



# Kohana 101

Kohana 101 is a quick introduction to the Kohana PHP5 framework.

## Table of contents

Overview .....	2
Setup .....	5
Configuration.....	6
Show Me .....	7
Kohana Magic.....	10
The Kohana Way .....	14
What Next? .....	33

© Oscar Bajner 2008  
Second Edition : October 9 2008  
First Edition : February 18 2008

## #1: Overview

Kohana is an open source framework for developing applications using PHP.

### Not Another Framework?

Yes! There are so many frameworks available, you could spend your life just trying them out. So what makes Kohana worth your time?

If you answer "Yes" to one of these questions:

1. I never use frameworks, PHP is all I need.
2. I use PHP framework X, but find it too restrictive, bloated, incomprehensible.
3. I use Django, Rails, ASP.NET, Java: My life could not be better. Could it?
4. My boss told me to research frameworks, what's a framework?

Developing websites and web applications involves a lot of repetitive work. Programmers, being lazy sods, have responded by making the software do the heavy lifting. Usually this means writing libraries and components which simplify and automate certain tasks.

Observant programmers noticed that developing web apps involve lots of the same type of repetitive work. Observant programmers, being clever, have responded by making the software implement a pattern, which simplifies and automates the solving of well defined problems.

A very common problem for many websites, is producing dynamic output, often based on user input. The most common pattern which solves this problem is known as *Model- View-Controller* (MVC). The concept is that an application should organise around a logical separation of operations. The *Controller* "controls" things, deciding which resources are used, what data should be modified and when output is displayed. The *Model* "models" the data, abstracting the retrieving, storing, updating and validating of data. The final output is presented in the *View*, which is a "view" of what just happened.

A Framework can be:

1. An implementation of a problem solving pattern.
2. An organised collection of reusable software components.
3. Both.

Kohana implements the MVC pattern(1), by linking the *Controller* (through methods) to a logical representation of the *Uniform Resource Identifier* (through segments). This means, that a given URI has a *conventional* meaning to Kohana, and a particular meaning to an application written in Kohana. Model and View are implemented optionally, it is highly recommended to use a Model and a View, but only the Controller is required.

Kohana implements reusable components(2), via an organised hierarchy of *Libraries and Helpers*. These components are utilised by Controllers, Models and Views, which reside within the same hierarchical structure. This arrangement allows for exceptional flexibility. Kohana components may be overridden by another, located higher in the order, they may be extended, or replaced by custom components, most importantly, they may be modularised, by function or organisation. Finally, certain Libraries are implemented via Drivers, combining a uniform application programming interface, with a choice of *backend*.

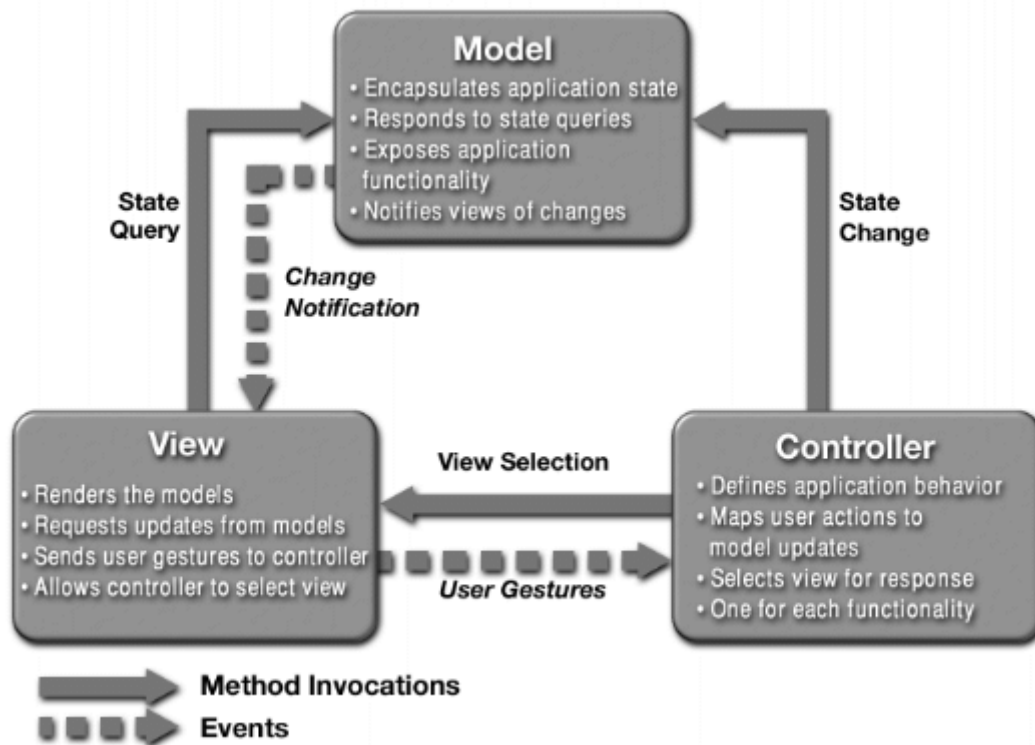
*There are times in life when it is necessary to panic. Now is not one of them, be of good cheer, and read on.*

### Model View Controller

There are different opinions on what constitutes **TRUE MVC**. Frankly, it does not matter. What interests me is knowing a good methodology to design web applications. The essence of *MVC* is about isolating an application's user interface, from it's data, and the presentation of that data. The rationale is that application development is simpler and more efficient when the presentation of data is separate from the definition and manipulation of the data.

In my opinion, Kohana does not **implement** MVC of any kind. That statement may seem contradictory, given the Kohana sales talk. So let's take a look at a generic *Model-View-Controller* implementation, and then see what Kohana thinks about it.

A diagram of *MVC*



The solid arrows indicate a direct interaction (Method invocations). Note that the View pulls data from the Model (renders the model). The Controller directs the Model to update state. The Controller selects the View to be presented.

The dashed arrows represent indirect association. (Events) Note that the View notifies the Controller about user interactions. The Model notifies the View of state changes.

All of these interactions occur in a repetitive-cyclic manner, initiated by user input. Note that the Model never communicates with the Controller.

Confused? Yes, I certainly was!

Here's what you need to remember if you are doing *MVC*

- The View is about what the user **sees**.
- The Controller is about what the user **does**.
- The Model is about what the user **gets**.

Earlier I said Kohana does not **implement** *MVC*. If you want to apply the *MVC* pattern in your application, then Kohana makes it very easy.

I repeat, the **only** component **required** to build a complete Kohana application is the Controller. In Kohana the Controller can directly access data, and directly present that data to the browser.

If you are building interactive web applications, it makes a lot of sense to use *MVC*. Kohana enables you to implement *MVC* in a sane and consistent manner.

The simplest barebones Kohana *MVC* application will be based upon the logic of it's user interaction. When the user clicks link A, then controller X must do something. If link A requires data to be fetched, then controller X will tell model Z to fetch it. Controller A will pass that data to View Q which will display it, and wait for further user input.

With Kohana, your application can have one Controller, many Views, and one Model. It could have many Controllers, each with associated Views and Models.

It could also enforce a much stricter pattern, which strives to emulate generic *MVC* and implements Events

## Kohana 101

which allow for communication between Views and Models without Controller intervention. You can make it as complex or as simple as you need.

## #2: Setup

Let's start by installing Kohana on your development machine.

Currently, Kohana is at version 2.2.x. Go to <http://kohanaphp.com/download> and download the latest archive.

At this time, we will not make use of any modules. Select your required language. If your language is not listed, select *English (US)*. Select *Markdown* and *SwiftMailer* from the vendor tools section. Click the "Download Kohana!" button.

Unzip *Kohana\_v2.2.x.zip* into a temporary folder. Have a look inside the folder, you should see:

```
temp_folder
  Kohana_v2.2
    index.php
    example.htaccess
    -- application
    -- modules
    -- system
```

Put the files into your webserver document root. On *Apache* that's normally *htdocs* or */var/www*. Create a new folder in document root, called "kohana". Now copy the contents of *Kohana\_v2.2* into */var/www/kohana* or *htdocs/kohana*

Your Kohana framework is now installed!

Test it out. Point your browser at <http://localhost/kohana> and load the page. If all goes well, you should see the Kohana welcome page in your browser.

Something went wrong ... ? This can happen.

Check permissions on *application/logs*. The web server user must have write permission to the folder. On Unix the web server must belong to a group with write permissions on the folder, or make the folder world writable, *chmod 777 application/logs*

You need an operating system that supports Unicode. PHP must be version 5.2 or higher. Extensions *PCRE* and *iconv* are required. *SPL* must be enabled. Your system *hosts* file must have an entry that maps *localhost* to 127.0.0.1

### #3: Configuration

Kohana favours *convention* over *configuration*. What does this mean? Kohana expects that certain things be named in a certain way, and certain resources to be located in a specific place. That allows Kohana to “Just Work” if normal conventions are followed. To change conventional behaviour, you need to tell Kohana about it, by configuration.

One important convention is that **every** Kohana application must have a configuration file called *application/config/config.php*. It may not be renamed or relocated.

Navigate to */htdocs/kohana/application/config* and open file *config.php* in a text editor.<sup>1</sup> Note that all Kohana configurations are maintained as PHP array entries. The first entry is the *site\_domain* which defaults to a path:

```
$config['site_domain'] = '/kohana/';
```

Note the trailing */* on the path.

In development, it's often easier to create your Kohana projects as sub-folders like *www/html/project\_folder* which would be configured as */project\_folder/*. If you specify a domain, then a full URL will be used eg. *http://domain.tld/kohana/*. In production, the config entry would include the domain and look like:

```
$config['site_domain'] = 'www.domain.tld/';
```

We will want to use a full URL and not just a path, so change the *site\_domain* as follows:

```
$config['site_domain'] = 'localhost/kohana/';
```

Save the changes, and we are finished configuration, for now.<sup>2</sup> So it's Kohana time!

---

1. Kohana files are UTF-8 encoded, without a Byte Order Mark.  
2. We will make some configuration changes later on.

## #4: Show me. Don't tell me.

Let's learn by doing here. Point your browser back at <http://localhost/kohana> and reload the page, so you should see the Kohana welcome page.

So what is happening, that makes this work? First, take a look in your *project* folder:

```
index.php
example.htaccess
application
  -- cache
  -- config
  -- controllers
  -- helpers
  -- hooks
  -- libraries
  -- logs
  -- models
  -- views
modules
system
```

Every Kohana application has a front controller, *index.php* which must be located within the document root of your webserver. It can be in a sub-folder, but it must be publicly accessible via a browser. *index.php* is processed every time you access your website from a URI such as <http://localhost/kohana>

When PHP processes *index.php* the Kohana framework is invoked, system files are loaded, and the URI is inspected to determine what actions need to be taken. These actions correspond to *Controllers* and their associated methods.

Let's revisit URIs. You will be familiar with Google search, you enter some keywords and hit the search button. I'm looking for information on PHP, so I enter "PHP". Look at the URI in your browser. It has <http://www.google.com/search?hl=en&q=php&btnG=Search&meta=> When you hit the search button, information is passed to the google server, asking it to execute a search and providing a search term.

To use Kohana, it is fundamental to understand that your application will be built around the URIs that correspond to resources and operations your application provides. How does Kohana use the URI? Everything after *index.php* is a **segment**.

```
| domain information | front | segments 1/2/3/...
URI: http://localhost/kohana/index.php/welcome/search/php
      ^             ^     ^     ^
      |             1     2     3
      [Front Controller] [ segments ]
                          1 [Controller]
                           2 [Method]
                            3 [Arg 1]
```

In the first example the URI was <http://localhost/kohana>. So where are the segments? Once again, Kohana conventions explain:

- <http://localhost/kohana> : No segments, so a default controller is used.
- <http://localhost/kohana/index.php> : Still no segments, so same as before.
- <http://localhost/kohana/index.php/welcome> : Only one segment, which is segment 1. This corresponds to a controller called "Welcome"
- <http://localhost/kohana/index.php/welcome/search/php> : Three segments given:
  - Segment 1 is "welcome" and corresponds to controller "Welcome"
  - Segment 2 is "search" and corresponds to method "search()" in controller "Welcome"
  - Segment 3 is "php" and corresponds to argument 1 passed to method "search()" in controller "Welcome"

The default controller is named "Welcome" It is possible to configure this differently.

The default method is called *index()* and is assumed if no segment, or only segment 1 is given.

## Kohana 101

Note:

- A URI may have any number of segments (limited by URL length permitted by browser)
- Segments 3 and higher are automatically passed to the appropriate method as arguments.
- You cannot pass arguments to the *index()* method

Let's create our own Controller. Copy the code below into your editor, and save the file as *application/controllers/test.php*

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Test_Controller extends Controller {  
    public function index()  
    {  
        echo 'goodbye world';  
    }  
}
```

*Controller* is a Kohana class. All application controllers will *extend* this class. Your application may have just one controller, or it may have many, depending on how it is organised.

A single method, *index()* is defined. If the controller is invoked without a method segment, this function is called.

Point your browser at <http://localhost/kohana/index.php/test> You should see "goodbye world"

Let's create our own View. Save the code below into a file called *application/views/test\_content.php*

```
<h2>Good Bye Cruel World</h2>
```

*View* is a Kohana system library. It implements easy manipulation of View objects. *application/views/test\_content.php* is a View file. A View file consists of the information you want to output to a browser. The View file is not the same as the View object. The View file *test\_content.php* is passed to the View library as an argument at instantiation.

A View file can contain just XHTML, for outputting a static type page. It can contain the complete XHTML required to output a valid page, or it can contain a fragment of XHTML, for inclusion into another View file. Note, a View file may also contain style sheet information, or javascript.

To use a View, we instantiate a View object. Edit *application/controllers/test.php* and add this code:

```
public function bye()  
{  
    $test = new View('test_content');  
    $test->render(TRUE);  
}
```

Quite a lot is happening here. We're telling Kohana: Use the Class *View* in *system/libraries/view.php* to create a View object for me, from a template called *application/views/test\_content.php*. We assign the object to variable *\$test*. To display the contents of the template in the browser, we invoke method, *render()* of class *View*.

Reload <http://localhost/kohana/index.php/test/bye> in the browser.

You should see

```
Good Bye Cruel World
```

A View file can contain placeholders for dynamic data. These are properties assigned to the View object, and the information is passed to a variable of the **same name** in the View file, for output in the browser. Lets add some dynamic information to our View file.

Edit *application/controllers/test.php* and make sure the *index()* method contains only the code below.

## Kohana 101

```
$test = new View('test_content');  
$test->message = 'Kohana rules on '  
$test->now = date(DATE_RFC822);  
$test->render(TRUE);
```

Edit `application/views/test_content.php` and make sure it contains the code below.

```
<h2>Welcome!</h2>  
<p><?php echo $message.$now ?></p>  
<hr/>
```

Load up `http://localhost/kohana/index.php/test` in your browser. The message now includes some dynamic information that changes with each page load.

How did the output actually go to the browser? `$test->render(TRUE)` is the View library method which displays data. By default, the `render()` method does not display data, so you can assign the output to a variable.

## #5: Kohana Magic

You now have some idea about how Kohana goes about it's business. Let's explore the framework now, we want to know a few important things:

1. What are Kohanan things called
2. Where do I find these things, and how does Kohana find them
3. How should I name my stuff
4. How do I use Kohanan things
5. Where do I put my stuff

### 1 : Kohana System Components

The most important Kohanans<sup>3</sup> are *Controllers, Libraries, Helpers, Modules, Models* and *Views*

- **Controllers** : Are the heart of your application, implementing it's logic.
- **Libraries** : Provide the functionality that drives your application; Database access, Sessions, Cacheing etc.
- **Helpers** : Assist you with common tasks; Formatting HTML, Sending email, Validating, Creating HTML forms etc.
- **Modules** : Assist in grouping common components or functionality: Authentication, Handling media etc.
- **Models** : Separate the data access from the logic and presentation.
- **Views** : Simplifies the organisation of information for display.

### 2 : Kohana System Layout

There are three layers to Kohana; *Application, Module, System*. A very important concept to understand in Kohana is the *cascade*.

The *cascade* refers to where and how Kohana searches for resources. A specific naming convention and directory layout is enforced. A specific search order is enforced.

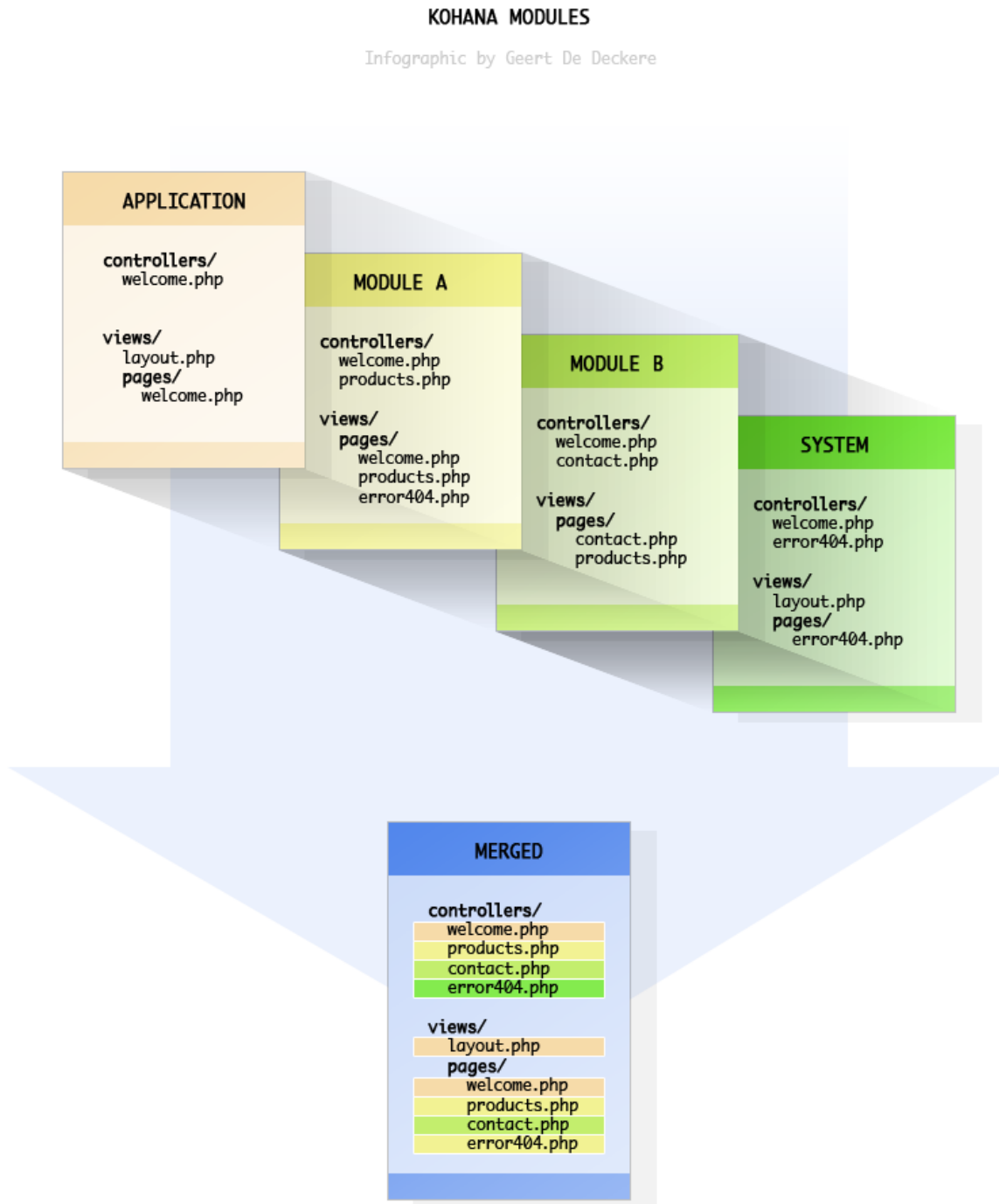
```
Search ----->
(1) application
  -- controllers
  -- helpers
  -- libraries
  -- models
  -- views
----->
(2) modules
  -- controllers
  -- helpers
  -- libraries
  -- models
  -- views
----->
(3) system
  -- controllers
  -- helpers
  -- libraries
  -- models
  -- views
```

When Kohana looks for a resource, it first searches in *application*, if nothing is found, *modules* is searched (if the module path is enabled) and finally *system* is searched.

---

3. Actually, there are Core Kohanans, who are more important.

A picture of the *cascade*



The *cascade* allows for transparent extension of the framework. Example:

When your application controller instantiates the View library `$welcome = new View('welcome')` Kohana first looks in *application/libraries* for a file called *view.php*. If it does not find one, it will use the default in *system*. It is possible to replace almost any system library<sup>4</sup>, merely by placing your own version in *application/libraries*.

---

4. Libraries which implement Drivers cannot easily be replaced

In Kohana, resources such as Controllers and Views may be **nested**. That means you may nest these resources in sub-folders, to any depth. Kohana will automatically locate your resource. Example  
We want to locate the welcome View in *application/views/pages*

```
application
-- views
  -- pages
    welcome.php
```

The View will now be accessed as follows:

```
$welcome = new View('pages/welcome');
```

### 3 : Kohana System Naming

Kohana implements a namespace, so controllers, models and views can have the same name.

Kohana convention is to use *under\_score* naming.

- Controllers  
Class name: *Name\_Controller* **extends** *Controller* (First letters uppercase)  
File name: *controllers/controller.php* (lowercase)
- Helpers  
Class name: *name\_Core* (First letter lowercase)  
File name: *helpers/helper.php* (lowercase)
- Libraries  
Class name: *Name\_Core* (First letters uppercase)  
File name: *libraries/Library.php* (First letter uppercase)
- Models  
Class name: *Name\_Model* **extends** *Model* (First letters uppercase)  
File name: *models/model.php* (lowercase)
- Views  
Class name: It's not a class.  
File name: *views/view.php* (lowercase)

### 4 : Kohana System Usage

Using Kohana components is just like using any PHP Objected Oriented component. Extend class, Instantiate object, assign object properties, apply class methods.

Controllers are not instantiated manually, the system does that, you only extend the class

Helpers have static methods, they do not have to be instantiated, just call the class method. Example:  
*url::redirect('http://www.whitehouse.gov');* will emit a location header.

You can use a helper **anywhere**, controller, model, view.

Models must be instantiated. The object must be assigned. *\$this->model = new Model\_Class\_Name('model\_file\_name')* A model should be instantiated in the controller. Note:  
A model does not have access to library objects instantiated by the controller, you need to instantiate a new object, or use *Kohana::instance()->library->method()*

Libraries must be instantiated. The object must be assigned. *\$this->library = new Library\_Class\_Name('Library\_file\_name')* A library should be instantiated in the controller.

Some system methods are static. This allows for convenient access to the system, you do not need to instantiate anything to use them. Examples are *Kohana::config('item')* which returns a config entry.  
*Kohana::debug(\$variable)* which pretty prints any variable for debugging.

## 5 : Kohana application Layout

The default install is to place all files below the webserver document root. Kohana is designed so that you can easily move the *application*, *modules* and *system* folders above document root. The location of *application*, *modules*, *system* is defined in each application's front controller, *index.php*.

This allows for improved security, maintainability and organization. Multiple *applications* can be run from a single *system* install.

Put your controllers in *application/controllers* You can nest them.

Put your models in *application/models* You can nest them.

Put your views in *application/views* You can nest them.

Put your static resources like style sheets, images and javascript into separate folders. It is easier to use relative file paths, so place the folders relative to the front controller, *index.php*

A recommended Kohana layout would look like this

```
Your Root or Home folder on system
/home/user or similar
  applications
    app_1
    app_2
  system
    2.2
    2.3
  modules
    2.2
    2.3
-----
Document Root of Webserver
/home/user/htdocs or similar
domain_name_for_app_1
  index.php
  .htaccess
  media (contains all your static resources)
    css
    js
    images
```

## #6: The Kohana Way

Let's do a quick Kohana website.

Our Website will be quite spartan but will comprise the basis for any website built with Kohana.

Here's the drill:

1. Build a website for my new business (*L33t Accessories*)
2. I want a home page, about page, a product and prices page and a contact page
3. The product page must list all our products, and their prices
4. I want it by lunchtime today!

Bosses! You gotta love them!

Okay, Kohanians, off we go ...

### Layout your new site

Let's put all the things we'll need into place

So, we'll need a template right? For the basic layout. And a style sheet for that. And we'll need views for each page, and Controllers. And we need to get the products and prices details from somewhere.

We've got *localhost/kohana* already installed, so we'll just use that.

Let's hack up our test controller and view for our home page. First thing we need is to output a proper XHTML page. Edit *applications/views/test\_content.php* to contain this code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<title><?php echo html::specialchars($title) ?></title>

</head>
<body>
<h1>L33t Str33t</h1>
<h2>Home</h2>
<p><?php echo $message.$now ?></p>
<hr/>

</body>
</html>
```

Edit *controllers/test.php* so it looks like the code below, and save the file as *application/controllers/home.php*

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Home_Controller extends Controller {

    public function index()
    {
        $home = new View('test_content');
        $home->title = 'L33t Str33t:Home';
        $home->message = 'Kohana rules on ';
        $home->now = date(DATE_RFC822);
        $home->render(TRUE);
    }
}
```

Load up `http://localhost/kohana/index.php/home` Check that the title is correct, and that the headings are changed.

Now we create a template View. Simply save `application/views/test_content.php` as `template.php`. All our view content will be displayed using this template.

## Use System Templating

Now for some *Kohana Goodness!* Edit `application/controllers/home.php` as per the code below. Then reload `http://localhost/kohana/index.php/home`

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Home_Controller extends Template_Controller {
```

```
    public function index()
```

```
    {
```

```
        $this->template->title = 'Home::L33t Accessories';
```

```
        $this->template->message = 'Kohana rules on ';
```

```
        $this->template->now = date(DATE_RFC822);
```

```
    }
```

```
}
```

We are using a system controller class, `Template_Controller` to automatically render our template view!

Kohana convention requires us to name the view, `template.php` because `Template_Controller` is instantiating a View by that name. You can change this by copying `system/controllers/template.php` to `application/controllers/template.php` and changing the name of the "template" for example, you could call it "layout".

Our `Home_Controller` **extends** `Template_Controller` so it inherits the methods and properties of the parent.

Why use a template controller? Because it helps you organise functionality in your application. Instead of doing everything in one controller, you group functions into "Task" controllers, extending from the parent as necessary.

This is a **recommended**<sup>5</sup> way to structure your application controllers.

1. `Foo_Controller extends Website_Controller >>>` URI-specific controller, eg: `Blog_Controller`
2. `Website_Controller extends Template_Controller >>>` "Base" controller to load default application resources: Database, Session, Auth...
3. `Template_Controller extends Controller >>>` Standardized template loading

So let's follow the recommendation, and create a website controller. Save the code below as `application/controllers/website.php`

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Website_Controller extends Template_Controller {
```

```
    public function __construct()
```

```
    {
```

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

```
    }
```

```
}
```

Edit `application/controllers/home.php` and change the class declaration to read `class Home_Controller extends Website_Controller`

---

5. A *Shadowhand*<sup>TM</sup> production.

## Kohana 101

Now, we get rid of the junk in our template, and add in a menu.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<title><?php echo html::specialchars($title) ?></title>

</head>
<body>
<h1>L33t Str33t</h1>
<ul>
<?php foreach ($links as $link => $url): ?>
<li><?php echo html::anchor($url, $link) ?></li>
<?php endforeach ?>
</ul>
<?php echo $content ?>

</body>
</html>
```

### Create a Menu

Add in the controller code, to produce the menu.

Note: We do this in the website controller, because all our controllers will need a menu. So we add a constructor, and call the parent constructor, because we need to be able to access the template object properties, to add our links.

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Website_Controller extends Template_Controller {

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

    $this->template->links = array
    (
        'Home' => 'home',
        'About' => 'about',
        'Products' => 'products',
        'Contact' => 'contact',
    );
}
```

Edit the Home controller to initialise some content.

```
public function index()
{
    $this->template->title = 'Home::L33t Accessories';

    $this->template->content = '';
}
}
```

## Kohana 101

Load up `http://localhost/kohana/index.php/home` you should see the *L33t Str33t* heading and four links. If you don't, then review your code, check the controllers are declared correctly, that the controller and view files exist, and are correctly named.

Now, what happens if you click on the *About* link? Try it. We have not yet created an *about* controller, so the page does not exist. Kohana displays a nice error page, the message **Page not found** and some useful information in a stack trace.

Try load `http://localhost/kohana` Did you see the default, "Welcome to Kohana" page? That's not what we want, so lets fix it. We must tell Kohana that the default controller is no longer called "Welcome" Copy `system/config/routes.php` to `application/config/routes.php` and edit the file, changing the defined default route, as below:

```
$config['_default'] = 'home';
```

Reload `http://localhost/kohana` and you should see the *L33t Str33t* home page!

## Get Stylish

Now we're getting somewhere, let's add some style.

Stylesheets are a static resource, and you should let the webserver serve them. Create a folder called *media* in the document root, with a sub folder called *css* and save the code below in a file called *site.css* Your layout should look something like this:

```
Document root : htdocs or var/www or home/user/public_html or similar
kohana
  index.php : The application front controller
  application
  media
    css
      site.css
  system
```

### Stylesheet

```
html { background: #DDD }
body { width: 700px; margin: 2em auto;
font-size: 80%; font-family: Arial, Verdana, sans-serif; }
h1, h2 { font-family: Helvetica, serif; }
a { text-decoration: underline; }
ul { list-style: none; padding: 1em 0; }
ul li { display: inline; padding-right: 1em; }
p { margin: 0.5em,0.5em,0,0; padding: 0.5em; }
```

Now edit `application/views/template.php` and add this code to the HTML Document *head* just below the *title*

```
<?php echo html::stylesheet(array
(
    'media/css/site',
),
array
(
    'screen',
));
?>
```

## Kohana 101

This code inserts a link for the stylesheet into the HTML. It will be served from `http://localhost/kohana/media/css/site.css` You may specify any number of stylesheets, and any number of targets; screen, print, etc. Reload your page. You should be looking at the Home webpage, with a horizontal menu of page links, and the worst styling ever!

Time to create some content for the Home page. Create a new view folder "pages" and a new view file named `application/views/pages/home.php` as follows:

```
<h2>Home</h2>
<h3>Get the Latest L33t Gear Today!</h3>
<p>
Are you a <em>Lamer</em>? are you a <em>Babe Banisher</em>?</p>
<p>
Get L33t today!</p>
<p>
We have the gear you need to join the <em>L33t</em> cliques,<br />
From Pocket calculators to Pocket protectors, we have the look!
You will never be <em>lamer</em> againer!</p>
```

Edit the Home controller and change this line in the `index()` method:

```
$this->template->content = new View('pages/home');
```

Reload the page, neat huh!

We are almost done, we just need to add the remaining views, and create controllers for them. Create view files for "about", "products" and "contact" and add some content.

```
file - application/views/pages/about.php

<h2>About</h2>
<h3>About Us</h3>
<p>
L33t Accessories sells the gear that makes you look really Geeky!
</p>
<p>
L33t since 1983 and when Microsoft took a bite out of Apple.</p>

file - application/views/pages/products.php

<h2>Products</h2>
<h3>L33t Products</h3>
<ol>
<li>Pocket Calculator : 55¢ each</li>
<li>Pocket Protector : 45¢ each</li>
</ol>

file - application/views/pages/contact.php

<h2>Contact</h2>
<h3>We don't want to hear from you</h3>
<p>If you were really L33t, you would know how to contact us dude!
</p>
```

## Take Control

Let's do the controllers. For each new controller, copy *application/controllers/home.php* to *application/controllers/new\_name.php* and edit each file as follows:

```
file - application/controllers/about.php

class About_Controller extends Website_Controller
public function index()
{
    $this->template->title = 'About::L33t Accessories';
    $this->template->content = new View('pages/about');
}

file - application/controllers/products.php

class Products_Controller extends Website_Controller
public function index()
{
    $this->template->title = 'Products::L33t Accessories';
    $this->template->content = new View('pages/products');
}

file - application/controllers/contact.php

public function index()
{
    $this->template->title = 'Contact::L33t Accessories';
    $this->template->content = new View('pages/contact');
}
```

## Data, data, everywhere,

Nice! We have a functioning website, with all our pages in place, and stubs for the content.

Our next task is to display real information for the products page. For that, we will use a MySQL database. So first thing to do, is create a database, some tables, and then populate the tables with data.

Create a database called *leetdb* with a user called *leet* and password *l33t*. The user should have *SELECT*, *UPDATE*, *INSERT*, *DELETE*, *CREATE*, *ALTER*, *INDEX* permissions for *leetdb*. It is recommended to use *utf-8* for the database character set.

*L33t Str33t* sells various products. We need to display the product information to the user, including prices. We also need to store relevant information, such as; Products on special, discounts, stock availability. Products are grouped by type, such as; Stationery, Clothing etc. Let's create the tables.

```
Table : categories - Stationery, Clothing, Accessories
A Category has many products which belong to it

DROP TABLE IF EXISTS `categories`;

CREATE TABLE `categories` (
  `id` int(9) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL,
  `description` varchar(120) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## Kohana 101

Table : products - pens, shirts, etc  
A Product belongs to only one Category.  
cat\_id is the foreign key relation to category id

```
DROP TABLE IF EXISTS `products`;  
  
CREATE TABLE `products` (  
  `id` int(9) unsigned NOT NULL AUTO_INCREMENT,  
  `cat_id` int(9) unsigned NOT NULL,  
  `code` varchar(20) NOT NULL DEFAULT '',  
  `description` varchar(120) NOT NULL,  
  `unit` int(6) NOT NULL,  
  `price` int(9) NOT NULL,  
  `special` tinyint(1) NOT NULL DEFAULT '0',  
  `discount` tinyint(3) unsigned NOT NULL DEFAULT '0',  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `code` (`code`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

### Populate the tables with some data

```
Categories  
insert into `categories`(`id`,`name`,`description`)  
values  
(1,'Stationery','Leet Stationery'),  
(2,'Clothing','Leet Clothing'),  
(3,'Accessorie','Leet Gadgets');  
  
Products  
insert into `products`  
(`id`,`cat_id`,`code`,`description`,`unit`,`price`,`special`,`discount`)  
values  
(1,1,'PEN001','Mean Green BallPoints',1,100,0,0),  
(2,2,'CAP001','Red Peakless Baseball Caps',1,1000,0,0),  
(3,3,'CAL001','Luminous Reverse Polish Calculator',1,10000,0,0),  
(4,1,'PEN002','Hot Red BallPoints',1,111,0,0),  
(5,2,'CAP002','Green Peaked Soccer Helmets',1,1111,0,0),  
(6,3,'CAL002','Black Inverse Mongolian Calculator',1,11111,0,0);
```

We have a database and tables setup, now let's configure the connection. Copy *system/config/database.php* to *application/config/database.php* and edit the file as follows:

```

$config['default'] = array
(
    'benchmark'      => TRUE,
    'persistent'     => FALSE,
    'connection'     => array
    (
        'type'       => 'mysql',
        'user'       => 'leet',           // set to db user name
        'pass'      => 'l33t',         // set to db user password
        'host'      => 'localhost',
        'port'      => FALSE,
        'socket'    => FALSE,
        'database' => 'leetdb'         // set to db name
    ),
    'character_set' => 'utf8',
    'table_prefix' => '',
    'object'       => TRUE,
    'cache'        => FALSE,
    'escape'       => TRUE
);

```

## Start your Engines

Let's test the database connection.

Edit *application/controllers/website.php* we will load the database here.

```

$this->template->links = array
(
    'Home'      => 'home',
    'About'     => 'about',
    'Products' => 'products',
    'Contact'   => 'contact',
);

$this->db = Database::instance(); // add this line
// makes database object available to all controllers

```

Edit *application/controllers/products.php* we will test our database connection here. Note the use of static method *Kohana::debug* which is very useful for debugging. It can print out the contents of any variable or returned data, in a readable format.

```

$this->template->title = 'L33t Str33t::Products';
$this->template->content = new View('pages/products');

echo Kohana::debug($this->db->list_fields('categories')); // add this line

```

Reload <http://localhost/kohana/index.php/products> in your browser. You should see (*array*) Array and a list of the data fields in table *categories*. If you encounter an error, check the stack trace for clues. Make sure, all the names are correct, database exists and MySQL is running, user permissions are ok.

If all is working, remove the *Kohana::debug* line from the products controller.

## Master Data

Using a Database in Kohana, is easier than falling off a bicycle. You have already seen how to configure a connection to a database, and how to make the connection. Now let's explain it all in more detail.

*Database* is the class that handles all the interaction with your database. It makes a connection when needed, and kills the connection when no longer in use. Methods are provided to query the database, and handle the returned results. *Database* actually calls another class, a "driver" which implements the vendor specific methods. Kohana has official drivers for *MySQL* and *PostgreSQL*. There are unofficial drivers for *SQLite* and *MSSQL*.

Kohana provides five ways to work with relational databases:

1. Write your own SQL. Use the *query()* method. You **must** escape the query manually.
2. Write custom SQL, that is automatically escaped. Use Query Binding.
3. Use the Query Builder. Provides methods that generate platform independant SQL statements that are automatically escaped.
4. Use a Model. Utilise Query or Query Builder to create your own data access methods.
5. Use Object Relational Mapping. *ORM* is a class that extends the basic Model, to implement data access without using SQL statements.

Kohana provides two ways to work with data returned from a query:

1. Result Object. The default is to return result data as an object. Methods, eg *query->result()* are provided to work with the result object.
2. Result Array. Results may be returned as an array, and data accessed with array notation.

Here are some examples. We want to fetch some information from the *categories* table defined earlier.

```
Write our own SQL

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

// Display all rows and access results as an array
$query->result(FALSE)
foreach ($query as $row)
{
    echo $row['name'];
}

// Display only the first row of the result set, and access results as an object
echo $query->current(); // short cut, initial position of pointer is first result
```

The Query Builder provides a comfortable balance between performance and development speed. It enables you to build applications which are vendor independant. If you need to do custom SQL statements, then it is easier to isolate the vendor specific code.

A very attractive feature of Query Builder is the ability to use method chaining:

```
Query Builder. Select data from table `products` for item id 3

$query = $this->db->select() // selects all fields by default
    ->where('id', 3)
    ->from('products')
    ->get();
```

## Claudia Who?

Now, everything is in place, let's employ some Models to work with the data.

Copy the code below, and save it in a new file called *application/models/product.php* Note the use of the singular, "product" for the Model, and plural "products" for the table. This is a Kohana convention. Note the Class names are first letter uppercase.

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
class Product_Model extends Model {

    public function __construct()
    {
        parent::__construct() // assigns database object to $this->db
    }
}
```

Models are used to abstract data access within an application. The Model does all the data retrieval and manipulation on request from a controller. The Model returns data results to a requesting controller. Neither the controller nor the View access data directly.

This is what we want to happen. When the user clicks the "products" link, the user should see all the available products, in table format, with a row for each item. We also would like the products to be displayed in a sort order: All items primarily sorted by category, and within category, sorted by product code. Edit *application/models/product.php* as follows:

```
class Product_Model extends Model {

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

    public function browse()
    {
        return $this->db->select
            (
                'categories.description AS cat_description',
                'products.code',
                'products.description AS prod_description',
                'products.price',
                'products.unit'
            )
            ->from('categories')
            ->join('products', 'categories.id = products.cat_id')
            ->orderby
                (array('categories.description' => 'ASC', 'products.code' => 'ASC'))
            ->get();
    }
}
```

The *browse* method will return a database result object. It will consist of every row from "products" that has a corresponding *category\_id* in "categories"

Edit Controller *application/controllers/products.php* to request this data from the Product\_Model, and assign the data to a variable in the products View.

```

class Products_Controller extends Website_Controller {

    public function index()
    {
        $this->template->title = 'L33t Str33t::Products';
        $this->template->content = new View('pages/products');

        $products = new Product_Model; // instantiate the model
        // result set is assigned to a variable called $products in the view
        $this->template->content->products = $products->browse();
    }
}

```

## Product Show

Edit View *application/views/pages/products.php* to display the product information in tabular format:

```

<h2>Products</h2>
<h3>L33t Products</h3>
<table>
<tr>
<th>Category</th>
<th>Product Code</th>
<th>Product Description</th>
<th>Price</th>
<th>Units</th>
</tr>
<?php foreach ($products as $item): ?>
    <tr>
        <td><?php echo html::specialchars($item->cat_description) ?></td>
        <td><?php echo html::specialchars($item->code) ?></td>
        <td><?php echo html::specialchars($item->prod_description) ?></td>
        <td>$ <?php echo number_format(($item->price / 100), 2) ?></td>
        <td>per</td>
        <td><?php echo html::specialchars($item->unit) ?></td>
    </tr>
<?php endforeach ?>
</table>

```

Load up <http://localhost/kohana> in your browser. Click on the "products" link. You should see a table with six product items.

Note the code `$(item->price / 100)`. I am storing all financial amounts in cents, without decimals, in Integer format. To display any amount in conventional, Dollars and Cents format, simply divide the amount by 100, and never worry about rounding errors again.

## Anything Special?

When the user loads our home page, we want them to see a notice for an item which is available at a discount price for the week.

Let's add a method to our products Model, to fetch the item on special. Edit *applications/models/product.php* and add this method:

```
public function special()
{
    return $this->db->select()
        ->where('special', 1)
        ->from('products')
        ->get();
}
```

Now, edit the home view *application/views/pages/home.php* and add in the notice block for the special:

```
<div>
<hr />
<h4>On Special this Week Only!</h4>
<p>Our amazing
<?php echo html::specialchars($product_special->current()->description) ?>
normally
<?php echo number_format(($product_special->current()->price / 100), 2) ?>
get one this week for <strong> <?php echo $product_special->current()->discount ?>
percent off! </strong></p></div>
```

Lastly, edit *application/controllers/home.php* to request the data from the model, and pass it to the view.

```
public function index()
{
    $this->template->title = 'L33t Str33t::Home';
    $this->template->content = new View('pages/home');
    // add the code below
    $product = new Product_Model;
    // this assigns the result set to a variable $product_special in the view
    $this->template->content->product_special = $product->special();
}
```

Click on the "Home" link. Did you get a big horrible stack trace?

Can you think why? Yes, off course, we don't have any items on special, and we have not allowed for that in the view. We can fix this in two ways. One is to enforce a rule that there must always be one item on special. Two is to cater for the possibility that there are no items on special. I'm going with Two.

Let's fix this by using a "View-in-View" We will conditionally include a fragment of HTML in the main View. Edit *application/views/pages/home.php* Cut the HTML we added for the special, and save it to a new view file *application/views/pages/home\_special.php* Your files should look like this:

## Kohana 101

```
new view file : application/views/pages/home_content.php

<hr />
<h4>On Special this Week Only!</h4>
<p>Our amazing
<?php echo html::anchor
('products', html::specialchars($product_special->current()->description)) ?>
  Normally $ <em>
  <?php echo number_format(($product_special->current()->price / 100), 2) ?></em>
  Get it this week only for <strong>
  <?php echo $product_special->current()->discount ?> percent
  off! </strong></p>

-----
edited view file : application/views/pages/home.php
home_content.php view will be rendered from variable $special

<h2>Home</h2>
<h3>Get the Latest L33t Gear Today!</h3>
<p>
Are you a <em>Lamer</em>? are you a <em>Babe Banisher</em>?</p>
<p>
Get L33t today!</p>
<p>
We have the gear you need to join the <em>L33t</em> cliques,<br />
From Pocket calculators to Pocket protectors, we have the look!
You will never be <em>lamer</em> againer!</p>

<div>
<?php echo $special ?>
</div>
```

Now, edit *application/controllers/home.php* and instantiate the new view.

```
public function index()
{
    $this->template->title = 'L33t Str33t::Home';
    $this->template->content = new View('pages/home');

    $product = new Product_Model; // instantiate the model
    $query = $product->special(); // assign the result set
    $this->template->content->special = ''; // initialize view fragment to null
    if ($query->count()) // do we have items on special?
    {
        // Yes, so assign the view to the view variable $special
        $this->template->content->special = new View('pages/home_special');
        // assign the result set to the view-in-view variable, $product_special
        $this->template->content->special->product_special = $query;
    }
}
}
```

Reload <http://localhost/kohana> The page should appear as normal, with no errors.

Update the last product item in table *products* to set the special flag on, with a discount of 20.

```
UPDATE `products`
SET `special`=1, `discount` = 20
WHERE id = 6;
```

Reload the Home page. The item-on-special block should be visible.

## Alien Contact

We want visitors to be able to send us a message from the contact page.

So, we need to put up a form, check that we are not accepting any rubbish, and then send the message to our email address

Edit the contact view `application/views/contact.php` We will pass two arrays to the view, `$form` contains the submitted form values, `$errors` contains any validation error messages.

```
<h2>Contact</h2>
<h3>Send us a message</h3>
<p>Tell us something we don't know already!</p>
<?php foreach ($errors as $error): ?>
  <div><?php echo $error ?></div>
<?php endforeach ?>
<form action="<?php echo url::site().'contact' ?>" method="post">
  <div><label for="name">Name</label>
  <input type="text" name="name" id="name"
  value="<?php echo html::specialchars($form['name']) ?>" />
  </div>
  <div><label for="email">Email</label>
  <input type="text" name="email" id="email"
  value="<?php echo html::specialchars($form['email']) ?>" />
  </div>
  <div><label for="email">Message</label></div>
  <div>
  <textarea name="message" cols="30"
  rows="4"><?php echo html::specialchars($form['message']) ?></textarea>
  </div>
  <div><input type="submit" value="Send" /></div>
</form>
```

We use the *Validation* class to validate user input. All we do is instantiate the class, passing the `$_POST` global to the validation object. We attach validation rules to the object, and run the object's `validate()` method. If errors are detected, we pass them to the view. The form input values are also passed to the view, to display again.

The form input will be re-validated each time the user submits the form, until everything is correct, or the user goes home.

The initial form is displayed. We test if the form was submitted, and only then perform validation.

A *callback* method is defined to perform a custom validation check. The method is *public* but prefixed with an underscore, to prevent it being called from a URL.

Edit the contact controller `application/controllers/contact.php` The index method will now also process the form. Edit it as follows:

```

public function index()
{
    $this->template->title = 'L33t Str33t::contact';
    $this->template->content = new View('pages/contact');

    $form = array
    (
        'name'    => '',
        'email'   => '',
        'message' => '',
    );
    $this->template->content->errors = array();
    if ( ! $_POST)
    {
        $this->template->content->form = $form;
    }
    else
    {
        $post = new Validation($_POST);
        $post->pre_filter('trim', TRUE);
        $post->add_rules('name','required', 'length[3,20]', 'alpha');
        $post->add_rules('email', 'required', 'valid::email');
        $post->add_rules('message', 'length[0,500]');
        $post->add_callbacks('message', array($this, '_msg_check'));
        $post->post_filter('ucfirst', 'name');

        $form = $post->as_array();

        if ( ! $post->validate())
        {
            // repopulate form fields and show errors
            $this->template->content->form = $form;
            $this->template->content->errors = $post->errors('form_error');
        }
        else
        {
            // send the email
            url::redirect();
        }
    }
}

public function _msg_check(Validation $post)
{
    if (array_key_exists('message', $post->errors()))
        return;

    if (strlen($post->message) < 5)
        $post->add_error( 'message', 'msg_check');
}

```

To display meaningful error messages, we need to define them somewhere. These messages are stored in a *language* file, as PHP array entries, similar to config entries.

Create a new folder called *application/i18n/en\_US* and a new file *application/i18n/en\_US/form\_error.php* containing the following code:

```
<?php defined('SYSPATH') or die('No direct script access.');
```

```
$lang = array
(
  'name' => Array
    (
      'required' => 'Your Name is required.',
      'alpha' => 'Name must have only alphabetic characters.',
      'length' => 'Name must be between three, and twenty letters.',
      'default' => 'Invalid Input.',
    ),
  'email' => Array
    (
      'required' => 'Email address is required.',
      'email' => 'Email address format is incorrect.',
      'default' => 'Email address is invalid.',
    ),
  'message' => Array
    (
      'length' => 'Message is too long',
      'msg_check' => 'Message is too short.',
      'default' => 'Message text is invalid.',
    ),
);
```

Load the contact page in the browser, <http://localhost/kohana/index.php/contact> and experiment with submitting the form, if all the input is valid, after submission, you are re-directed to the home page.

It is a good practice to redirect after successful form submission, usually, you will redirect to a "thank you" page or display a flash message.

All we need to do now, is email the message. Edit `application/controllers/contact.php` and look for the comment, "`// send the email`" Replace it with this code:

```
// send the email,
$to = 'nobody@localhost'; // change this for your requirements
$subject = $post->name.' : Message to Leet Street';
// email::send(recipient, from, subject, message)
$from = array($post->name, $post->email);
email::send($to, $from, $subject, $post->message);
url::redirect();
```

Emails are sent using *SwiftMailer*. I am using the email helper, but you can use the vendor class directly. Note: The example uses *smtp* to send a secure email via *Gmail* server. You need an account to do this. Please configure email for your own settings.

Copy the email config file from `system/config/email.php` to `application/config/email.php` and edit as follows:

```
$config['driver'] = 'smtp';
$config['options'] = array(
  'hostname' => 'smtp.gmail.com',
  'username' => 'account.holder@gmail.com',
  'password' => 'password',
  'port' => '465',
  'auth' => '',
  'encryption' => 'tls');
```

Try it out!

## Paging all passengers.

The products page is fine as long as there are only a few items in the database.

Add in some more items, and we see the page becomes unusably long. Time for pagination!

```

Insert these items into table : products

INSERT INTO `products`
(`id`,`cat_id`,`code`,`description`,`unit`,`price`,`special`,`discount`)
VALUES (7,1,'PEN003','Ultra Black BallPoints',1,999,0,0),
      (8,1,'PEN004','Super Red BallPoints',1,999,0,0),
      (9,1,'PEN005','Lime Green BallPoints',1,999,0,0),
      (10,1,'PEN006','Navy Blue BallPoints',1,999,0,0),
      (11,1,'PEN007','Ultra Red BallPoints',1,999,0,0),
      (12,1,'PEN008','Super Green BallPoints',1,999,0,0),
      (13,1,'PEN009','Forest Green BallPoints',1,999,0,0),
      (14,1,'PEN010','Sky Blue BallPoints',1,999,0,0)

```

Copy *system/config/pagination.php* to *application/config/pagination.php* We edit this file to configure how our pagination will work. What we want is this: When you visit <http://localhost/products> there should be a way to see the items a page at a time. We will add a *page* method to the controller, and the argument to that method will be the page number.

```

pagination config

$config['default'] = array
(
    // pagination links look like http://localhost/kohana/index.php/products/page/1
    'directory'      => 'pagination',
    'style'          => 'classic',
    'uri_segment'    => 3, // segment 3 is the page number
    'query_string'   => '',
    'items_per_page' => 5,
    'auto_hide'      => FALSE,
);

```

Pagination is configured. Now we must edit the controller, *application/controllers/products.php* to fetch and display the pages. We get the page number to display, from argument one, (URI segment three.)

We must get the total number of product items, from our model, (see model, further down) and the pagination class will calculate everything else for us. Note, we use a property of the class, *sql\_offset* to tell the model, from which row, (offset) to return data.

## Kohana 101

```
products controller

public function index()
{
    // No page number supplied, so default to page one
    url::redirect('products/page/1');
}

public function page($pagenum)
{
    $this->template->title = 'L33t Str33t::Products';
    $this->template->content = new View('pages/products');

    $products = new Product_Model;
    // Instantiate Pagination, passing it the total number of product rows.
    $paging = new Pagination(array
        (
            'total_items' => $products->count_products(),
        )
    );
    // render the page links
    $this->template->content->pagination = $paging->render();
    // Display X items per page, from offset = page number
    $this->template->content->products =
        $products->browse($paging->items_per_page, $paging->sql_offset);
}
```

Two things left to do. First, edit *application/models/product.php* to add a method to count the total number of rows, and amend the *browse* method to accept arguments, and limit the result set. Last, edit *application/views/pages/products.php* to display the pagination links.

```
Product model

public function count_products()
{ // new method
    $query = $this->db->select('id')->get('products');
    return $query->count();
}

public function browse($limit, $offset)
{ // edited method
    return $this->db->select
        (
            'categories.description AS cat_description',
            'products.code',
            'products.description AS prod_description',
            'products.price',
            'products.unit'
        )
        ->from('categories')
        ->join('products', 'categories.id = products.cat_id')
        ->orderby
            (array('categories.description' => 'ASC', 'products.code' => 'ASC'))
        ->limit($limit, $offset)
        ->get();
}
```

```
Products View - add the last line
```

```
<?php endforeach ?>
</table>
<?php echo $pagination ?>
```

Pagination is done! Try it out.

All working? Note, the pagination class will cast page number arguments as *integer* and tries to do the sane thing. Load <http://localhost/kohana/index.php/products/page/99> The last page will display. Try different numbers, even negative numbers, or garbage, and see what happens.

Now, you may have noticed a problem. Load <http://localhost/kohana/index.php/products/page> You should see a stack trace about *missing argument 1*. We will fix this problem, by defining a *route* that sends any products page URL, without a page number, to page one. Edit *application/config/routes.php* like the code below:

```
$config['_default'] = 'home';
// Route anything without a page number to page 1
$config['products/page'] = 'products/page/1';
```

The pagination class is flexible and configurable. You can define pagination config groups, which you instantiate at run time: `$p = new Pagination('config_group_name')` This makes it very easy to change your pagination on the fly, using different pagination styles, number of items per page, or even switching from using segment based arguments, to query string arguments.

### #7: What Next?

You should change the site domain configuration, from *localhost/kohana/* to *localhost/leetstreet/* and copy, or move the *htdocs/kohana* folder in your web server document root to *htdocs/leetstreet*.

Hopefully, you now have a basic, but fully functioning website built with Kohana!

To start cranking out sophisticated sites with Kohana, you will need more information than this guide can provide, here are some resources:

1. Read the user documentation at <http://doc.kohanaphp.com>
2. Search the forum, a gold mine at <http://forum.kohanaphp.com>
3. Never forget. **The source is with you.** Read it and learn.