can be formatted with or without seconds, and it can be set to European or US format. If only the timestamp is submitted it will return the time without seconds formatted for the U.S. Examples:

```
$now = time();

echo unix_to_human($now); // U.S. time, no seconds

echo unix_to_human($now, TRUE, 'us'); // U.S. time with seconds

echo unix_to_human($now, TRUE, 'eu'); // Euro time with seconds
```

human_to_unix()

The opposite of the above function. Takes a "human" time as input and returns it as Unix. This function is useful if you accept "human" formatted dates submitted via a form. Returns FALSE (boolean) if the date string passed to it is not formatted as indicated above. Example:

```
$now = time();

$human = unix_to_human($now);

$unix = human_to_unix($human);
```

timespan()

Formats a unix timestamp so that is appears similar to this:

```
1 Year, 10 Months, 2 Weeks, 5 Days, 10 Hours, 16 Minutes
```

The first parameter must contain a Unix timestamp. The second parameter must contain a timestamp that is greater that the first timestamp. If the second parameter empty, the current time will be used. The most common purpose for this function is to show how much time has elapsed from some point in time in the past to now. Example:

```
$post_date = '1079621429';
$now = time();

echo timespan($post_date, $now);
```

Note: The text generated by this function is found in the following language file: language/<your_lang>/date_lang.php

days_in_month()

Returns the number of days in a given month/year. Takes leap years into account. Example:

```
echo days_in_month(06, 2005);
```

If the second parameter is empty, the current year will be used.

timezones()

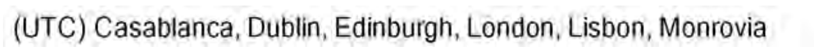
Takes a timezone reference (for a list of valid timezones, see the "Timezone Reference" below) and returns the number of hours offset from UTC.

```
echo timezones('UM5');
```

This function is useful when used with timezone_menu().

timezone_menu()

Generates a pull-down menu of timezones, like this one:



(UTC) Casablanca, Dublin, Edinburgh, London, Lisbon, Monrovia

This menu is useful if you run a membership site in which your users are allowed to set their local timezone value.

The first parameter lets you set the "selected" state of the menu. For example, to set Pacific time as the default you will do this:

```
echo timezone_menu('UM8');
```

Please see the timezone reference below to see the values of this menu.

The second parameter lets you set a CSS class name for the menu.

Note: The text contained in the menu is found in the following language file: language/<your_lang>/date_lang.php

Timezone Reference

The following table indicates each timezone and its location.

Time Zone	Location
UM12	(UTC - 12:00) Enitwetok, Kwajalien
UM11	(UTC - 11:00) Nome, Midway Island, Samoa
UM10	(UTC - 10:00) Hawaii

UM9	(UTC - 9:00) Alaska
UM8	(UTC - 8:00) Pacific Time
UM7	(UTC - 7:00) Mountain Time
UM6	(UTC - 6:00) Central Time, Mexico City
UM5	(UTC - 5:00) Eastern Time, Bogota, Lima, Quito
UM4	(UTC - 4:00) Atlantic Time, Caracas, La Paz
UM25	(UTC - 3:30) Newfoundland
UM3	(UTC - 3:00) Brazil, Buenos Aires, Georgetown, Falkland Is.
UM2	(UTC - 2:00) Mid-Atlantic, Ascention Is., St Helena
UM1	(UTC - 1:00) Azores, Cape Verde Islands
UTC	(UTC) Casablanca, Dublin, Edinburgh, London, Lisbon, Monrovia
UP1	(UTC + 1:00) Berlin, Brussels, Copenhagen, Madrid, Paris, Rome
UP2	(UTC + 2:00) Kaliningrad, South Africa, Warsaw
UP3	(UTC + 3:00) Baghdad, Riyadh, Moscow, Nairobi
UP25	(UTC + 3:30) Tehran
UP4	(UTC + 4:00) Adu Dhabi, Baku, Muscat, Tbilisi
UP35	(UTC + 4:30) Kabul
UP5	(UTC + 5:00) Islamabad, Karachi, Tashkent
UP45	(UTC + 5:30) Bombay, Calcutta, Madras, New Delhi
UP6	(UTC + 6:00) Almaty, Colomba, Dhaka
UP7	(UTC + 7:00) Bangkok, Hanoi, Jakarta
UP8	(UTC + 8:00) Beijing, Hong Kong, Perth, Singapore, Taipei
UP9	(UTC + 9:00) Osaka, Sapporo, Seoul, Tokyo, Yakutsk
UP85	(UTC + 9:30) Adelaide, Darwin
UP10	(UTC + 10:00) Melbourne, Papua New Guinea, Sydney, Vladivostok
UP11	(UTC + 11:00) Magadan, New Caledonia, Solomon Islands
UP12	(UTC + 12:00) Auckland, Wellington, Fiji, Marshall Island

Directory Helper

The Directory Helper file contains functions that assist in working with directories.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

directory_map('source directory')

This function reads the directory path specified in the first parameter and builds an array representation of it and all its contained files. Example:

```
$map = directory_map('./mydirectory/');
```

Note: Paths are almost always relative to your main index.php file.

Sub-folders contained within the directory will be mapped as well. If you wish to map only the top level directory set the second parameter to **true** (boolean):

```
$map = directory_map('./mydirectory/', TRUE);
```

By default, hidden files will not be included in the returned array. To override this behavior, you may set a third parameter to **true** (boolean):

```
$map = directory_map('./mydirectory/', FALSE, TRUE);
```

Each folder name will be an array index, while its contained files will be numerically indexed. Here is an example of a typical array:

```
Array
(
  [libraries] => Array
  (
     [0] => benchmark.html
     [1] => config.html
     [database] => Array
     (
         [0] => active_record.html
         [1] => binds.html
         [2] => configuration.html
         [3] => connecting.html
```

```
            [4] => examples.html
            [5] => fields.html
            [6] => index.html
            [7] => queries.html
        )
    [2] => email.html
    [3] => file_uploading.html
    [4] => image_lib.html
    [5] => input.html
    [6] => language.html
    [7] => loader.html
    [8] => pagination.html
    [9] => uri.html
)
```

Download Helper

The Download Helper lets you download data to your desktop.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

force_download('filename', 'data')

Generates server headers which force data to be downloaded to your desktop. Useful with file downloads. The first parameter is the **name you want the downloaded file to be named**, the second parameter is the file data. Example:

```
$data = 'Here is some text!';
$name = 'mytext.txt';

force_download($name, $data);
```

If you want to download an existing file from your server you'll need to read the file into a string:

```
$data = file_get_contents("/path/to/photo.jpg"); // Read the file's contents
$name = 'myphoto.jpg';

force_download($name, $data);
```

Email Helper

The Email Helper provides some assistive functions for working with Email. For a more robust email solution, see CodeIgniter's Email Class.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

valid_email('email')

Checks if an email is a correctly formatted email. Note that is doesn't actually prove the email will recieve mail, simply that it is a validly formed address.

It returns TRUE/FALSE

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

if (valid_email('email@somesite.com'))
{
    echo 'email is valid';
}
else
{
    echo 'email is not valid';
}
```

send_email('recipient', 'subject', 'message')

Sends an email using PHP's native mail() function. For a more robust email solution, see CodeIgniter's Email Class.

File Helper

The File Helper file contains functions that assist in working with files.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

read_file('path')

Returns the data contained in the file specified in the path. Example:

```
$string = read_file('./path/to/file.php');
```

The path can be a relative or full server path. Returns FALSE (boolean) on failure.

Note: The path is relative to your main site index.php file, NOT your controller or view files. CodeIgniter uses a front controller so paths are always relative to the main site index.

If your server is running an open_basedir restriction this function might not work if you are trying to access a file above the calling script.

write_file('path', $data)

Writes data to the file specified in the path. If the file does not exist the function will create it. Example:

```
$data = 'Some file data';

if ( ! write_file('./path/to/file.php', $data))
{
    echo 'Unable to write the file';
}
else
{
    echo 'File written!';
}
```

You can optionally set the write mode via the third parameter:

```
write_file('./path/to/file.php', $data, 'r+');
```

The default mode is **wb**. Please see the PHP user guide for mode options.

Note: In order for this function to write data to a file its file permissions must be set such that it is writable (666, 777, etc.). If the file does not already exist, the directory containing it must be writable.

Note: The path is relative to your main site index.php file, NOT your controller or view files. CodeIgniter uses a front controller so paths are always relative to the main site index.

delete_files('path')

Deletes ALL files contained in the supplied path. Example:

```
delete_files('./path/to/directory/');
```

If the second parameter is set to **true**, any directories contained within the supplied root path will be deleted as well. Example:

```
delete_files('./path/to/directory/', TRUE);
```

Note: The files must be writable or owned by the system in order to be deleted.

get_filenames('path/to/directory/')

Takes a server path as input and returns an array containing the names of all files contained within it. The file path can optionally be added to the file names by setting the second parameter to TRUE.

get_dir_file_info('path/to/directory/')

Reads the specified directory and builds an array containing the filenames, filesize, dates, and permissions. Any sub-folders contained within the specified path are read as well.

get_file_info('path/to/file', $file_information)

Given a file and path, returns the name, path, size, date modified. Second parameter allows you to explicitly declare what information you want returned; options are: name, server_path, size, date, readable, writable, executable, fileperms. Returns FALSE if the file cannot be found.

Note: The "writable" uses the PHP function is_writable() which is known to have issues on the IIS webserver. Consider using fileperms instead, which returns information from PHP's fileperms() function.

get_mime_by_extension('file')

Translates a file extension into a mime type based on config/mimes.php. Returns FALSE if it can't determine the type, or open the mime config file.

```
$file = "somefile.png";
echo $file . ' is has a mime type of ' . get_mime_by_extension($file);
```

Note: This is not an accurate way of determining file mime types, and is here strictly as a convenience. It should not be used for security.

symbolic_permissions($perms)

Takes numeric permissions (such as is returned by **fileperms()** and returns standard symbolic notation of file permissions.

```
echo symbolic_permissions(fileperms('./index.php'));

// -rw-r--r--
```

octal_permissions($perms)

Takes numeric permissions (such as is returned by **fileperms()** and returns a three character octal notation of file permissions.

```
echo octal_permissions(fileperms('./index.php'));

// 644
```

Form Helper

The Form Helper file contains functions that assist in working with forms.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

form_open()

Creates an opening form tag with a base URL **built from your config preferences**. It will optionally let you add form attributes and hidden input fields.

The main benefit of using this tag rather than hard coding your own HTML is that it permits your site to be more portable in the event your URLs ever change.

Here's a simple example:

```
echo form_open('email/send');
```

The above example would create a form that points to your base URL plus the "email/send" URI segments, like this:

```
<form method="post" action="http:/example.com/index.php/email/send" />
```

Adding Attributes

Attributes can be added by passing an associative array to the second parameter, like this:

```
$attributes = array('class' => 'email', 'id' => 'myform');

echo form_open('email/send', $attributes);
```

The above example would create a form similar to this:

```
<form method="post" action="http:/example.com/index.php/email/send"  class="email"  id="myform" />
```

Adding Hidden Input Fields

Hidden fields can be added by passing an associative array to the third parameter, like this:

```
$hidden = array('username' => 'Joe', 'member_id' => '234');
```

```
echo form_open('email/send', '', $hidden);
```

The above example would create a form similar to this:

```
<form method="post" action="http:/example.com/index.php/email/send">
<input type="hidden" name="username" value="Joe" />
<input type="hidden" name="member_id" value="234" />
```

form_open_multipart()

This function is absolutely identical to the **form_open()** tag above except that it adds a multipart attribute, which is necessary if you would like to use the form to upload files with.

form_hidden()

Lets you generate hidden input fields. You can either submit a name/value string to create one field:

```
form_hidden('username', 'johndoe');

// Would produce:

<input type="hidden" name="username" value="johndoe" />
```

Or you can submit an associative array to create multiple fields:

```
$data = array(
          'name'  => 'John Doe',
          'email' => 'john@example.com',
          'url'   => 'http://example.com'
        );

echo form_hidden($data);

// Would produce:

<input type="hidden" name="name" value="John Doe" />
<input type="hidden" name="email" value="john@example.com" />
<input type="hidden" name="url" value="http://example.com" />
```

form_input()

Lets you generate a standard text input field. You can minimally pass the field name and value in the first and second parameter:

```
echo form_input('username', 'johndoe');
```

Or you can pass an associative array containing any data you wish your form to contain:

```
$data = array(
          'name'       => 'username',
          'id'         => 'username',
          'value'      => 'johndoe',
          'maxlength'  => '100',
          'size'       => '50',
          'style'      => 'width:50%',
        );

echo form_input($data);

// Would produce:

<input type="text" name="username" id="username" value="johndoe" maxlength="100" size="50"
style="width:50%" />
```

If you would like your form to contain some additional data, like Javascript, you can pass it as a string in the third parameter:

```
$js = 'onClick="some_function()"';

echo form_input('username', 'johndoe', $js);
```

form_password()

This function is identical in all respects to the **form_input()** function above except that is sets it as a "password" type.

form_upload()

This function is identical in all respects to the **form_input()** function above except that is sets it as a "file" type, allowing it to be used to upload files.

form_textarea()

This function is identical in all respects to the **form_input()** function above except that it generates a "textarea" type. Note: Instead of the "maxlength" and "size" attributes in the above example, you will instead specify "rows" and "cols".

form_dropdown()

Lets you create a standard drop-down field. The first parameter will contain the name of the field, the second parameter will contain an associative array of options, and the third parameter will contain the value you wish to be selected. You can also pass an array of multiple items through the third parameter, and CodeIgniter will create a multiple select for you. Example:

```
$options = array(
             'small'  => 'Small Shirt',
             'med'    => 'Medium Shirt',
              'large'   => 'Large Shirt',
              'xlarge' => 'Extra Large Shirt',
            );

$shirts_on_sale = array('small', 'large');

echo form_dropdown('shirts', $options, 'large');

// Would produce:

<select name="shirts">
<option value="small">Small Shirt</option>
<option value="med">Medium Shirt</option>
<option value="large" selected="selected">Large Shirt</option>
<option value="xlarge">Extra Large Shirt</option>
</select>

echo form_dropdown('shirts', $options, $shirts_on_sale);

// Would produce:

<select name="shirts" multiple="multiple">
<option value="small" selected="selected">Small Shirt</option>
<option value="med">Medium Shirt</option>
<option value="large" selected="selected">Large Shirt</option>
<option value="xlarge">Extra Large Shirt</option>
</select>
```

If you would like the opening <select> to contain additional data, like an **id** attribute or JavaScript, you can pass it as a string in the fourth parameter:

```
$js = 'id="shirts" onChange="some_function();"';

echo form_dropdown('shirts', $options, 'large', $js);
```

If the array passed as $options is a multidimensional array, form_dropdown() will produce an <optgroup> with the array key as the label.

form_multiselect()

Lets you create a standard multiselect field. The first parameter will contain the name of the field, the second parameter will contain an associative array of options, and the third parameter will contain the value or values you wish to be selected. The parameter usage is identical to using **form_dropdown()** above, except of course that the name of the field will need to use POST array syntax, e.g. **foo[]**.

form_fieldset()

Lets you generate fieldset/legend fields.

```
echo form_fieldset('Address Information');
echo "<p>fieldset content here</p>\n";
echo form_fieldset_close();

// Produces
```

```
<fieldset>
<legend>Address Information</legend>
<p>form content here</p>
</fieldset>
```

Similar to other functions, you can submit an associative array in the second parameter if you prefer to set additional attributes.

```
$attributes = array('id' => 'address_info', 'class' => 'address_info');
echo form_fieldset('Address Information', $attributes);
echo "<p>fieldset content here</p>\n";
echo form_fieldset_close();

// Produces
<fieldset id="address_info" class="address_info">
<legend>Address Information</legend>
<p>form content here</p>
</fieldset>
```

form_fieldset_close()

Produces a closing </fieldset> tag. The only advantage to using this function is it permits you to pass data to it which will be added below the tag. For example:

```
$string = "</div></div>";

echo fieldset_close($string);

// Would produce:
</fieldset>
</div></div>
```

form_checkbox()

Lets you generate a checkbox field. Simple example:

```
echo form_checkbox('newsletter', 'accept', TRUE);

// Would produce:

<input type="checkbox" name="newsletter" value="accept" checked="checked" />
```

The third parameter contains a boolean TRUE/FALSE to determine whether the box should be checked or not.

Similar to the other form functions in this helper, you can also pass an array of attributes to the function:

```
$data = array(
    'name'        => 'newsletter',
    'id'          => 'newsletter',
    'value'       => 'accept',
    'checked'     => TRUE,
    'style'       => 'margin:10px',
    );

echo form_checkbox($data);

// Would produce:

<input type="checkbox" name="newsletter" id="newsletter" value="accept" checked="checked"
style="margin:10px" />
```

As with other functions, if you would like the tag to contain additional data, like JavaScript, you can pass it as a string in the fourth parameter:

```
$js = 'onClick="some_function()"';

echo form_checkbox('newsletter', 'accept', TRUE, $js)
```

form_radio()

This function is identical in all respects to the **form_checkbox()** function above except that is sets it as a "radio" type.

form_submit()

Lets you generate a standard submit button. Simple example:

```
echo form_submit('mysubmit', 'Submit Post!');

// Would produce:

<input type="submit" name="mysubmit" value="Submit Post!" />
```

Similar to other functions, you can submit an associative array in the first parameter if you prefer to set your own attributes. The third parameter lets you add extra data to your form, like JavaScript.

form_label()

Lets you generate a <label>. Simple example:

```
echo form_label('What is your Name', 'username');

// Would produce:
<label for="username">What is your Name</label>
```

Similar to other functions, you can submit an associative array in the third parameter if you prefer to set additional attributes.

```
$attributes = array(
    'class' => 'mycustomclass',
    'style' => 'color: #000;',
);
echo form_label('What is your Name', 'username', $attributes);

// Would produce:
<label for="username" class="mycustomclass" style="color: #000;">What is your Name</label>
```

form_reset()

Lets you generate a standard reset button. Use is identical to **form_submit()**.

form_button()

Lets you generate a standard button element. You can minimally pass the button name and content in the first and second parameter:

```
echo form_button('name','content');

// Would produce
<button name="name" type="button">Content</button>
```

Or you can pass an associative array containing any data you wish your form to contain:

```
$data = array(
    'name' => 'button',
    'id' => 'button',
    'value' => 'true',
    'type' => 'reset',
    'content' => 'Reset'
);

echo form_button($data);

// Would produce:
<button name="button" id="button" value="true" type="reset">Reset</button>
```

If you would like your form to contain some additional data, like JavaScript, you can pass it as a string in the third parameter:

```
$js = 'onClick="some_function()"';

echo form_button('mybutton', 'Click Me', $js);
```

form_close()

Produces a closing </form> tag. The only advantage to using this function is it permits you to pass data to it which will be added below the tag. For example:

```
$string = "</div></div>";
```

```
echo form_close($string);

// Would produce:

</form>
</div></div>
```

form_prep()

Allows you to safely use HTML and characters such as quotes within form elements without breaking out of the form. Consider this example:

```
$string = 'Here is a string containing "quoted" text.';

<input type="text" name="myform" value="$string" />
```

Since the above string contains a set of quotes it will cause the form to break. The form_prep function converts HTML so that it can be used safely:

```
<input type="text" name="myform" value="<?php echo form_prep($string); ?>" />
```

Note: If you use any of the form helper functions listed in this page the form values will be prepped automatically, so there is no need to call this function. Use it only if you are creating your own form elements.

set_value()

Permits you to set the value of an input form or textarea. You must supply the field name via the first parameter of the function. The second (optional) parameter allows you to set a default value for the form. Example:

```
<input type="text" name="quantity" value="<?php echo set_value('quantity', '0'); ?>" size="50" />
```

The above form will show "0" when loaded for the first time.

set_select()

If you use a **<select>** menu, this function permits you to display the menu item that was selected. The first parameter must contain the name of the select menu, the second parameter must contain the value of each item, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE).

Example:

```
<select name="myselect">
<option value="one" <?php echo set_select('myselect', 'one', TRUE); ?> >One</option>
```

```
<option value="two" <?php echo set_select('myselect', 'two'); ?> >Two</option>
<option value="three" <?php echo set_select('myselect', 'three'); ?> >Three</option>
</select>
```

set_checkbox()

Permits you to display a checkbox in the state it was submitted. The first parameter must contain the name of the checkbox, the second parameter must contain its value, and the third (optional) parameter lets you set an item as the default (use boolean TRUE/FALSE). Example:

```
<input type="checkbox" name="mycheck" value="1" <?php echo set_checkbox('mycheck', '1'); ?> />
<input type="checkbox" name="mycheck" value="2" <?php echo set_checkbox('mycheck', '2'); ?> />
```

set_radio()

Permits you to display radio buttons in the state they were submitted. This function is identical to the **set_checkbox()** function above.

```
<input type="radio" name="myradio" value="1" <?php echo set_radio('myradio', '1', TRUE); ?> />
<input type="radio" name="myradio" value="2" <?php echo set_radio('myradio', '2'); ?> />
```

HTML Helper

The HTML Helper file contains functions that assist in working with HTML.

- br()
- heading()
- img()
- link_tag()
- nbs()
- ol() and ul()
- meta()
- doctype()

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

br()

Generates line break tags (<br />) based on the number you submit. Example:

```
echo br(3);
```

The above would produce: <br /><br /><br />

heading()

Lets you create HTML <h1> tags. The first parameter will contain the data, the second the size of the heading. Example:

```
echo heading('Welcome!', 3);
```

The above would produce: <h3>Welcome!</h3>

img()

Lets you create HTML <img /> tags. The first parameter contains the image source. Example:

```
echo img('images/picture.jpg');
// gives <img src="http://site.com/images/picture.jpg" />
```

There is an optional second parameter that is a TRUE/FALSE value that specifics if the src should have the page specified by $config['index_page'] added to the address it creates. Presumably, this would be if you were using a media controller.

```
echo img('images/picture.jpg', TRUE);
// gives <img src="http://site.com/index.php/images/picture.jpg" />
```

Additionally, an associative array can be passed to the img() function for complete control over all attributes and values.

```
$image_properties = array(
        'src' => 'images/picture.jpg',
        'alt' => 'Me, demonstrating how to eat 4 slices of pizza at one time',
        'class' => 'post_images',
        'width' => '200',
        'height' => '200',
        'title' => 'That was quite a night',
        'rel' => 'lightbox',
);

img($image_properties);
// <img src="http://site.com/index.php/images/picture.jpg" alt="Me, demonstrating how to eat 4 slices of pizza
at one time" class="post_images" width="200" height="200" title="That was quite a night" rel="lightbox" />
```

link_tag()

Lets you create HTML <link /> tags. This is useful for stylesheet links, as well as other links. The parameters are href, with optional rel, type, title, media and index_page. index_page is a TRUE/FALSE value that specifics if the href should have the page specified by $config['index_page'] added to the address it creates.

```
echo link_tag('css/mystyles.css');
// gives <link href="http://site.com/css/mystyles.css" rel="stylesheet" type="text/css" />
```

Further examples:

```
echo link_tag('favicon.ico', 'shortcut icon', 'image/ico');
// <link href="http://site.com/favicon.ico" rel="shortcut icon" type="image/ico" />

echo link_tag('feed', 'alternate', 'application/rss+xml', 'My RSS Feed');
// <link href="http://site.com/feed" rel="alternate" type="application/rss+xml" title="My RSS Feed" />
```

Additionally, an associative array can be passed to the link() function for complete control over all attributes and values.

```
$link = array(
        'href' => 'css/printer.css',
        'rel' => 'stylesheet',
        'type' => 'text/css',
        'media' => 'print'
);

echo link_tag($link);
// <link href="http://site.com/css/printer.css" rel="stylesheet" type="text/css" media="print" />
```

nbs()

Generates non-breaking spaces (&nbsp;) based on the number you submit. Example:

```
echo nbs(3);
```

The above would produce: &nbsp;&nbsp;&nbsp;

ol() and ul()

Permits you to generate ordered or unordered HTML lists from simple or multi-dimensional arrays. Example:

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

$list = array(
         'red',
         'blue',
         'green',
         'yellow'
         );

$attributes = array(
                'class' => 'boldlist',
                'id'    => 'mylist'
                );

echo ul($list, $attributes);
```

The above code will produce this:

```
<ul class="boldlist" id="mylist">
  <li>red</li>
  <li>blue</li>
  <li>green</li>
  <li>yellow</li>
</ul>
```

Here is a more complex example, using a multi-dimensional array:

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

$attributes = array(
                'class' => 'boldlist',
                'id'    => 'mylist'
```

```
            );

$list = array(
        'colors' => array(
                    'red',
                    'blue',
                    'green'
                ),
        'shapes' => array(
                    'round',
                    'square',
                    'circles' => array(
                                'ellipse',
                                'oval',
                                'sphere'
                                )
                ),
        'moods'   => array(
                    'happy',
                    'upset' => array(
                                'defeated' => array(
                                            'dejected',
                                            'disheartened',
                                            'depressed'
                                            ),
                                'annoyed',
                                'cross',
                                'angry'
                            )
                )
        );


echo ul($list, $attributes);
```

The above code will produce this:

```
<ul class="boldlist" id="mylist">
  <li>colors
    <ul>
      <li>red</li>
      <li>blue</li>
      <li>green</li>
    </ul>
  </li>
  <li>shapes
    <ul>
      <li>round</li>
      <li>suare</li>
      <li>circles
        <ul>
          <li>elipse</li>
          <li>oval</li>
          <li>sphere</li>
        </ul>
      </li>
    </ul>
  </li>
  <li>moods
    <ul>
      <li>happy</li>
      <li>upset
        <ul>
          <li>defeated
            <ul>
              <li>dejected</li>
              <li>disheartened</li>
              <li>depressed</li>
            </ul>
          </li>
          <li>annoyed</li>
          <li>cross</li>
          <li>angry</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

meta()

Helps you generate meta tags. You can pass strings to the function, or simple arrays, or multidimensional ones. Examples:

```
echo meta('description', 'My Great site');
// Generates: <meta name="description" content="My Great Site" />


echo meta('Content-type', 'text/html; charset=utf-8', 'equiv'); // Note the third parameter. Can be "equiv" or
"name"
// Generates: <meta http-equiv="Content-type" content="text/html; charset=utf-8" />


echo meta(array('name' => 'robots', 'content' => 'no-cache'));
// Generates: <meta name="robots" content="no-cache" />


$meta = array(
      array('name' => 'robots', 'content' => 'no-cache'),
      array('name' => 'description', 'content' => 'My Great Site'),
      array('name' => 'keywords', 'content' => 'love, passion, intrigue, deception'),
      array('name' => 'robots', 'content' => 'no-cache'),
      array('name' => 'Content-type', 'content' => 'text/html; charset=utf-8', 'type' => 'equiv')
   );

echo meta($meta);
// Generates:
// <meta name="robots" content="no-cache" />
// <meta name="description" content="My Great Site" />
// <meta name="keywords" content="love, passion, intrigue, deception" />
// <meta name="robots" content="no-cache" />
// <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
```

doctype()

Helps you generate document type declarations, or DTD's. XHTML 1.0 Strict is used by default, but many doctypes are available.

```
echo doctype();
```

```
// <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">

echo doctype('html4-trans');
// <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
```

The following is a list of doctype choices. These are configurable, and pulled from **application/config/doctypes.php**

Doctype	Option	Result
XHTML 1.1	doctype('xhtml11')	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
XHTML 1.0 Strict	doctype('xhtml1-strict')	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
XHTML 1.0 Transitional	doctype('xhtml1-trans')	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
XHTML 1.0 Frameset	doctype('xhtml1-frame')	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
HTML 5	doctype('html5')	<!DOCTYPE html>
HTML 4 Strict	doctype('html4-strict')	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
HTML 4 Transitional	doctype('html4-trans')	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
HTML 4 Frameset	doctype('html4-frame')	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/html4/frameset.dtd">

Inflector Helper

The Inflector Helper file contains functions that permits you to change words to plural, singular, camel case, etc.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

singular()

Changes a plural word to singular. Example:

```
$word = "dogs";
echo singular($word); // Returns "dog"
```

plural()

Changes a singular word to plural. Example:

```
$word = "dog";
echo plural($word); // Returns "dogs"
```

To force a word to end with "es" use a second "true" argument.

```
$word = "pass";
echo plural($word, TRUE); // Returns "passes"
```

camelize()

Changes a string of words separated by spaces or underscores to camel case. Example:

```
$word = "my_dog_spot";
echo camelize($word); // Returns "myDogSpot"
```

underscore()

Takes multiple words separated by spaces and underscores them. Example:

```
$word = "my dog spot";
echo underscore($word); // Returns "my_dog_spot"
```

humanize()

Takes multiple words separated by underscores and adds spaces between them. Each word is capitalized. Example:

```
$word = "my_dog_spot";
echo humanize($word); // Returns "My Dog Spot"
```

Language Helper

The Language Helper file contains functions that assist in working with language files.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

lang('language line', 'element id')

This function returns a line of text from a loaded language file with simplified syntax that may be more desirable for view files than calling **$this->lang->line()**. The optional second parameter will also output a form label for you. Example:

```
echo lang('language_key', 'form_item_id');
// becomes <label for="form_item_id">language_key</label>
```

Number Helper

The Number Helper file contains functions that help you work with numeric data.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

byte_format()

Formats a numbers as bytes, based on size, and adds the appropriate suffix. Examples:

```
echo byte_format(456); // Returns 456 Bytes
echo byte_format(4567); // Returns 4.5 KB
echo byte_format(45678); // Returns 44.8 KB
echo byte_format(456789); // Returns 447.8 KB
echo byte_format(3456789); // Returns 3.3 MB
echo byte_format(12345678912345); // Returns 1.8 GB
echo byte_format(123456789123456789); // Returns 11,228.3 TB
```

Note: The text generated by this function is found in the following language file: language//number_lang.php

Path Helper

The Path Helper file contains functions that permits you to work with file paths on the server.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

set_realpath()

Checks to see if the path exists. This function will return a server path without symbolic links or relative directory structures. An optional second argument will cause an error to be triggered if the path cannot be resolved.

```
$directory = '/etc/passwd';
echo set_realpath($directory);
// returns "/etc/passwd"

$non_existent_directory = '/path/to/nowhere';
echo set_realpath($non_existent_directory, TRUE);
// returns an error, as the path could not be resolved

echo set_realpath($non_existent_directory, FALSE);
// returns "/path/to/nowhere"
```

Security Helper

The Security Helper file contains security related functions.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

xss_clean()

Provides Cross Site Script Hack filtering. This function is an alias to the one in the Input class. More info can be found there.

dohash()

Permits you to create SHA1 or MD5 one way hashes suitable for encrypting passwords. Will create SHA1 by default. Examples:

```
$str = dohash($str); // SHA1

$str = dohash($str, 'md5'); // MD5
```

strip_image_tags()

This is a security function that will strip image tags from a string. It leaves the image URL as plain text.

```
$string = strip_image_tags($string);
```

encode_php_tags()

This is a security function that converts PHP tags to entities. Note: If you use the XSS filtering function it does this automatically.

```
$string = encode_php_tags($string);
```

Smiley Helper

The Smiley Helper file contains functions that let you manage smileys (emoticons).

Loading this Helper

This helper is loaded using the following code:

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

Overview

The Smiley helper has a renderer that takes plain text simileys, like **:-)** and turns them into a image representation, like

It also lets you display a set of smiley images that when clicked will be inserted into a form field. For example, if you have a blog that allows user commenting you can show the smileys next to the comment form. Your users can click a desired smiley and with the help of some JavaScript it will be placed into the form field.

Clickable Smileys Tutorial

Here is an example demonstrating how you might create a set of clickable smileys next to a form field. This example requires that you first download and install the smiley images, then create a controller and the View as described.

Important: Before you begin, please download the smiley images and put them in a publicly accessible place on your server. This helper also assumes you have the smiley replacement array located at **application/config/smileys.php**

The Controller

In your **application/controllers/** folder, create a file called **smileys.php** and place the code below in it.

Important: Change the URL in the **get_clickable_smileys()** function below so that it points to your **smiley** folder.

You'll notice that in addition to the smiley helper we are using the Table Class.

```
<?php

class Smileys extends Controller {

	function Smileys()
	{
		parent::Controller();
	}

	function index()
	{
		$this->load->helper('smiley');
		$this->load->library('table');

		$image_array = get_clickable_smileys('http://example.com/images/smileys/', 'comments');

		$col_array = $this->table->make_columns($image_array, 8);

		$data['smiley_table'] = $this->table->generate($col_array);

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

}
?>
```

In your **application/views/** folder, create a file called **smiley_view.php** and place this code in it:

```
<html>
<head>
<title>Smileys</title>

<?php echo smiley_js(); ?>

</head>
<body>

<form name="blog">
<textarea name="comments" id="comments" cols="40" rows="4"></textarea>
</form>

<p>Click to insert a smiley!</p>

<?php echo $smiley_table; ?>

</body>
</html>
```

When you have created the above controller and view, load it by visiting **http://www.example.com/index.php/smileys/**

Field Aliases

When making changes to a view it can be inconvenient to have the field id in the controller. To work around this, you can give your smiley links a generic name that will be tied to a specific id in your view.

```
$image_array = get_smiley_links("http://example.com/images/smileys/", "comment_textarea_alias");
```

To map the alias to the field id, pass them both into the smiley_js function:

```
$image_array = smiley_js("comment_textarea_alias", "comments");
```

Function Reference

get_clickable_smileys()

Returns an array containing your smiley images wrapped in a clickable link. You must supply the URL to your smiley folder and a field id or field alias.

```
$image_array = get_smiley_links("http://example.com/images/smileys/", "comment");
```

Note: Usage of this function without the second parameter, in combination with js_insert_smiley has been deprecated.

smiley_js()

Generates the JavaScript that allows the images to be clicked and inserted into a form field. If you supplied an alias instead of an id when generating your smiley links, you need to pass the alias and corresponding form id into the function. This function is designed to be placed into the <head> area of your web page.

```
<?php echo smiley_js(); ?>
```

Note: This function replaces js_insert_smiley, which has been deprecated.

parse_smileys()

Takes a string of text as input and replaces any contained plain text smileys into the image equivalent. The first parameter must contain your string, the second must contain the URL to your smiley folder:

```
$str = 'Here are some simileys: :-) ;-)'; $str = parse_smileys($str, "http://example.com/images/smileys/"); echo
$str;
```

String Helper

The String Helper file contains functions that assist in working with strings.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

random_string()

Generates a random string based on the type and length you specify. Useful for creating passwords or generating random hashes.

The first parameter specifies the type of string, the second parameter specifies the length. The following choices are available:

- **alnum**: Alpha-numeric string with lower and uppercase characters.
- **numeric**: Numeric string.
- **nozero**: Numeric string with no zeros.
- **unique**: Encrypted with MD5 and uniqid(). Note: The length parameter is not available for this type. Returns a fixed length 32 character string.

Usage example:

```
echo random_string('alnum', 16);
```

alternator()

Allows two or more items to be alternated between, when cycling through a loop. Example:

```
for ($i = 0; $i < 10; $i++)
{
    echo alternator('string one', 'string two');
}
```

You can add as many parameters as you want, and with each iteration of your loop the next item will be returned.

```
for ($i = 0; $i < 10; $i++)
{
    echo alternator('one', 'two', 'three', 'four', 'five');
}
```

Note: To use multiple separate calls to this function simply call the function with no arguments to re-initialize.

repeater()

Generates repeating copies of the data you submit. Example:

```
$string = "\n";
echo repeater($string, 30);
```

The above would generate 30 newlines.

reduce_double_slashes()

Converts double slashes in a string to a single slash, except those found in http://. Example:

```
$string = "http://example.com//index.php";
echo reduce_double_slashes($string); // results in "http://example.com/index.php"
```

trim_slashes()

Removes any leading/trailing slashes from a string. Example:

```
$string = "/this/that/theother/";
echo trim_slashes($string); // results in this/that/theother
```

reduce_multiples()

Reduces multiple instances of a particular character occuring directly after each other. Example:

```
$string="Fred, Bill,, Joe, Jimmy";
$string=reduce_multiples($string,","); //results in "Fred, Bill, Joe, Jimmy"
```

The function accepts the following parameters:

```
reduce_multiples(string: text to search in, string: character to reduce, boolean: whether to remove the
character from the front and end of the string)
```

The first parameter contains the string in which you want to reduce the multiplies. The second parameter contains the character you want to have reduced. The third parameter is FALSE by default; if set to TRUE it will remove occurences of the character at the beginning and the end of the string. Example:

```
$string=",Fred, Bill,, Joe, Jimmy,";
$string=reduce_multiples($string, ", ", TRUE); //results in "Fred, Bill, Joe, Jimmy"
```

quotes_to_entities()

Converts single and double quotes in a string to the corresponding HTML entities. Example:

```
$string="Joe's \"dinner\"";
$string=quotes_to_entities($string); //results in "Joe&#39;s &quot;dinner&quot;"
```

strip_quotes()

Removes single and double quotes from a string. Example:

```
$string="Joe's \"dinner\"";
$string=strip_quotes($string); //results in "Joes dinner"
```

Text Helper

The Text Helper file contains functions that assist in working with text.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

word_limiter()

Truncates a string to the number of **words** specified. Example:

```
$string = "Here is a nice text string consisting of eleven words.";

$string = word_limiter($string, 4);

// Returns: Here is a nice…
```

The third parameter is an optional suffix added to the string. By default it adds an ellipsis.

character_limiter()

Truncates a string to the number of **characters** specified. It maintains the integrity of words so the character count may be slightly more or less then what you specify. Example:

```
$string = "Here is a nice text string consisting of eleven words.";

$string = character_limiter($string, 20);

// Returns: Here is a nice text string…
```

The third parameter is an optional suffix added to the string, if undeclared this helper uses an ellipsis.

ascii_to_entities()

Converts ASCII values to character entities, including high ASCII and MS Word characters that can cause problems when used in a web page, so that they can be shown consistently regardless of browser settings or stored reliably in a database. There is some dependence on your server's supported character sets, so it may not be 100% reliable in all cases, but for the most part it should correctly identify characters outside the normal range (like accented characters). Example:

```
$string = ascii_to_entities($string);
```

entities_to_ascii()

This function does the opposite of the previous one; it turns character entities back into ASCII.

word_censor()

Enables you to censor words within a text string. The first parameter will contain the original string. The second will contain an array of words which you disallow. The third (optional) parameter can contain a replacement value for the words. If not specified they are replaced with pound signs: ####. Example:

```
$disallowed = array('darn', 'shucks', 'golly', 'phooey');

$string = word_censor($string, $disallowed, 'Beep!');
```

highlight_code()

Colorizes a string of code (PHP, HTML, etc.). Example:

```
$string = highlight_code($string);
```

The function uses PHP's highlight_string() function, so the colors used are the ones specified in your php.ini file.

highlight_phrase()

Will highlight a phrase within a text string. The first parameter will contain the original string, the second will contain the phrase you wish to highlight. The third and fourth parameters will contain the opening/closing HTML tags you would like the phrase wrapped in. Example:

```
$string = "Here is a nice text string about nothing in particular.";

$string = highlight_phrase($string, "nice text", '<span style="color:#990000">', '</span>');
```

The above text returns:

Here is a nice text string about nothing in particular.

word_wrap()

Wraps text at the specified **character** count while maintaining complete words. Example:

```
$string = "Here is a simple string of text that will help us demonstrate this function.";

echo word_wrap($string, 25);

// Would produce:

Here is a simple string
of text that will help
us demonstrate this
function
```

Typography Helper

The Typography Helper file contains functions that help your format text in semantically relevant ways.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

auto_typography()

Formats text so that it is semantically and typographically correct HTML. Please see the Typography Class for more info.

Usage example:

```
$string = auto_typography($string);
```

Note: Typographic formatting can be processor intensive, particularly if you have a lot of content being formatted. If you choose to use this function you may want to consider caching your pages.

nl2br_except_pre()

Converts newlines to <br /> tags unless they appear within <pre> tags. This function is identical to the native PHP **nl2br()** function, except that it ignores <pre> tags.

Usage example:

```
$string = nl2br_except_pre($string);
```

URL Helper

The URL Helper file contains functions that assist in working with URLs.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

site_url()

Returns your site URL, as specified in your config file. The index.php file (or whatever you have set as your site **index_page** in your config file) will be added to the URL, as will any URI segments you pass to the function.

You are encouraged to use this function any time you need to generate a local URL so that your pages become more portable in the event your URL changes.

Segments can be optionally passed to the function as a string or an array. Here is a string example:

```
echo site_url("news/local/123");
```

The above example would return something like: http://example.com/index.php/news/local/123

Here is an example of segments passed as an array:

```
$segments = array('news', 'local', '123');

echo site_url($segments);
```

base_url()

Returns your site base URL, as specified in your config file. Example:

```
echo base_url();
```

current_url()

Returns the full URL (including segments) of the page being currently viewed.

uri_string()

Returns the URI segments of any page that contains this function. For example, if your URL was this:

```
http://some-site.com/blog/comments/123
```

The function would return:

```
/blog/comments/123
```

index_page()

Returns your site "index" page, as specified in your config file. Example:

```
echo index_page();
```

anchor()

Creates a standard HTML anchor link based on your local site URL:

```
<a href="http://example.com">Click Here</a>
```

The tag has three optional parameters:

```
anchor(uri segments, text, attributes)
```

The first parameter can contain any segments you wish appended to the URL. As with the **site_url()** function above, segments can be a string or an array.

Note: If you are building links that are internal to your application do not include the base URL (http://...). This will be added automatically from the information specified in your config file. Include only the URI segments you wish appended to the URL.

The second segment is the text you would like the link to say. If you leave it blank, the URL will be used.

The third parameter can contain a list of attributes you would like added to the link. The attributes can be a simple string or an associative array.

Here are some examples:

```
echo anchor('news/local/123', 'title="My News"');
```

Would produce: <a href="http://example.com/index.php/news/local/123" title="My News">My News</a>

```
echo anchor('news/local/123', 'My News', array('title' => 'The best news!'));
```

Would produce: <a href="http://example.com/index.php/news/local/123" title="The best news!">My News</a>

anchor_popup()

Nearly identical to the **anchor()** function except that it opens the URL in a new window. You can specify JavaScript window attributes in the third parameter to control how the window is opened. If the third parameter is not set it will simply open a new window with your own browser settings. Here is an example with attributes:

```
$atts = array(
          'width'      => '800',
          'height'     => '600',
          'scrollbars' => 'yes',
          'status'     => 'yes',
          'resizable'  => 'yes',
          'screenx'    => '0',
          'screeny'    => '0'
        );

echo anchor_popup('news/local/123', 'Click Me!', $atts);
```

Note: The above attributes are the function defaults so you only need to set the ones that are different from what you need. If you want the function to use all of its defaults simply pass an empty array in the third parameter:

```
echo anchor_popup('news/local/123', 'Click Me!', array());
```

mailto()

Creates a standard HTML email link. Usage example:

```
echo mailto('me@my-site.com', 'Click Here to Contact Me');
```

As with the **anchor()** tab above, you can set attributes using the third parameter.

safe_mailto()

Identical to the above function except it writes an obfuscated version of the mailto tag using ordinal numbers written with JavaScript to help prevent the email address from being harvested by spam bots.

auto_link()

Automatically turns URLs and email addresses contained in a string into links. Example:

```
$string = auto_link($string);
```

The second parameter determines whether URLs and emails are converted or just one or the other. Default behavior is both if the parameter is not specified. Email links are encoded as safe_mailto() as shown above.

Converts only URLs:

```
$string = auto_link($string, 'url');
```

Converts only Email addresses:

```
$string = auto_link($string, 'email');
```

The third parameter determines whether links are shown in a new window. The value can be TRUE or FALSE (boolean):

```
$string = auto_link($string, 'both', TRUE);
```

url_title()

Takes a string as input and creates a human-friendly URL string. This is useful if, for example, you have a blog in which you'd like to use the title of your entries in the URL. Example:

```
$title = "What's wrong with CSS?";

$url_title = url_title($title);

// Produces: Whats-wrong-with-CSS
```

The second parameter determines the word delimiter. By default dashes are used. Options are: **dash**, or **underscore**:

```
$title = "What's wrong with CSS?";

$url_title = url_title($title, 'underscore');

// Produces: Whats_wrong_with_CSS
```

The third parameter determines whether or not lowercase characters are forced. By default they are not. Options are boolean **TRUE**/**FALSE**:

```
$title = "What's wrong with CSS?";

$url_title = url_title($title, 'underscore', TRUE);

// Produces: whats_wrong_with_css
```

prep_url()

This function will add **http://** in the event it is missing from a URL. Pass the URL string to the function like this:

```
$url = "example.com";

$url = prep_url($url);
```

redirect()

Does a "header redirect" to the local URI specified. Just like other functions in this helper, this one is designed to redirect to a local URL within your site. You will **not** specify the full site URL, but rather simply the URI segments to the controller you want to direct to. The function will build the URL based on your config file values.

The optional second parameter allows you to choose between the "location" method (default) or the "refresh" method. Location is faster, but on Windows servers it can sometimes be a problem. The optional third parameter allows you to send a specific HTTP Response Code - this could be used for example to create 301 redirects for search engine purposes. The default Response Code is 302. The third parameter is *only* available with 'location' redirects, and not 'refresh'. Examples:

```
if ($logged_in == FALSE)
{
    redirect('/login/form/', 'refresh');
}

// with 301 redirect
redirect('/article/13', 'location', 301);
```

Note: In order for this function to work it must be used before anything is outputted to the browser since it utilizes server headers.
Note: For very fine grained control over headers, you should use the Output Library's set_header() function.

XML Helper

The XML Helper file contains functions that assist in working with XML data.

Loading this Helper

This helper is loaded using the following code:

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

The following functions are available:

xml_convert('string')

Takes a string as input and converts the following reserved XML characters to entities:

Ampersands: &
Less then and greater than characters: < >
Single and double quotes: ' "
Dashes: -

This function ignores ampersands if they are part of existing character entities. Example:

```
$string = xml_convert($string);
```