Kohana 3.2 Documentation

Official documentation from Kohana into one page. Compiled by Xavi Esteve, last updated Friday, 24 February 2012.

What is Kohana?

Kohana is an open source, object oriented MVC web framework built using PHP5 by a team of volunteers that aims to be swift, secure, and small.

Kohana is licensed under a BSD license, so you can legally use it for any kind of open source, commercial, or personal project.

What makes Kohana great?

Anything can be extended using the unique filesystem design, little or no configuration is necessary, error handling helps locate the source of errors quickly, and debugging and profiling provide insight into the application.

To help secure your applications, tools for input validation, signed cookies, form and HTML generators are all included. The database layer provides protection against SQL injection. Of course, all official code is carefully written and reviewed for security.

Contribute to the Documentation

We are working very hard to provide complete documentation. To help improve the guide, please fork the userguide, make your changes, and send a pull request. If you are not familiar with git, you can also submit a feature request (requires registration).

Unofficial Documentation

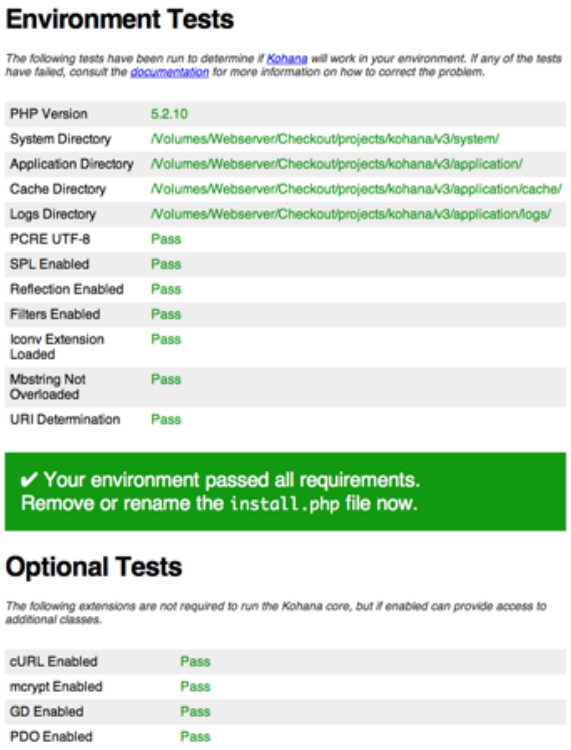
If you are having trouble finding an answer here, have a look through the unofficial wiki. Your answer may also be found by searching the forum or stackoverflow followed by asking your question on either. Additionally, you can chat with the community of developers on the freenode #kohana IRC channel.

Installation

1. Download the latest **stable** release from the Kohana website.
2. Unzip the downloaded package to create a `kohana` directory.
3. Upload the contents of this folder to your webserver.
4. Open `application/bootstrap.php` and make the following changes:
    - Set the default timezone for your application.
    - Set the `base_url` in the Kohana::init call to reflect the location of the kohana folder on your server relative to the document root.
5. Make sure the `application/cache` and `application/logs` directories are writable by the web server.
6. Test your installation by opening the URL you set as the `base_url` in your favorite browser.

Depending on your platform, the installation's subdirs may have lost their permissions thanks to zip extraction. Chmod them all to 755 by running `find . -type d -exec chmod 0755 {} \;` from the root of your Kohana installation.

You should see the installation page. If it reports any errors, you will need to correct them before continuing.



Environment Tests

The following tests have been run to determine if Kohana will work in your environment. If any of the tests have failed, consult the documentation for more information on how to correct the problem.

PHP Version	5.2.10
System Directory	/Volumes/Webserver/Checkout/projects/kohana/v3/system/
Application Directory	/Volumes/Webserver/Checkout/projects/kohana/v3/application/
Cache Directory	/Volumes/Webserver/Checkout/projects/kohana/v3/application/cache/
Logs Directory	/Volumes/Webserver/Checkout/projects/kohana/v3/application/logs/
PCRE UTF-8	Pass
SPL Enabled	Pass
Reflection Enabled	Pass
Filters Enabled	Pass
Iconv Extension Loaded	Pass
Mbstring Not Overloaded	Pass
URI Determination	Pass

✔ Your environment passed all requirements.
Remove or rename the `install.php` file now.

Optional Tests

The following extensions are not required to run the Kohana core, but if enabled can provide access to additional classes.

cURL Enabled	Pass
mcrypt Enabled	Pass
GD Enabled	Pass
PDO Enabled	Pass

Once your install page reports that your environment is set up correctly you need to either rename or delete `install.php` in the root directory. Kohana is now installed and you should see the output of the welcome controller:

hello, world!

Installing Kohana 3.1 From GitHub

The source code for Kohana 3.1 is hosted with GitHub. To install Kohana using the github source code first you need to install git. Visit http://help.github.com for details on how to install git on your platform.

For more information on installing Kohana using git submodules, see the Working with Git tutorial.

Conventions and Coding Style

It is encouraged that you follow Kohana's coding style. This makes code more readable and allows for easier code sharing and contributing.

Class Names and File Location

Class names in Kohana follow a strict convention to facilitate autoloading. Class names should have uppercase first letters with underscores to separate words. Underscores are significant as they directly reflect the file location in the filesystem.

The following conventions apply:

1. CamelCased class names should not be used, except when it is undesirable to create a new directory level.
2. All class file names and directory names are lowercase.
3. All classes should be in the `classes` directory. This may be at any level in the cascading filesystem.

Examples

Remember that in a class, an underscore means a new directory. Consider the following examples:

Class Name	File Path
Controller_Template	classes/controller/template.php
Model_User	classes/model/user.php
Database	classes/database.php
Database_Query	classes/database/query.php
Form	classes/form.php

Coding Standards

In order to produce highly consistent source code, we ask that everyone follow the coding standards as closely as possible.

Brackets

Please use BSD/Allman Style bracketing.

Curly Brackets

Curly brackets are placed on their own line, indented to the same level as the control statement.

```
// Correct
if ($a === $b)
{
    ...
}
else
{
    ...
}

// Incorrect
if ($a === $b) {
    ...
} else {
    ...
}
```

Class Brackets

The only exception to the curly bracket rule is, the opening bracket of a class goes on the same line.

```
// Correct
class Foo {

// Incorrect
class Foo
{
```

Empty Brackets

Don't put any characters inside empty brackets.

```
// Correct
class Foo {}

// Incorrect
class Foo { }
```

Array Brackets

Arrays may be single line or multi-line.

```
array('a' => 'b', 'c' => 'd')

array(
    'a' => 'b',
    'c' => 'd',
)
```

Opening Parenthesis

The opening array parenthesis goes on the same line.

```
// Correct
array(
    ...
)

// Incorrect:
array
(
    ...
)
```

Closing parenthesis

Single Dimension

The closing parenthesis of a multi-line single dimension array is placed on its own line, indented to the same level as the assignment or statement.

```
// Correct
$array = array(
    ...
)

// Incorrect
$array = array(
    ...
    )
```

Multidimensional

The nested array is indented one tab to the right, following the single dimension rules.

```
// Correct
array(
    'arr' => array(
        ...
    ),
    'arr' => array(
        ...
    ),
)

array(
    'arr' => array(...),
    'arr' => array(...),
)
```

Arrays as Function Arguments

```
// Correct
do(array(
    ...
))

// Incorrect
do(array(
    ...
    ))
```

As noted at the start of the array bracket section, single line syntax is also valid.

```
// Correct
do(array(...))
```

```
// Alternative for wrapping long lines
do($bar, 'this is a very long line',
    array(...));
```

Naming Conventions

Kohana uses under_score naming, not camelCase naming.

Classes

```
// Controller class, uses Controller_ prefix
class Controller_Apple extends Controller {

// Model class, uses Model_ prefix
class Model_Cheese extends Model {

// Regular class
class Peanut {
```

When creating an instance of a class, don't use parentheses if you're not passing something on to the constructor:

```
// Correct:
$db = new Database;

// Incorrect:
$db = new Database();
```

Functions and Methods

Functions should be all lowercase, and use under_scores to separate words:

```
function drink_beverage($beverage)
{
```

Variables

All variables should be lowercase and use under_score, not camelCase:

```
// Correct:
$foo = 'bar';
$long_example = 'uses underscores';

// Incorrect:
$weDontWantThis = 'understood?';
```

Indentation

You must use tabs to indent your code. Using spaces for tabbing is strictly forbidden.

Vertical spacing (for multi-line) is done with spaces. Tabs are not good for vertical alignment because different people have different tab widths.

```
$text = 'this is a long text block that is wrapped. Normally, we aim for '
      .'wrapping at 80 chars. Vertical alignment is very important for '
      .'code readability. Remember that all indentation is done with tabs,'
      .'but vertical alignment should be completed with spaces, after '
      .'indenting with tabs.';
```

String Concatenation

Do not put spaces around the concatenation operator:

```
// Correct:
$str = 'one'.$var.'two';
```

```
// Incorrect:
$str = 'one'. $var .'two';
$str = 'one' . $var . 'two';
```

Single Line Statements

Single-line IF statements should only be used when breaking normal execution (e.g. return or continue):

```
// Acceptable:
if ($foo == $bar)
    return $foo;

if ($foo == $bar)
    continue;

if ($foo == $bar)
    break;

if ($foo == $bar)
    throw new Exception('You screwed up!');

// Not acceptable:
if ($baz == $bun)
    $baz = $bar + 2;
```

Comparison Operations

Please use OR and AND for comparison:

```
// Correct:
if (($foo AND $bar) OR ($b AND $c))

// Incorrect:
if (($foo && $bar) || ($b && $c))
```

Please use elseif, not else if:

```
// Correct:
elseif ($bar)

// Incorrect:
else if($bar)
```

Switch Structures

Each case, break and default should be on a separate line. The block inside a case or default must be indented by 1 tab.

```
switch ($var)
{
    case 'bar':
    case 'foo':
        echo 'hello';
    break;
    case 1:
        echo 'one';
    break;
    default:
        echo 'bye';
    break;
}
```

Parentheses

There should be one space after statement name, followed by a parenthesis. The ! (bang) character must have a space on either side to ensure maximum readability. Except in the case of a bang or type casting, there should be no whitespace after an opening parenthesis or before a closing parenthesis.

```
// Correct:
if ($foo == $bar)
if ( ! $foo)

// Incorrect:
if($foo == $bar)
if(!$foo)
if ((int) $foo)
if ( $foo == $bar )
if (! $foo)
```

Ternaries

All ternary operations should follow a standard format. Use parentheses around expressions only, not around just variables.

```
$foo = ($bar == $foo) ? $foo : $bar;
$foo = $bar ? $foo : $bar;
```

All comparisons and operations must be done inside of a parentheses group:

```
$foo = ($bar > 5) ? ($bar + $foo) : strlen($bar);
```

When separating complex ternaries (ternaries where the first part goes beyond ~80 chars) into multiple lines, spaces should be used to line up operators, which should be at the front of the successive lines:

```
$foo = ($bar == $foo)
     ? $foo
     : $bar;
```

Type Casting

Type casting should be done with spaces on each side of the cast:

```
// Correct:
$foo = (string) $bar;
if ( (string) $bar)

// Incorrect:
$foo = (string)$bar;
```

When possible, please use type casting instead of ternary operations:

```
// Correct:
$foo = (bool) $bar;

// Incorrect:
$foo = ($bar == TRUE) ? TRUE : FALSE;
```

When casting type to integer or boolean, use the short format:

```
// Correct:
$foo = (int) $bar;
$foo = (bool) $bar;

// Incorrect:
$foo = (integer) $bar;
$foo = (boolean) $bar;
```

Constants

Always use uppercase for constants:

```
// Correct:
define('MY_CONSTANT', 'my_value');
$a = TRUE;
$b = NULL;

// Incorrect:
define('MyConstant', 'my_value');
$a = True;
$b = null;
```

Place constant comparisons at the end of tests:

```
// Correct:
if ($foo !== FALSE)

// Incorrect:
if (FALSE !== $foo)
```

This is a slightly controversial choice, so I will explain the reasoning. If we were to write the previous example in plain English, the correct example would read:

```
if variable $foo is not exactly FALSE
```

And the incorrect example would read:

```
if FALSE is not exactly variable $foo
```

Since we are reading left to right, it simply doesn't make sense to put the constant first.

Comments

One-line Comments

Use //, preferably above the line of code you're commenting on. Leave a space after it and start with a capital. Never use #.

```
// Correct

//Incorrect
// incorrect
# Incorrect
```

Regular Expressions

When coding regular expressions please use PCRE rather than the POSIX flavor. PCRE is considered more powerful and faster.

```
// Correct:
if (preg_match('/abc/i', $str))

// Incorrect:
if (eregi('abc', $str))
```

Use single quotes around your regular expressions rather than double quotes. Single-quoted strings are more convenient because of their simplicity. Unlike double-quoted strings they don't support variable interpolation nor integrated backslash sequences like \n or \t, etc.

```
// Correct:
preg_match('/abc/', $str);

// Incorrect:
preg_match("/abc/", $str);
```

When performing a regular expression search and replace, please use the $n notation for backreferences. This is preferred over \n.

```
// Correct:
```

```
preg_replace('/(\d+) dollar/', '$1 euro', $str);

// Incorrect:
preg_replace('/(\d+) dollar/', '\\1 euro', $str);
```

Finally, please note that the $ character for matching the position at the end of the line allows for a following newline character. Use the D modifier to fix this if needed. More info.

```
$str = "email@example.com\n";

preg_match('/^.+@.+$/', $str);  // TRUE
preg_match('/^.+@.+$/D', $str); // FALSE
```

http://kohanaframework.org/guide/about.mvc

Discuss the MVC pattern, as it pertains to Kohana. Perhaps have an image, etc.

Models

From Wikipedia:

> The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

Creating a simple model:

```
class Model_Post extends Model
{
    public function do_stuff()
    {
        // This is where you do domain logic...
    }
}
```

If you want database access, have your model extend the Model_Database class:

```
class Model_Post extends Model_Database
{
    public function do_stuff()
    {
        // This is where you do domain logic...
    }

    public function get_stuff()
    {
        // Get stuff from the database:
        return $this->db->query(...);
    }
}
```

If you want CRUD/ORM capabilities, see the ORM Module

Views

Views are files that contain the display information for your application. This is most commonly HTML, CSS and Javascript but can be anything you require such as XML or JSON for AJAX output. The purpose of views is to keep this information separate from your application logic for easy reusability and cleaner code.

Views themselves can contain code used for displaying the data you pass into them. For example, looping through an array of product information and display each one on a new table row. Views are still PHP files so you can use any code you normally would. However, you should try to keep your views as "dumb" as possible and retreive all data you need in your controllers, then pass it to the view.

Creating View Files

View files are stored in the `views` directory of the filesystem. You can also create sub-directories within the `views` directory to organize your files. All of the following examples are reasonable view files:

```
APPPATH/views/home.php
APPPATH/views/pages/about.php
APPPATH/views/products/details.php
MODPATH/error/views/errors/404.php
MODPATH/common/views/template.php
```

Loading Views

View objects will typically be created inside a Controller using the View::factory method. Typically the view is then assigned as the Request::$response property or to another view.

```
public function action_about()
{
    $this->response->body(View::factory('pages/about'));
}
```

When a view is assigned as the Response::body, as in the example above, it will automatically be rendered when necessary. To get the rendered result of a view you can call the View::render method or just type cast it to a string. When a view is rendered, the view file is loaded and HTML is generated.

```
public function action_index()
{
    $view = View::factory('pages/about');

    // Render the view
    $about_page = $view->render();

    // Or just type cast it to a string
    $about_page = (string) $view;

    $this->response->body($about_page);
}
```

Variables in Views

Once view has been loaded, variables can be assigned to it using the View::set and View::bind methods.

```
public function action_roadtrip()
{
    $view = View::factory('user/roadtrip')
        ->set('places', array('Rome', 'Paris', 'London', 'New York', 'Tokyo'));
        ->bind('user', $this->user);

    // The view will have $places and $user variables
    $this->response->body($view);
}
```

The only difference between `set()` and `bind()` is that `bind()` assigns the variable by reference. If you `bind()` a variable before it has been defined, the variable will be created with a value of `NULL`.

You can also assign variables directly to the View object. This is identical to calling `set()`;

```
public function action_roadtrip()
{
    $view = View::factory('user/roadtrip');

    $view->places = array('Rome', 'Paris', 'London', 'New York', 'Tokyo');
    $view->user = $this->user;
```

```
    // The view will have $places and $user variables
    $this->response->body($view);
}
```

Global Variables

An application may have several view files that need access to the same variables. For example, to display a page title in both the header of your template and in the body of the page content. You can create variables that are accessible in any view using the View::set_global and View::bind_global methods.

```
// Assign $page_title to all views
View::bind_global('page_title', $page_title);
```

If the application has three views that are rendered for the home page: `template`, `template/sidebar`, and `pages/home`. First, an abstract controller to create the template will be created:

```
abstract class Controller_Website extends Controller_Template {

    public $page_title;

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

        // Make $page_title available to all views
        View::bind_global('page_title', $this->page_title);

        // Load $sidebar into the template as a view
        $this->template->sidebar = View::factory('template/sidebar');
    }

}
```

Next, the home controller will extend `Controller_Website`:

```
class Controller_Home extends Controller_Website {

    public function action_index()
    {
        $this->page_title = 'Home';

        $this->template->content = View::factory('pages/home');
    }

}
```

Views Within Views

If you want to include another view within a view, there are two choices. By calling View::factory you can sandbox the included view. This means that you will have to provide all of the variables to the view using View::set or View::bind:

```
// In your view file:

// Only the $user variable will be available in "views/user/login.php"
<?php echo View::factory('user/login')->bind('user', $user) ?>
```

The other option is to include the view directly, which makes all of the current variables available to the included view:

```
// In your view file:

// Any variable defined in this view will be included in "views/message.php"
<?php include Kohana::find_file('views', 'user/login') ?>
```

You can also assign a variable of your parent view to be the child view from within your controller. For example:

```
// In your controller:
```

```
public functin action_index()
{
    $view = View::factory('common/template);

    $view->title = "Some title";
    $view->body = View::factory('pages/foobar');
}

// In views/common/template.php:

<html>
<head>
    <title><?php echo $title></title>
</head>

<body>
    <?php echo $body ?>
</body>
</html>
```

Of course, you can also load an entire Request within a view:

```
<?php echo Request::factory('user/login')->execute() ?>
```

This is an example of [HMVC], which makes it possible to create and read calls to other URLs within your application.

Cascading Filesystem

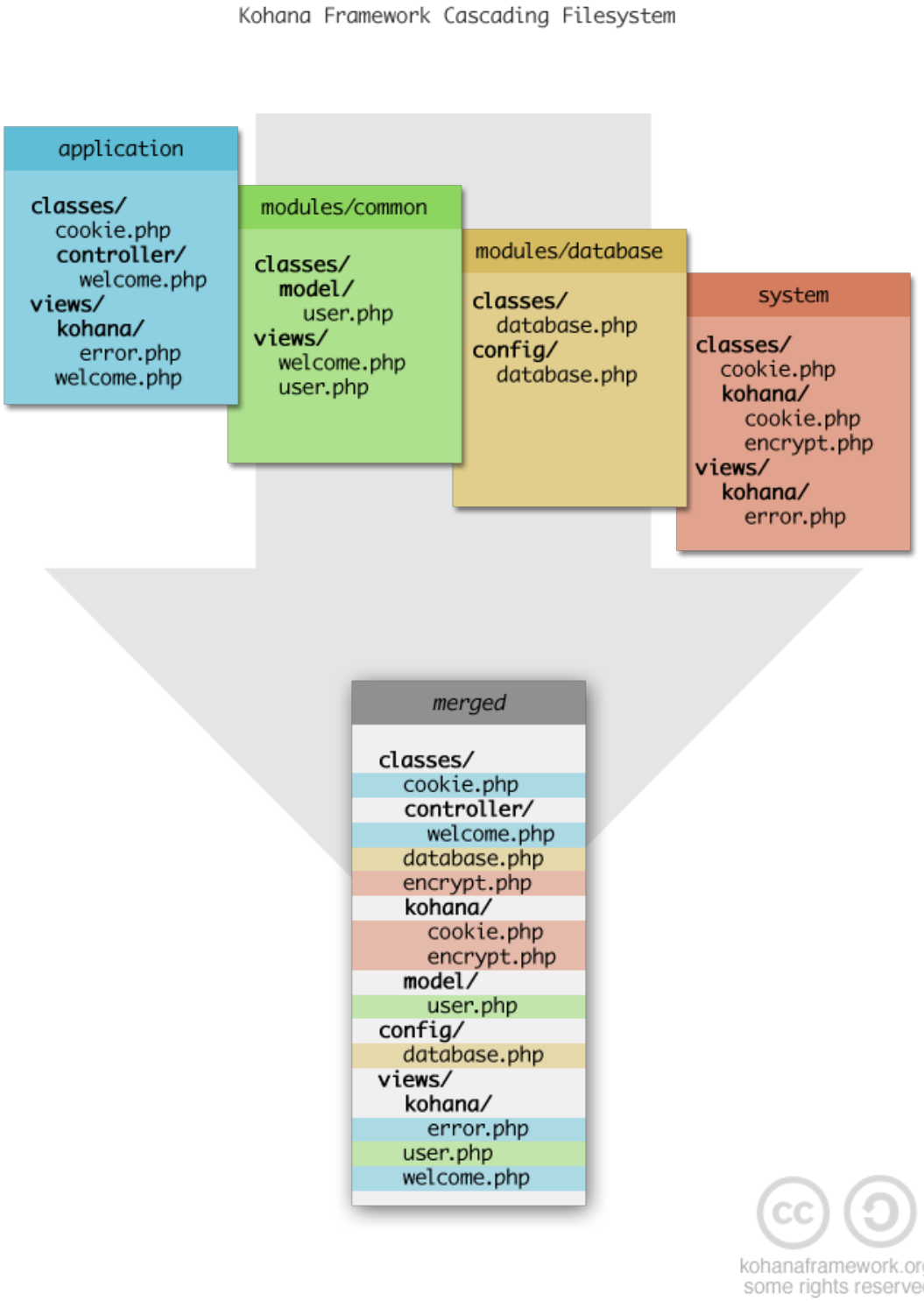
The Kohana filesystem is a hierarchy of similar directory structures that cascade. The hierarchy in Kohana (used when a file is loaded by Kohana::find_file) is in the following order:

1. **Application Path**
   Defined as `APPPATH` in `index.php`. The default value is `application`.

2. **Module Paths**
   This is set as an associative array using Kohana::modules in `APPPATH/bootstrap.php`. Each of the values of the array will be searched **in the order that the modules are defined**.

3. **System Path**
   Defined as `SYSPATH` in `index.php`. The default value is `system`. All of the main or "core" files and classes are defined here.

Files that are in directories higher up the include path order take precedence over files of the same name lower down the order, which makes it is possible to overload any file by placing a file with the same name in a "higher" directory:



This image is only shows certain files, but we can use it to illustrate some examples of the cascading filesystem:

- If Kohana catches an error, it would display the `kohana/error.php` view, So it would call `Kohana::find_file('views', 'kohana/error')`. This would return `application/views/kohana/error.php` because it takes precidence over `system/views/kohana/error.php`. By doing this we can change the error view without editing the system folder.
- If we used `View::factory('welcome')` it would call `Kohana::find_file('views','welcome')` which would return `application/views/welcome.php` because it takes precidence over `modules/common/views/welcome.php`. By doing this, you can overwrite things in a module without editing the modules files.
- If use the Cookie class, Kohana::auto_load will call `Kohana::find_file('classes', 'cookie')` which will return `application/classes/cookie.php`. Assuming Cookie extends Kohana_Cookie, the autoloader would then call `Kohana::find_file('classes','kohana/cookie')` which will return

`system/classes/kohana/cookie.php` because that file does not exist anywhere higher in the cascade. This is an example of transparent extension.

- If you used `View::factory('user')` it would call `Kohana::find_file('views','user')` which would return `modules/common/views/user.php`.
- If we wanted to change something in `config/database.php` we could copy the file to `application/config/database.php` and make the changes there. Keep in mind that config files are merged rather than overwritten by the cascade.

Types of Files

The top level directories of the application, module, and system paths have the following default directories:

classes/
: All classes that you want to autoload should be stored here. This includes controllers, models, and all other classes. All classes must follow the class naming conventions.

config/
: Configuration files return an associative array of options that can be loaded using Kohana::$config. Config files are merged rather than overwritten by the cascade. See config files for more information.

i18n/
: Translation files return an associative array of strings. Translation is done using the `__()` method. To translate "Hello, world!" into Spanish, you would call `__('Hello, world!')` with I18n::$lang set to "es-es". I18n files are merged rather than overwritten by the cascade. See I18n files for more information.

messages/
: Message files return an associative array of strings that can be loaded using Kohana::message. Messages and i18n files differ in that messages are not translated, but always written in the default language and referred to by a single key. Message files are merged rather than overwritten by the cascade. See message files for more information.

views/
: Views are plain PHP files which are used to generate HTML or other output. The view file is loaded into a View object and assigned variables, which it then converts into an HTML fragment. Multiple views can be used within each other. See views for more information.

other
: You can include any other folders in your cascading filesystem. Examples include, but are not limited to, `guide`, `vendor`, `media`, whatever you want. For example, to find `media/logo.png` in the cascading filesystem you would call `Kohana::find_file('media','logo','png')`.

Finding Files

The path to any file within the filesystem can be found by calling Kohana::find_file:

```
// Find the full path to "classes/cookie.php"
$path = Kohana::find_file('classes', 'cookie');

// Find the full path to "views/user/login.php"
$path = Kohana::find_file('views', 'user/login');
```

If the file doesn't have a `.php` extension, pass the extension as the third param.

```
// Find the full path to "guide/menu.md"
$path = Kohana::find_file('guide', 'menu', 'md');

// If $name is "2000-01-01-first-post" this would look for "posts/2000-01-01-first-post.textile
$path = Kohana::find_file('posts', $name, '.textile');
```

Vendor Extensions

We call extensions or external libraries that are not specific to Kohana "vendor" extensions, and they go in the vendor folder, either in application or in a module. Because these libraries do not follow Kohana's file naming conventions, they cannot be autoloaded by Kohana, so you will have to manually included them. Some examples of vendor libraries are Markdown, DOMPDF, Mustache and Swiftmailer.

For example, if you wanted to use DOMPDF, you would copy it to `application/vendor/dompdf` and include the DOMPDF autoloading class. It can be useful to do this in a controller's before method, as part of a module's init.php, or

the contstructor of a singleton class.

```
require Kohana::find_file('vendor', 'dompdf/dompdf/dompdf_config','inc');
```

Now you can use DOMPDF without loading any more files:

```
$pdf = new DOMPDF;
```

If you want to convert views into PDFs using DOMPDF, try the PDFView module.

Config Files

Configuration files are used to store any kind of configuration needed for a module, class, or anything else you want. They are plain PHP files, stored in the `config/` directory, which return an associative array:

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

return array(
    'setting' => 'value',
    'options' => array(
        'foo' => 'bar',
    ),
);
```

If the above configuration file was called `myconf.php`, you could access it using:

```
$config = Kohana::$config->load('myconf');
$options = $config->get('options')
```

Merge

Configuration files are slightly different from most other files within the cascading filesystem in that they are **merged** rather than overloaded. This means that all configuration files with the same file path are combined to produce the final configuration. The end result is that you can overload *individual* settings rather than duplicating an entire file.

For example, if we wanted to change or add to an entry in the inflector configuration file, we would not need to duplicate all the other entries from the default configuration file.

```
// config/inflector.php

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

return array(
    'irregular' => array(
        'die' => 'dice', // does not exist in default config file
        'mouse' => 'mouses', // overrides 'mouse' => 'mice' in the default config file
);
```

Creating your own config files

Let's say we want a config file to store and easily change things like the title of a website, or the google analytics code. We would create a config file, let's call it `site.php`:

```
// config/site.php

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

return array(
    'title' => 'Our Shiny Website',
    'analytics' => FALSE, // analytics code goes here, set to FALSE to disable
);
```

We could now call `Kohana::$config->load('site.title')` to get the site name, and `Kohana::$config->load('site.analytics')` to get the analytics code.

Let's say we want an archive of versions of some software. We could use config files to store each version, and include links to download, documentation, and issue tracking.

```
// config/versions.php

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

return array(
    '1.0.0' => array(
        'codename' => 'Frog',
        'download' => 'files/ourapp-1.0.0.tar.gz',
        'documentation' => 'docs/1.0.0',
        'released' => '06/05/2009',
        'issues' => 'link/to/bug/tracker',
    ),
    '1.1.0' => array(
        'codename' => 'Lizard',
        'download' => 'files/ourapp-1.1.0.tar.gz',
        'documentation' => 'docs/1.1.0',
        'released' => '10/15/2009',
        'issues' => 'link/to/bug/tracker',
    ),
    /// ... etc ...
);
```

You could then do the following:

```
// In your controller
$view->versions = Kohana::$config->load('versions');

// In your view:
foreach ($versions as $version)
{
    // echo some html to display each version
}
```

I18n

Kohana has a fairly simple and easy to use i18n system. It is slightly modeled after gettext, but is not as featureful. If you need the features of gettext, please use that :)

__()

Kohana has a __() function to do your translations for you. This function is only meant for small sections of text, not entire paragraphs or pages of translated text.

To echo a translated string:

```
<?php echo __('Hello, world!');?>
```

This will echo 'Home' unless you've changed the defined language, which is explained below.

Changing the displayed language

Use the I18n::lang() method to change the displayed language:

```
I18n::lang('fr');
```

This will change the language to 'es-es'.

Defining language files

To define the language file for the above language change, create a `i18n/fr.php` that contains:

```php
<?php

return array
(
    'Hello, world!' => 'Bonjour, monde!',
);
```

Now when you do `__('Hello, world!')`, you will get `Bonjour, monde!`

I18n variables

You can define variables in your __() calls like so:

```php
echo __('Hello, :user', array(':user' => $username));
```

Your i18n key in your translation file will need to be defined as:

```php
<?php

return array
(
    'Hello, :user' => 'Bonjour, :user',
);
```

Defining your own __() function

You can define your own __() function by simply defining your own i18n class:

```php
<?php

class I18n extends Kohana_I18n
{
    // Intentionally empty
}

function __($string, array $values = NULL, $lang = 'en-us')
{
    // Your functionality here
}
```

This will cause the built-in __() function to be ignored.

Messages

Kohana has a robust key based lookup system so you can define system messages.

Getting a message

Use the Kohana::message() method to get a message key:

```php
Kohana::message('forms', 'foobar');
```

This will look in the `messages/forms.php` file for the `foobar` key:

```php
<?php

return array(
    'foobar' => 'Hello, world!',
);
```

You can also look in subfolders and sub-keys:

```php
Kohana::message('forms/contact', 'foobar.bar');
```

This will look in the `messages/forms/contact.php` for the `[foobar][bar]` key:

```
<?php

return array(
    'foobar' => array(
        'bar' => 'Hello, world!',
    ),
);
```

Notes

- Don't use __() in your messages files, as these files can be cached and will not work properly.
- Messages are merged by the cascading file system, not overwritten like classes and views.

Configuration

By default kohana is setup to load configuration values from config files in the cascading filesystem. However, it is very easy to adapt it to load config values in other locations/formats.

Sources

The system is designed around the concept of **Config Sources**, which loosely means a method of storing configuration values.

To read config from a source you need a **Config Reader**. Similarly, to write config to a source you need a **Config Writer**.

Implementing them is as simpe as extending the Kohana_Config_Reader / Kohana_Config_Writer interfaces:

```
class Kohana_Config_Database_Reader implements Kohana_Config_Reader
class Kohana_Config_Database_Writer extends Kohana_Config_Database_Reader implements Kohana_Con
```

You'll notice in the above example that the Database Writer extends the Database Reader. This is the convention with config sources, the reasoning being that if you can write to a source chances are you can also read from it as well. However, this convention is not enforced and is left to the developer's discretion.

Groups

In order to aide organisation config values are split up into logical "groups". For example database related settings go in a `database` group, and session related settings go in a `session` group.

How these groups are stored/organised is up to the config source. For example the file source puts different config groups into different files (`database.php`, `session.php`) whereas the database source uses a column to distinguish between groups.

To load a config group simply call `Kohana::$config->load()` with the name of the group you wish to load:

```
$config = Kohana::$config->load('my_group');
```

`load()` will return an instance of Config_Group which encapsulates the config values and ensures that any modifications made will be passed back to the config writers.

To get a config value from a Config_Group object simply call Config_Group::get:

```
$config = Kohana::$config->load('my_group');
$value  = $config->get('var');
```

To modify a value call Config_Group::set:

```
$config = Kohana::$config->load('my_group');
$config->set('var', 'new_value');
```

Alternative methods for getting / setting config

In addition to the methods described above you can also access config values using dots to outline a path from the config group to the value you want:

```
// Config file: database.php
return array(
    'default' => array(
        'connection' => array(
            'hostname' => 'localhost'
        )
    )
);

// Code which needs hostname:
$hostname = Kohana::$config->load('database.default.connection.hostname');
```

Which is equivalent to:

```
$config = Kohana::$config->load('database')->get('default');

$hostname = $config['connection']['hostname'];
```

Obviously this method is a lot more compact than the original, however please bear in mind that using `dot.notation` is a *lot* slower than calling `get()` and traversing the array yourself. Dot notation can be useful if you only need one specific variable, but otherwise it's best to use `get()`.

As Config_Group extends Array_Object you can also use array syntax to get/set config vars:

```
$config = Kohana::$config->load('database');

// Getting the var
$hostname = $config['default']['connection']['hostname'];

// Setting the var
$config['default']['connection']['hostname'] = '127.0.0.1';
```

Again, this syntax is more costly than calling `get()` / `set()`.

Config Merging

One of the useful features of the config system is config group merging. This works in a similar way to the cascading filesystem, with configuration from lower sources lower down the source stack being merged with sources further up the stack.

If two sources contain the same config variables then the one from the source further up the stack will override the one from the "lower" source. However, if the source from higher up the stack does not contain a particular config variable but a source lower down the stack does then the value from the lower source will be used.

The position of sources in the stack is determined by how they are loaded in your bootstrap. By default when you load a source it is pushed to the top of a stack:

```
// Stack: <empty>
Kohana::$config->attach(new Config_File);
// Stack: Config_File
Kohana::$config->attach(new Config_Database);
// Stack: Config_Database, Config_File
```

In the example above, any config values found in the database will override those found in the filesystem. For example, using the setup outlined above:

```
// Configuration in the filesystem:
    email:
        sender:
            email: my.awesome.address@example.com
            name:  Unknown
        method: smtp
```

```
// Configuration in the database:
    email:
        sender:
            email: my.supercool.address@gmail.com
            name:  Kohana Bot

// Configuration returned by Kohana::$config->load('email')
    email:
        sender:
            email: my.supercool.address@gmail.com
            name:  Kohana Bot
        method: smtp
```

Note: The above syntax is simply pseudo code to illustrate the concept of config merging.

On some occasions you may want to append a config source to the bottom of the stack, to do this pass `FALSE` as the second parameter to `attach()`:

```
// Stack: <empty>
Kohana::$config->attach(new Config_File);
// Stack: Config_File
Kohana::$config->attach(new Config_Database, FALSE);
// Stack: Config_File, Config_Database
```

In this example, any values found in the filesystem will override those found in the db. For example:

```
// Configuration in the filesystem:
    email:
        sender:
            email: my.awesome.address@example.com
            name:  Unknown
        method: smtp

// Configuration in the database:
    email:
        sender:
            email: my.supercool.address@gmail.com
            name:  Kohana Bot

// Configuration returned by Kohana::$config->load('email')
    email:
        sender:
            email: my.awesome.address@example.com
            name:  Unknown
        method: smtp
```

Using different config sources based on the environment

In some situation's you'll need to use different config values depending on which state `Kohana::$environment` is in. Unit testing is a prime example of such a situation. Most setups have two databases; one for normal development and a separate one for unit testing (to isolate the tests from your development).

In this case you still need access to the config settings stored in the `config` directory as it contains generic settings that are needed whatever environment your application is in (e.g. encryption settings),)o replacing the default `Config_File` source isn't really an option.

To get around this you can attach a separate config file reader which loads its config from a subdir of `config` called "testing":

```
Kohana::$config->attach(new Config_File);

Kohana::$config->attach(new Config_Database);

if (Kohana::$environment === Kohana::TESTING)
{
```

```
    Kohana::$config->attach(new Config_File('config/testing'));
}
```

During normal development the config source stack looks like `Config_Database, Config_File('config')`. However, when `Kohana::$environment === Kohana::TESTING` the stack looks like `Config_File('config/testing'), Config_Database, Config_File('config')`

Request Flow

Every application follows the same flow:

1. Application starts from `index.php`.
    1. The application, module, and system paths are set. (`APPPATH`, `MODPATH`, and `SYSPATH`)
    2. Error reporting levels are set.
    3. Install file is loaded, if it exists.
    4. The bootstrap file, `APPPATH/bootstrap.php`, is included.
2. Once we are in `bootstrap.php`:
    1. The Kohana class is loaded.
    2. Kohana::init is called, which sets up error handling, caching, and logging.
    3. Kohana_Config readers and Kohana_Log writers are attached.
    4. Kohana::modules is called to enable additional modules.
        - Module paths are added to the cascading filesystem.
        - Includes each module's `init.php` file, if it exists.
        - The `init.php` file can perform additional environment setup, including adding routes.
    5. Route::set is called multiple times to define the application routes.
    6. Request::instance is called to start processing the request.
        1. Checks each route that has been set until a match is found.
        2. Creates the controller instance and passes the request to it.
        3. Calls the Controller::before method.
        4. Calls the controller action, which generates the request response.
        5. Calls the Controller::after method.
            - The above 5 steps can be repeated multiple times when using HMVC sub-requests.
3. Application flow returns to index.php
    1. The main Request response is displayed

Bootstrap

The bootstrap is located at `application/bootstrap.php`. It is responsible for setting up the Kohana environment and executing the main response. It is included by `index.php` (see Request flow)

The bootstrap is responsible for the flow of your application. In previous versions of Kohana the bootstrap was in `system` and was somewhat of an unseen, uneditible force. In Kohana 3 the bootstrap takes on a much more integral and versatile role. Do not be afraid to edit and change your bootstrap however you see fit.

Environment setup

First the bootstrap sets the timezone and the locale, and adds Kohana's autoloader so the cascading filesystem works. You could add any other settings that all your application needed here.

```
// Sample excerpt from bootstrap.php with comments trimmed down

// Set the default time zone.
date_default_timezone_set('America/Chicago');

// Set the default locale.
setlocale(LC_ALL, 'en_US.utf-8');

// Enable the Kohana auto-loader.
spl_autoload_register(array('Kohana', 'auto_load'));

// Enable the Kohana auto-loader for unserialization.
ini_set('unserialize_callback_func', 'spl_autoload_call');
```

Initialization and Configuration

Kohana is then initialized by calling Kohana::init, and the log and config reader/writers are enabled.

```
// Sample excerpt from bootstrap.php with comments trimmed down

Kohana::init(array('
    base_url' => '/kohana/',
    index_file => false,
));

// Attach the file writer to logging. Multiple writers are supported.
Kohana::$log->attach(new Kohana_Log_File(APPPATH.'logs'));

// Attach a file reader to config. Multiple readers are supported.
Kohana::$config->attach(new Kohana_Config_File);
```

You can add conditional statements to make the bootstrap have different values based on certain settings. For example, detect whether we are live by checking `$_SERVER['HTTP_HOST']` and set caching, profiling, etc. accordingly. This is just an example, there are many different ways to accomplish the same thing.

```
// Excerpt from http://github.com/isaiahdw/kohanaphp.com/blob/f2afe8e28b/application/bootstrap.
... [trimmed]

/**
 * Set the environment status by the domain.
 */
if (strpos($_SERVER['HTTP_HOST'], 'kohanaphp.com') !== FALSE)
{
    // We are live!
    Kohana::$environment = Kohana::PRODUCTION;

    // Turn off notices and strict errors
    error_reporting(E_ALL ^ E_NOTICE ^ E_STRICT);
}

/**
 * Initialize Kohana, setting the default options.
 ... [trimmed]
 */
Kohana::init(array(
    'base_url'   => Kohana::$environment === Kohana::PRODUCTION ? '/' : '/kohanaphp.com/',
    'caching'    => Kohana::$environment === Kohana::PRODUCTION,
    'profile'    => Kohana::$environment !== Kohana::PRODUCTION,
    'index_file' => FALSE,
));

... [trimmed]
```

Note: The default bootstrap will set `Kohana::$environment = $_ENV['KOHANA_ENV']` if set. Docs on how to supply this variable are available in your web server's documentation (e.g. Apache, Lighttpd). This is considered better practice than many alternative methods to set `Kohana::$enviroment`, as you can change the setting per server, without having to rely on config options or hostnames.

Modules

Read the Modules page for a more detailed description.

Modules are then loaded using Kohana::modules(). Including modules is optional.

Each key in the array should be the name of the module, and the value is the path to the module, either relative or absolute.

```
// Example excerpt from bootstrap.php
```

```
Kohana::modules(array(
    'database'   => MODPATH.'database',
    'orm'        => MODPATH.'orm',
    'userguide'  => MODPATH.'userguide',
));
```

Routes

Read the Routing page for a more detailed description and more examples.

Routes are then defined via Route::set().

```
// The default route that comes with Kohana 3
Route::set('default', '(<controller>(/<action>(/<id>)))')
    ->defaults(array(
        'controller' => 'welcome',
        'action'     => 'index',
    ));
```

Modules

Modules are simply an addition to the Cascading Filesystem. A module can add any kind of file (controllers, views, classes, config files, etc.) to the filesystem available to Kohana (via Kohana::find_file). This is useful to make any part of your application more transportable or shareable between different apps. For example, creating a new modeling system, a search engine, a css/js manager, etc.

Where to find modules

Kolanos has created kohana-universe, a fairly comprehensive list of modules that are available on Github. To get your module listed there, send him a message via Github.

Mon Geslani created a very nice site that allows you to sort Github modules by activity, watchers, forks, etc. It seems to not be as comprehensive as kohana-universe.

Andrew Hutchings has created kohana-modules which is similar to the above sites.

Enabling modules

Modules are enabled by calling Kohana::modules and passing an array of `'name' => 'path'`. The name isn't important, but the path obviously is. A module's path does not have to be in `MODPATH`, but usually is. You can only call Kohana::modules once.

```
Kohana::modules(array(
    'auth'       => MODPATH.'auth',       // Basic authentication
    'cache'      => MODPATH.'cache',      // Caching with multiple backends
    'codebench'  => MODPATH.'codebench',  // Benchmarking tool
    'database'   => MODPATH.'database',   // Database access
    'image'      => MODPATH.'image',      // Image manipulation
    'orm'        => MODPATH.'orm',        // Object Relationship Mapping
    'oauth'      => MODPATH.'oauth',      // OAuth authentication
    'pagination' => MODPATH.'pagination', // Paging of results
    'unittest'   => MODPATH.'unittest',   // Unit testing
    'userguide'  => MODPATH.'userguide',  // User guide and API documentation
    ));
```

Init.php

When a module is activated, if an `init.php` file exists in that module's directory, it is included. This is the ideal place to have a module include routes or other initialization necessary for the module to function. The Userguide and Codebench modules have init.php files you can look at.

How modules work

A file in an enabled module is virtually the same as having that exact file in the same place in the application folder. The main difference being that it can be overwritten by a file of the same name in a higher location (a module enabled after it, or the application folder) via the Cascading Filesystem. It also provides an easy way to organize and share your code.

Creating your own module

To create a module simply create a folder (usually in `DOCROOT/modules`) and place the files you want to be in the module there, and activate that module in your bootstrap. To share your module, you can upload it to Github. You can look at examples of modules made by Kohana or other users.

Routing

Kohana provides a very powerful routing system. In essence, routes provide an interface between the urls and your controllers and actions. With the correct routes you could make almost any url scheme correspond to almost any arrangement of controllers, and you could change one without impacting the other.

As mentioned in the Request Flow section, a request is handled by the Request class, which will look for a matching Route and load the appropriate controller to handle that request.

It is important to understand that **routes are matched in the order they are added**, and as soon as a URL matches a route, routing is essentially "stopped" and *the remaining routes are never tried*. Because the default route matches almost anything, including an empty url, new routes must be place before it.

Creating routes

If you look in `APPPATH/bootstrap.php` you will see the "default" route as follows:

```
Route::set('default', '(<controller>(/<action>(/<id>)))')
->defaults(array(
    'controller' => 'welcome',
    'action'     => 'index',
));
```

The default route is simply provided as a sample, you can remove it and replace it with your own routes.

So this creates a route with the name `default` that will match urls in the format of `(<controller>(/<action>(/<id>)))`.

Let's take a closer look at each of the parameters of Route::set, which are `name`, `uri`, and an optional array `regex`.

Name

The name of the route must be a **unique** string. If it is not it will overwrite the older route with the same name. The name is used for creating urls by reverse routing, or checking which route was matched.

URI

The uri is a string that represents the format of urls that should be matched. The tokens surrounded with `<>` are *keys* and anything surrounded with `()` are *optional* parts of the uri. In Kohana routes, any character is allowed and treated literally aside from `()<>`. The `/` has no meaning besides being a character that must match in the uri. Usually the `/` is used as a static seperator but as long as the regex makes sense, there are no restrictions to how you can format your routes.

Lets look at the default route again, the uri is `(<controller>(/<action>(/<id>)))`. We have three keys or params: controller, action, and id. In this case, the entire uri is optional, so a blank uri would match and the default controller and action (set by defaults(), covered below) would be assumed resulting in the `Controller_Welcome` class being loaded and the `action_index` method being called to handle the request.

You can use any name you want for your keys, but the following keys have special meaning to the Request object, and will influence which controller and action are called:

- **Directory** - The sub-directory of `classes/controller` to look for the controller ([covered below](#directory))
- **Controller** - The controller that the request should execute.

- **Action** - The action method to call.

Regex

The Kohana route system uses perl compatible regular expressions in its matching process. By default each key (surrounded by `<>`) will match `[^/.,;?\n]++` (or in english: anything that is not a slash, period, comma, semicolon, question mark, or newline). You can define your own patterns for each key by passing an associative array of keys and patterns as an additional third argument to Route::set.

In this example, we have controllers in two directories, `admin` and `affiliate`. Because this route will only match urls that begin with `admin` or `affiliate`, the default route would still work for controllers in `classes/controller`.

```
Route::set('sections', '<directory>(/<controller>(/<action>(/<id>)))',
    array(
        'directory' => '(admin|affiliate)'
    ))
    ->defaults(array(
        'controller' => 'home',
        'action'     => 'index',
    ));
```

You can also use a less restrictive regex to match unlimited parameters, or to ignore overflow in a route. In this example, the url `foobar/baz/and-anything/else_that/is-on-the/url` would be routed to `Controller_Foobar::action_baz()` and the `"stuff"` parameter would be `"and-anything/else_that/is-on-the/url"`. If you wanted to use this for unlimited parameters, you could explode it, or you just ignore the overflow.

```
Route::set('default', '(<controller>(/<action>(/<stuff>)))', array('stuff' => '.*'))
    ->defaults(array(
        'controller' => 'welcome',
        'action' => 'index',
  ));
```

Default values

If a key in a route is optional (or not present in the route), you can provide a default value for that key by passing an associated array of keys and default values to Route::defaults, chained after your Route::set. This can be useful to provide a default controller or action for your site, among other things.

The `controller` and `action` key must always have a value, so they either need to be required in your route (not inside of parentheses) or have a default value provided.

In the default route, all the keys are optional, and the controller and action are given a default. If we called an empty url, the defaults would fill in and `Controller_Welcome::action_index()` would be called. If we called `foobar` then only the default for action would be used, so it would call `Controller_Foobar::action_index()` and finally, if we called `foobar/baz` then neither default would be used and `Controller_Foobar::action_baz()` would be called.

TODO: need an example here

You can also use defaults to set a key that isn't in the route at all.

TODO: example of either using directory or controller where it isn't in the route, but set by defaults

Directory

Lambda/Callback route logic

In 3.1, you can specify advanced routing schemes by using lambda routes. Instead of a URI, you can use an anonymous function or callback syntax to specify a function that will process your routes. Here's a simple example:

If you want to use reverse routing with lambda routes, you must pass the third parameter:

```
Route::set('testing', function($uri)
    {
        if ($uri == 'foo/bar')
            return array(
                'controller' => 'welcome',
```

```
                'action'     => 'foobar',
            );
    },
    'foo/bar'
);
```

As you can see in the below route, the reverse uri parameter might not make sense.

```
Route::set('testing', function($uri)
    {
        if ($uri == '</language regex/>(.+)')
        {
            Cookie::set('language', $match[1]);
            return array(
                'controller' => 'welcome',
                'action'     => 'foobar'
            );
        }
    },
    '<language>/<rest_of_uri>
);
```

If you are using php 5.2, you can still use callbacks for this behavior (this example omits the reverse route):

```
Route::set('testing', array('Class', 'method_to_process_my_uri'));
```

Examples

There are countless other possibilities for routes. Here are some more examples:

```
/*
 * Authentication shortcuts
 */
Route::set('auth', '<action>',
  array(
    'action' => '(login|logout)'
  ))
  ->defaults(array(
    'controller' => 'auth'
  ));

/*
 * Multi-format feeds
 *   452346/comments.rss
 *   5373.json
 */
Route::set('feeds', '<user_id>(/<action>).<format>',
  array(
    'user_id' => '\d+',
    'format' => '(rss|atom|json)',
  ))
  ->defaults(array(
    'controller' => 'feeds',
    'action' => 'status',
  ));

/*
 * Static pages
 */
Route::set('static', '<path>.html',
  array(
    'path' => '[a-zA-Z0-9_/]+',
  ))
  ->defaults(array(
    'controller' => 'static',
```

```
    'action' => 'index',
  ));

/*
 * You don't like slashes?
 *   EditGallery:bahamas
 *   Watch:wakeboarding
 */
Route::set('gallery', '<action>(<controller>):<id>',
  array(
    'controller' => '[A-Z][a-z]++',
    'action'     => '[A-Z][a-z]++',
  ))
  ->defaults(array(
    'controller' => 'Slideshow',
  ));

/*
 * Quick search
 */
Route::set('search', ':<query>', array('query' => '.*'))
  ->defaults(array(
    'controller' => 'search',
    'action' => 'index',
  ));
```

Request parameters

The `directory`, `controller` and `action` can be accessed from the Request as public properties like so:

```
// From within a controller:
$this->request->action();
$this->request->controller();
$this->request->directory();

// Can be used anywhere:
Request::current()->action();
Request::current()->controller();
Request::current()->directory();
```

All other keys specified in a route can be accessed via Request::param():

```
// From within a controller:
$this->request->param('key_name');

// Can be used anywhere:
Request::current()->param('key_name');
```

The Request::param method takes an optional second argument to specify a default return value in case the key is not set by the route. If no arguments are given, all keys are returned as an associative array. In addition, `action`, `controller` and `directory` are not accessible via Request::param().

For example, with the following route:

```
Route::set('ads','ad/<ad>(/<affiliate>)')
->defaults(array(
    'controller' => 'ads',
    'action'     => 'index',
));
```

If a url matches the route, then `Controller_Ads::index()` will be called. You can access the parameters by using the `param()` method of the controller's Request. Remember to define a default value (via the second, optional parameter of Request::param) if you didn't in `->defaults()`.

```
class Controller_Ads extends Controller {
```

```
    public function action_index()
    {
        $ad = $this->request->param('ad');
        $affiliate = $this->request->param('affiliate',NULL);
    }
```

Where should routes be defined?

The established convention is to either place your custom routes in the `MODPATH/<module>/init.php` file of your module if the routes belong to a module, or simply insert them into the `APPPATH/bootstrap.php` file (be sure to put them **above** the default route) if they are specific to the application. Of course, nothing stops you from including them from an external file, or even generating them dynamically.

A deeper look at how routes work

TODO: talk about how routes are compiled

Creating URLs and links using routes

Along with Kohana's powerful routing capabilities are included some methods for generating URLs for your routes' uris. You can always specify your uris as a string using URL::site to create a full URL like so:

```
URL::site('admin/edit/user/'.$user_id);
```

However, Kohana also provides a method to generate the uri from the route's definition. This is extremely useful if your routing could ever change since it would relieve you from having to go back through your code and change everywhere that you specified a uri as a string. Here is an example of dynamic generation that corresponds to the `feeds` route example from above:

```
Route::get('feeds')->uri(array(
  'user_id' => $user_id,
  'action' => 'comments',
  'format' => 'rss'
));
```

Let's say you decided later to make that route definition more verbose by changing it to `feeds/<user_id>(/<action>).<format>`. If you wrote your code with the above uri generation method you wouldn't have to change a single line! When a part of the uri is enclosed in parentheses and specifies a key for which there in no value provided for uri generation and no default value specified in the route, then that part will be removed from the uri. An example of this is the `(/<id>)` part of the default route; this will not be included in the generated uri if an id is not provided.

One method you might use frequently is the shortcut Request::uri which is the same as the above except it assumes the current route, directory, controller and action. If our current route is the default and the uri was `users/list`, we can do the following to generate uris in the format `users/view/$id`:

```
$this->request->uri(array('action' => 'view', 'id' => $user_id));
```

Or if within a view, the preferable method is:

```
Request::instance()->uri(array('action' => 'view', 'id' => $user_id));
```

TODO: examples of using html::anchor in addition to the above examples

Testing routes

TODO: mention bluehawk's devtools module

Tips and Common Mistakes

This is a collection of tips and common mistakes or errors you may encounter.

Never edit the `system` folder!

You should (almost) never edit the system folder. Any change you want to make to files in system and modules can be made via the cascading filesystem and transparent extension and won't break when you try to update your Kohana version.

Don't try and use one route for everything

Kohana 3 routes are very powerful and flexible, don't be afraid to use as many as you need to make your app function the way you want!

Handling lots of routes

Sometimes your application is sufficiently complex that you have many routes and it becomes unmanageable to put them all in bootstrap.php. If this is the case, simply make a `routes.php` file in APPPATH and require that in your bootstrap: `require_once APPPATH.'routes'.EXT;`

Reflection_Exception

If you get a Reflection_Exception when setting up your site, it is almost certainly because your Kohana::init 'base_url' setting is wrong. If your base url is correct something is probably wrong with your routes.

```
ReflectionException [ -1 ]: Class controller_<something> does not exist
// where <something> is part of the url you entered in your browser
```

Solution

Set your Kohana::init 'base_url' to the correct setting. The base url should be the path to your index.php file relative to the webserver document root.

ORM/Session __sleep() bug

There is a bug in php which can corrupt your session after a fatal error. A production server shouldn't have uncaught fatal errors, so this bug should only happen during development, when you do something stupid and cause a fatal error. On the next page load you will get a database connection error, then all subsequent page loads will display the following error:

```
ErrorException [ Notice ]: Undefined index: id
MODPATH/orm/classes/kohana/orm.php [ 1308 ]
```

Solution

To fix this, clear your cookies for that domain to reset your session. This should never happen on a production server, so you won't have to explain to your clients how to clear their cookies. You can see the discussion on this issue for more details.

Migrating from 3.1.x

Config

The configuration system has been rewritten to make it more flexible. The most significant change is that config groups are now merged across all readers, similar to how files cascade within the cascading filesystem. Therefore you can now override a single config value (perhaps a database connection string, or a 'send-from' email address) and store it in a separate config source (environment config file / database / custom) while inheriting the remaining settings from that group from your application's configuration files.

In other respects, the majority of the public API should still operate in the same way, however the one major change is the transition from using `Kohana::config()` to `Kohana::$config->load()`, where `Kohana::$config` is an instance of `Config`.

`Config::load()` works almost identically to `Kohana::config()`, e.g.:

```
Kohana::$config->load('dot.notation')
Kohana::$config->load('dot')->notation
```

A simple find/replace for `Kohana::config`/`Kohana::$config->load` within your project should fix this.

The terminology for config sources has also changed. Pre 3.2 config was loaded from "Config Readers" which both read and wrote config. In 3.2 there are **Config Readers** and **Config Writers**, both of which are a type of **Config Source**.

A **Config Reader** is implemented by implementing the `Kohana_Config_Reader` interface; similarly a **Config Writer** is implemented by implementing the `Kohana_Config_Writer` interface.

e.g. for Database:

```
class Kohana_Config_Database_Reader implements Kohana_Config_Reader
class Kohana_Config_Database_Writer extends Kohana_Config_Database_Reader implements Kohana_Con
```

Although not enforced, the convention is that writers extend config readers.

To help maintain backwards compatability when loading config sources empty classes are provided for the db/file sources which extends the source's reader/writer.

e.g.

```
class Kohana_Config_File extends Kohana_Config_File_Reader
class Kohana_Config_Database extends Kohana_Config_Database_Writer
```

External requests

In Kohana 3.2, `Request_Client_External` now has three separate drivers to handle external requests;

- `Request_Client_Curl` is the default driver, using the PHP Curl extension
- `Request_Client_HTTP` uses the PECL HTTP extension
- `Request_Client_Stream` uses streams native to PHP and requires no extensions. However this method is slower than the alternatives.

Unless otherwise specified, `Request_Client_Curl` will be used for all external requests. This can be changed for all external requests, or for individual requests.

To set an external driver across all requests, add the following to the `application/bootstrap.php` file;

```
// Set all external requests to use PECL HTTP
Request_Client_External::$client = 'Request_Client_HTTP';
```

Alternatively it is possible to set a specific client to an individual Request.

```
// Set the Stream client to an individual request and
// Execute
$response = Request::factory('http://kohanaframework.org')
    ->client(new Request_Client_Stream)
    ->execute();
```

HTTP cache control

Kohana 3.1 introduced HTTP cache control, providing RFC 2616 fully compliant transparent caching of responses. Kohana 3.2 builds on this moving all caching logic out of `Request_Client` into `HTTP_Cache`.

HTTP Cache requires the Cache module to be enabled in all versions of Kohana!

In Kohana 3.1, HTTP caching was enabled doing the following;

```
// Apply cache to a request
$request = Request::factory('foo/bar', Cache::instance('memcache'));

// In controller, ensure response sets cache control,
// this will cache the response for one hour
$this->response->headers('cache-control',
    'public, max-age=3600');
```

In Kohana 3.2, HTTP caching is enabled slightly differently;

```
// Apply cache to request
$request = Request::factory('foo/bar',
    HTTP_Cache::factory('memcache'));

// In controller, ensure response sets cache control,
// this will cache the response for one hour
$this->response->headers('cache-control',
    'public, max-age=3600');
```

Controller Action Parameters

In 3.1, controller action parameters were deprecated. In 3.2, these behavior has been removed. If you had any code like:

```
public function action_index($id)
{
    // ... code
}
```

You'll need to change it to:

```
public function action_index()
{
    $id = $this->request->param('id');

    // ... code
}
```

Form Class

If you used Form::open(), the default behavior has changed. It used to default to "/" (your home page), but now an empty parameter will default to the current URI.

Debugging

Kohana includes several tools to help you debug your application.

The most basic of these is Debug::vars. This simple method will display any number of variables, similar to var_export or print_r, but using HTML for extra formatting.

```
// Display a dump of the $foo and $bar variables
echo Debug::vars($foo, $bar);
```

Kohana also provides a method to show the source code of a particular file using Debug::source.

```
// Display this line of source code
echo Debug::source(__FILE__, __LINE__);
```

If you want to display information about your application files without exposing the installation directory, you can use Debug::path:

```
// Displays "APPPATH/cache" rather than the real path
echo Debug::path(APPPATH.'cache');
```

If you are having trouble getting something to work correctly, you could check your Kohana logs and your webserver logs, as well as using a debugging tool like Xdebug.

Loading Classes

Kohana takes advantage of PHP autoloading. This removes the need to call include or require before using a class. When you use a class Kohana will find and include the class file for you. For instance, when you want to use the Cookie::set method, you simply call:

```
Cookie::set('mycookie', 'any string value');
```

Or to load an Encrypt instance, just call Encrypt::instance:

```
$encrypt = Encrypt::instance();
```

Classes are loaded via the Kohana::auto_load method, which makes a simple conversion from class name to file name:

1. Classes are placed in the `classes/` directory of the filesystem
2. Any underscore characters in the class name are converted to slashes
3. The filename is lowercase

When calling a class that has not been loaded (eg: `Session_Cookie`), Kohana will search the filesystem using Kohana::find_file for a file named `classes/session/cookie.php`.

If your classes do not follow this convention, they cannot be autoloaded by Kohana. You will have to manually included your files, or add your own autoload function.

Custom Autoloaders

Kohana's default autoloader is enabled in `application/bootstrap.php` using spl_autoload_register:

```
spl_autoload_register(array('Kohana', 'auto_load'));
```

This allows Kohana::auto_load to attempt to find and include any class that does not yet exist when the class is first used.

Example: Zend

You can easily gain access to other libraries if they include an autoloader. For example, here is how to enable Zend's autoloader so you can use Zend libraries in your Kohana application.

Download and install the Zend Framework files

- Download the latest Zend Framework files.
- Create a `vendor` directory at `application/vendor`. This keeps third party software separate from your application classes.
- Move the decompressed Zend folder containing Zend Framework to `application/vendor/Zend`.

Include Zend's Autoloader in your bootstrap

Somewhere in `application/bootstrap.php`, copy the following code:

```
/**
 * Enable Zend Framework autoloading
 */
if ($path = Kohana::find_file('vendor', 'Zend/Loader'))
{
    ini_set('include_path',
    ini_get('include_path').PATH_SEPARATOR.dirname(dirname($path)));

    require_once 'Zend/Loader/Autoloader.php';
    Zend_Loader_Autoloader::getInstance();
}
```

Usage example

You can now autoload any Zend Framework classes from inside your Kohana application.

```
if ($validate($_POST))
{
    $mailer = new Zend_Mail;

    $mailer->setBodyHtml($view)
        ->setFrom(Kohana::$config->load('site')->email_from)
        ->addTo($email)
        ->setSubject($message)
```

```
        ->send();
}
```

Transparent Class Extension

The cascading filesystem allows transparent class extension. For instance, the class Cookie is defined in `SYSPATH/classes/cookie.php` as:

```
class Cookie extends Kohana_Cookie {}
```

The default Kohana classes, and many extensions, use this definition so that almost all classes can be extended. You extend any class transparently, by defining your own class in `APPPATH/classes/cookie.php` to add your own methods.

You should **never** modify any of the files that are distributed with Kohana. Always make modifications to classes using transparent extension to prevent upgrade issues.

For instance, if you wanted to create method that sets encrypted cookies using the Encrypt class, you would create a file at `application/classes/cookie.php` that extends Kohana_Cookie, and adds your functions:

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

class Cookie extends Kohana_Cookie {

    /**
     * @var  mixed  default encryption instance
     */
    public static $encryption = 'default';

    /**
     * Sets an encrypted cookie.
     *
     * @uses  Cookie::set
     * @uses  Encrypt::encode
     */
     public static function encrypt($name, $value, $expiration = NULL)
     {
         $value = Encrypt::instance(Cookie::$encrpytion)->encode((string) $value);

         parent::set($name, $value, $expiration);
     }

     /**
      * Gets an encrypted cookie.
      *
      * @uses  Cookie::get
      * @uses  Encrypt::decode
      */
      public static function decrypt($name, $default = NULL)
      {
          if ($value = parent::get($name, NULL))
          {
              $value = Encrypt::instance(Cookie::$encryption)->decode($value);
          }

          return isset($value) ? $value : $default;
      }

} // End Cookie
```

Now calling `Cookie::encrypt('secret', $data)` will create an encrypted cookie which we can decrypt with `$data = Cookie::decrypt('secret')`.

How it works

To understand how this works, let's look at what happens normally. When you use the Cookie class, Kohana::autoload looks for `classes/cookie.php` in the cascading filesystem. It looks in `application`, then each module, then `system`. The file is found in `system` and is included. Of coures, `system/classes/cookie.php` is just an empty class which extends `Kohana_Cookie`. Again, Kohana::autoload is called this time looking for `classes/kohana/cookie.php` which it finds in `system`.

When you add your transparently extended cookie class at `application/classes/cookie.php` this file essentially "replaces" the file at `system/classes/cookie.php` without actually touching it. This happens because this time when we use the Cookie class Kohana::autoload looks for `classes/cookie.php` and finds the file in `application` and includes that one, instead of the one in system.

Example: changing Cookie settings

If you are using the Cookie class, and want to change a setting, you should do so using transparent extension, rather than editing the file in the system folder. If you edit it directly, and in the future you upgrade your Kohana version by replacing the system folder, your changes will be reverted and your cookies will probably be invalid. Instead, create a cookie.php file either in `application/classes/cookie.php` or a module (`MODPATH/<modulename>/classes/cookie.php`).

```
class Cookie extends Kohana_Cookie {

    // Set a new salt
    public $salt = "some new better random salt phrase";

    // Don't allow javascript access to cookies
    public $httponly = TRUE;

}
```

Example: TODO: an example

Just post the code and brief description of what function it adds, you don't have to do the "How it works" like above.

Example: TODO: something else

Just post the code and brief description of what function it adds, you don't have to do the "How it works" like above.

More examples

TODO: Provide some links to modules on github, etc that have examples of transparent extension in use.

Multiple Levels of Extension

If you are extending a Kohana class in a module, you should maintain transparent extensions. In other words, do not include any variables or function in the "base" class (eg. Cookie). Instead make your own namespaced class, and have the "base" class extend that one. With our Encrypted cookie example we can create `MODPATH/mymod/encrypted/cookie.php`:

```
class Encrypted_Cookie extends Kohana_Cookie {

    // Use the same encrypt() and decrypt() methods as above

}
```

And create `MODPATH/mymod/cookie.php`:

```
class Cookie extends Encrypted_Cookie {}
```

This will still allow users to add their own extension to Cookie while leaving your extensions intact. To do that they would make a cookie class that extends `Encrypted_Cookie` (rather than `Kohana_Cookie`) in their application folder.

Helpers

Kohana comes with many static "helper" functions to make certain tasks easier.

You can make your own helpers by simply making a class and putting it in the `classes` directory, and you can also extend any helper to modify or add new functions using transparent extension.

- **Arr** - Array functions. Get an array key or default to a set value, get an array key by path, etc.
- **CLI** - Parse command line options.
- **Cookie** - Covered in more detail on the Cookies page.
- **Date** - Useful date functions and constants. Time between two dates, convert between am/pm and military, date offset, etc.
- **Encrypt** - Covered in more detail on the Security page.
- **Feed** - Parse and create RSS feeds.
- **File** - Get file type by mime, split and merge a file into small pieces.
- **Form** - Create HTML form elements.
- **Fragment** - Simple file based caching. Covered in more detail on the Fragments page.
- **HTML** - Useful HTML functions. Encode, obfuscate, create script, anchor, and image tags, etc.
- **I18n** - Internationalization helper for creating multilanguage sites.
- **Inflector** - Change a word into plural or singular form, camelize or humanize a phrase, etc.
- **Kohana** - The Kohana class is also a helper. Debug variables (like print_r but better), file loading, etc.
- **Num** - Provides locale aware formating and english ordinals (th, st, nd, etc).
- **Profiler** - Covered in more detail on the Profiling page.
- **Remote** - Remote server access helper using CURL.
- **Request** - Get the current request url, create expire tags, send a file, get the user agent, etc.
- **Route** - Create routes, create an internal link using a route.
- **Security** - Covered in more detail on the Security page.
- **Session** - Covered in more detail on the Sessions page.
- **Text** - Autolink, prevent window words, convert a number to text, etc.
- **URL** - Create a relative or absolute URL, make a URL-safe title, etc.
- **UTF8** - Provides multi-byte aware string functions like strlen, strpos, substr, etc.
- **Upload** - Helper for uploading files from a form.

Requests

Kohana includes a flexible HMVC request system. It supports out of the box support for internal requests and external requests.

HMVC stands for `Hierarchical Model View Controller` and basically means requests can each have MVC triads called from inside each other.

The Request object in Kohana is HTTP/1.1 compliant.

Creating Requests

Creating a request is very easy:

Internal Requests

An internal request is a request calling to the internal application. It utilizes routes to direct the application based on the URI that is passed to it. A basic internal request might look something like:

```
$request = Request::factory('welcome');
```

In this example, the URI is 'welcome'.

The initial request

Since Kohana uses HMVC, you can call many requests inside each other. The first request (usually called from `index.php`) is called the "initial request". You can access this request via:

```
Request::initial();
```

You should only use this method if you are absolutely sure you want the initial request. Otherwise you should use the `Request::current()` method.

Sub-requests

You can call a request at any time in your application by using the `Request::factory()` syntax. All of these requests will be considered sub-requests.

Other than this difference, they are exactly the same. You can detect if the request is a sub-request in your controller with the is_initial() method:

```
$sub_request = ! $this->request->is_initial()
```

External Requests

An external request calls out to a third party website.

You can use this to scrape HTML from a remote site, or make a REST call to a third party API:

```
// This uses GET
$request = Request::factory('http://www.google.com/');

// This uses PUT
$request = Request::factory('http://example.com/put_api')->method(Request::PUT)->body(json_enco

// This uses POST
$request = Request::factory('http://example.com/post_api')->method(Request::POST)->post(array('
```

Executing Requests

To execute a request, use the `execute()` method on it. This will give you a response object.

```
$request = Request::factory('welcome');
$response = $request->execute();
```

Request Cache Control

You can cache requests for fast execution by passing a cache instance in as the second parameter of factory:

```
$request = Request::factory('welcome', Cache::instance());
```

TODO

Sessions

Kohana provides classes that make it easy to work with both cookies and sessions. At a high level both sessions and cookies provide the same functionality. They allow the developer to store temporary or persistent information about a specific client for later retrieval, usually to make something persistent between requests.

Sessions should be used for storing temporary or private data. Very sensitive data should be stored using the Session class with the "database" or "native" adapters. When using the "cookie" adapter, the session should always be

encrypted.

For more information on best practices with session variables see the seven deadly sins of sessions.

Storing, Retrieving, and Deleting Data

Cookie and Session provide a very similar API for storing data. The main difference between them is that sessions are accessed using an object, and cookies are accessed using a static class.

Accessing the session instance is done using the Session::instance method:

```
// Get the session instance
$session = Session::instance();
```

When using sessions, you can also get all of the current session data using the Session::as_array method:

```
// Get all of the session data as an array
$data = $session->as_array();
```

You can also use this to overload the `$_SESSION` global to get and set data in a way more similar to standard PHP:

```
// Overload $_SESSION with the session data
$_SESSION =& $session->as_array();

// Set session data
$_SESSION[$key] = $value;
```

Storing Data

Storing session or cookie data is done using the `set` method:

```
// Set session data
$session->set($key, $value);
// Or
Session::instance()->set($key, $value);

// Store a user id
$session->set('user_id', 10);
```

Retrieving Data

Getting session or cookie data is done using the `get` method:

```
// Get session data
$data = $session->get($key, $default_value);

// Get the user id
$user = $session->get('user_id');
```

Deleting Data

Deleting session or cookie data is done using the `delete` method:

```
// Delete session data
$session->delete($key);


// Delete the user id
$session->delete('user_id');
```

Session Configuration

Always check these settings before making your application live, as many of them will have a direct affect on the security of your application.

Session Adapters

When creating or accessing an instance of the Session class you can decide which session adapter or driver you wish to use. The session adapters that are available to you are:

Native
: Stores session data in the default location for your web server. The storage location is defined by session.save_path in `php.ini` or defined by ini_set.

Database
: Stores session data in a database table using the Session_Database class. Requires the Database module to be enabled.

Cookie
: Stores session data in a cookie using the Cookie class. **Sessions will have a 4KB limit when using this adapter, and should be encrypted.**

The default adapter can be set by changing the value of Session::$default. The default adapter is "native".

To access a Session using the default adapter, simply call Session::instance(). To access a Session using something other than the default, pass the adapter name to `instance()`, for example: `Session::instance('cookie')`

Session Adapter Settings

You can apply configuration settings to each of the session adapters by creating a session config file at `APPPATH/config/session.php`. The following sample configuration file defines all the settings for each adapter:

As with cookies, a "lifetime" setting of "0" means that the session will expire when the browser is closed.

```
return array(
    'native' => array(
        'name' => 'session_name',
        'lifetime' => 43200,
    ),
    'cookie' => array(
        'name' => 'cookie_name',
        'encrypted' => TRUE,
        'lifetime' => 43200,
    ),
    'database' => array(
        'name' => 'cookie_name',
        'encrypted' => TRUE,
        'lifetime' => 43200,
        'group' => 'default',
        'table' => 'table_name',
        'columns' => array(
            'session_id'  => 'session_id',
            'last_active' => 'last_active',
            'contents'    => 'contents'
        ),
        'gc' => 500,
    ),
);
```

Native Adapter

Type	Setting	Description	Default
`string`	name	name of the session	`"session"`
`integer`	lifetime	number of seconds the session should live for	`0`

Cookie Adapter

Type	Setting	Description	Default
`string`	name	name of the cookie used to store the session data	`"session"`
`boolean`	encrypted	encrypt the session data using Encrypt?	`FALSE`

integer	lifetime	number of seconds the session should live for	0

Database Adapter

Type	Setting	Description	Default
string	group	Database::instance group name	"default"
string	table	table name to store sessions in	"sessions"
array	columns	associative array of column aliases	array
integer	gc	1:x chance that garbage collection will be run	500
string	name	name of the cookie used to store the session data	"session"
boolean	encrypted	encrypt the session data using Encrypt?	FALSE
integer	lifetime	number of seconds the session should live for	0

Table Schema

You will need to create the session storage table in the database. This is the default schema:

```
CREATE TABLE  `sessions` (
    `session_id` VARCHAR(24) NOT NULL,
    `last_active` INT UNSIGNED NOT NULL,
    `contents` TEXT NOT NULL,
    PRIMARY KEY (`session_id`),
    INDEX (`last_active`)
) ENGINE = MYISAM;
```

Table Columns

You can change the column names to match an existing database schema when connecting to a legacy session table. The default value is the same as the key value.

session_id
: the name of the "id" column

last_active
: UNIX timestamp of the last time the session was updated

contents
: session data stored as a serialized string, and optionally encrypted

Cookies

Kohana provides classes that make it easy to work with both cookies and sessions. At a high level both sessions and cookies provide the same functionality. They allow the developer to store temporary or persistent information about a specific client for later retrieval, usually to make something persistent between requests.

Cookies should be used for storing non-private data that is persistent for a long period of time. For example storing a user preference or a language setting. Use the Cookie class for getting and setting cookies.

Kohana uses "signed" cookies. Every cookie that is stored is combined with a secure hash to prevent modification of the cookie. If a cookie is modified outside of Kohana the hash will be incorrect and the cookie will be deleted. This hash is generated using Cookie::salt(), which uses the Cookie::$salt property. You must define this setting in your bootstrap.php:

```
Cookie::$salt = 'foobar';
```

Or define an extended cookie class in your application:

```
class Cookie extends Kohana_Cookie
{
    public static $salt = 'foobar';
}
```

You should set the salt to a secure value. The example above is only for demonstrative purposes.

Nothing stops you from using `$_COOKIE` like normal, but you can not mix using the Cookie class and the regular `$_COOKIE` global, because the hash that Kohana uses to sign cookies will not be present, and Kohana will delete the cookie.

Storing, Retrieving, and Deleting Data

Cookie and Session provide a very similar API for storing data. The main difference between them is that sessions are accessed using an object, and cookies are accessed using a static class.

Storing Data

Storing session or cookie data is done using the Cookie::set method:

```
// Set cookie data
Cookie::set($key, $value);

// Store a user id
Cookie::set('user_id', 10);
```

Retrieving Data

Getting session or cookie data is done using the Cookie::get method:

```
// Get cookie data
$data = Cookie::get($key, $default_value);

// Get the user id
$user = Cookie::get('user_id');
```

Deleting Data

Deleting session or cookie data is done using the Cookie::delete method:

```
// Delete cookie data
Cookie::delete($key);

// Delete the user id
Cookie::delete('user_id');
```

Cookie Settings

All of the cookie settings are changed using static properties. You can either change these settings in `bootstrap.php` or by using transparent extension. Always check these settings before making your application live, as many of them will have a direct affect on the security of your application.

The most important setting is Cookie::$salt, which is used for secure signing. This value should be changed and kept secret:

```
Cookie::$salt = 'your secret is safe with me';
```

Changing this value will render all cookies that have been set before invalid.

By default, cookies are stored until the browser is closed. To use a specific lifetime, change the Cookie::$expiration setting:

```
// Set cookies to expire after 1 week
Cookie::$expiration = 604800;

// Alternative to using raw integers, for better clarity
Cookie::$expiration = Date::WEEK;
```

The path that the cookie can be accessed from can be restricted using the Cookie::$path setting.

```
// Allow cookies only when going to /public/*
Cookie::$path = '/public/';
```

The domain that the cookie can be accessed from can also be restricted, using the Cookie::$domain setting.

```
// Allow cookies only on the domain www.example.com
Cookie::$domain = 'www.example.com';
```

If you want to make the cookie accessible on all subdomains, use a dot at the beginning of the domain.

```
// Allow cookies to be accessed on example.com and *.example.com
Cookie::$domain = '.example.com';
```

To only allow the cookie to be accessed over a secure (HTTPS) connection, use the Cookie::$secure setting.

```
// Allow cookies to be accessed only on a secure connection
Cookie::$secure = TRUE;

// Allow cookies to be accessed on any connection
Cookie::$secure = FALSE;
```

To prevent cookies from being accessed using Javascript, you can change the Cookie::$httponly setting.

```
// Make cookies inaccessible to Javascript
Cookie::$httponly = TRUE;
```

Fragments

Fragments are a quick and simple way to cache HTML or other output. Fragments are not useful for caching objects or raw database results, in which case you should use a more robust caching method, which can be achieved with the Cache module. Fragments use Kohana::cache() and will be placed in the cache directory (`application/cache` by default).

You should use Fragment (or any caching solution) when reading the cache is faster than reprocessing the result. Reading and parsing a remote file, parsing a complicated template, calculating something, etc.

Fragments are typically used in view files.

Usage

Fragments are used by calling Fragment::load() in an `if` statement at the beginning of what you want cached, and Fragment::save() at the end. They use output buffering to capture the output between the two function calls.

You can specify the lifetime (in seconds) of the Fragment using the second parameter of Fragment::load(). The default lifetime is 30 seconds. You can use the Date helper to make more readable times.

Fragments will store a different cache for each language (using I18n) if you pass `true` as the third parameter to Fragment::load();

You can force the deletion of a Fragment using Fragment::delete(), or specify a lifetime of 0.

```
// Cache for 5 minutes, and cache each language
if ( ! Fragment::load('foobar', Date::MINUTE * 5, true))
{
    // Anything that is echo'ed here will be saved
    Fragment::save();
}
```

Example: Calculating Pi

In this example we will calculate pi to 1000 places, and cache the result using a fragment. The first time you run this it will probably take a few seconds, but subsequent loads will be much faster, until the fragment lifetime runs out.

```
if ( ! Fragment::load('pi1000', Date::HOUR * 4))
{
    // Change function nesting limit
    ini_set('xdebug.max_nesting_level',1000);

    // Source: http://mgccl.com/2007/01/22/php-calculate-pi-revisited
    function bcfact($n)
    {
      return ($n == 0 || $n== 1) ? 1 : bcmul($n,bcfact($n-1));
    }
```

```
    function bcpi($precision)
    {
        $num = 0;$k = 0;
        bcscale($precision+3);
        $limit = ($precision+3)/14;
        while($k < $limit)
        {
            $num = bcadd($num, bcdiv(bcmul(bcadd('13591409',bcmul('545140134', $k)),bcmul(bcpow
            ++$k;
        }
        return bcdiv(1,(bcmul(12,($num))),$precision);
    }

    echo bcpi(1000);

    Fragment::save();
}

echo View::factory('profiler/stats');

?>
```

Example: Recent Wikipedia edits

In this example we will use the Feed class to retrieve and parse an RSS feed of recent edits to http://en.wikipedia.org, then use Fragment to cache the results.

```
$feed = "http://en.wikipedia.org/w/index.php?title=Special:RecentChanges&feed=rss";
$limit = 50;

// Displayed feeds are cached for 30 seconds (default)
if ( ! Fragment::load('rss:'.$feed)):

    // Parse the feed
    $items = Feed::parse($feed, $limit);

    foreach ($items as $item):

        // Convert $item to object
        $item = (object) $item;

        echo HTML::anchor($item->link,$item->title);

        ?>
        <blockquote>
            <p>author: <?php echo $item->creator ?></p>
            <p>date: <?php echo $item->pubDate ?></p>
        </blockquote>
        <?php

    endforeach;

    Fragment::save();

endif;

echo View::factory('profiler/stats');
```

Example: Nested Fragments

You can nest fragments with different lifetimes to provide more specific control. For example, let's say your page has lots of dynamic content so we want to cache it with a lifetime of five minutes, but one of the pieces takes much longer to generate, and only changes every hour anyways. No reason to generate it every 5 minutes, so we will use a nested fragment.

If a nested fragment has a shorter lifetime than the parent, it will only get processed when the parent has expired.

```
// Cache homepage for five minutes
if ( ! Fragment::load('homepage', Date::MINUTE * 5)):

    echo "<p>Home page stuff</p>";

    // Pretend like we are actually doing something :)
    sleep(2);

    // Cache this every hour since it doesn't change as often
    if ( ! Fragment::load('homepage-subfragment', Date::HOUR)):

        echo "<p>Home page special thingy</p>";

        // Pretend like this takes a long time
        sleep(5);

    Fragment::save(); endif;

    echo "<p>More home page stuff</p>";

    Fragment::save();

endif;

echo View::factory('profiler/stats');
```

Profiling

Kohana provides a very simple way to display statistics about your application:

1. Common Kohana method calls, such as Kohana::find_file().
2. Requests. Including the main request, as well as any sub-requests.
3. Database queries
4. Average execution times for your application

In order for profiling to work, the `profile` setting must be `TRUE` in your Kohana::init() call in your bootstrap.

Profiling your code

You can easily add profiling to your own functions and code. This is done using the Profiler::start() function. The first parameter is the group, the second parameter is the name of the benchmark.

```
public function foobar($input)
{
    // Be sure to only profile if it's enabled
    if (Kohana::$profiling === TRUE)
    {
        // Start a new benchmark
        $benchmark = Profiler::start('Your Category', __FUNCTION__);
    }

    // Do some stuff

    if (isset($benchmark))
    {
        // Stop the benchmark
        Profiler::stop($benchmark);
    }

    return $something;
}
```

How to read the profiling report

The benchmarks are sorted into groups. Each benchmark will show its name, how many times it was run (show in parenthesis after the benchmark name), and then the min, max, average, and total time and memory spent on that benchmark. The total column will have shaded backgrounds to show the relative times between benchmarks in the same group.

At the very end is a group called "Application Execution". This keeps track of how long each execution has taken. The number in parenthesis is how many executions are being compared. It shows the fastest, slowest, and average time and memory usage of the last several requsets. The last box is the time and memory usage of the current request.

((This could use a picture of a profiler with some database queries, etc. with annotations to point out each area as just described.))

Displaying the profiler

You can display or collect the current profiler statistics at any time:

```
<?php echo View::factory('profiler/stats') ?>
```

Preview

(This is the actual profiler stats for this page.)

Requests				0.020559 s 1,654.2813 kB
BENCHMARK	MIN	MAX	AVERAGE	TOTAL
"guide/kohana/profiling" (1)	**0.020784** s 1,661.9531 kB	**0.020784** s 1,661.9531 kB	**0.020784** s 1,661.9531 kB	**0.020784** s 1,661.9531 kB

Kohana				0.002239 s 15.6172 kB
BENCHMARK	MIN	MAX	AVERAGE	TOTAL
find_file (28)	**0.000057** s 0.5234 kB	**0.000223** s 0.6406 kB	**0.000080** s 0.5578 kB	**0.002239** s 15.6172 kB

Application Execution (341)				
	0.032835 s 2,998.4453 kB	**0.293552** s 3,000.7656 kB	**0.048119** s 3,000.2100 kB	**0.044976** s 2,999.0078 kB

General security concerns, like using the Security class, CSRF, and a brief intro to XSS, database security, etc. Also mention the security features that Kohana provides, like cleaning globals.

Validation

This page needs to be reviewed for accuracy by the development team. Better examples would be helpful.

Validation can be performed on any array using the Validation class. Labels and rules can be attached to a Validation object by the array key, called a "field name".

labels
: A label is a human-readable version of the field name.

rules
: A rule is a callback used to decide whether or not to add an error to a field

Note that any valid PHP callback can be used as a rule.

Using `TRUE` as the field name when adding a rule will be applied to all named fields.

Creating a validation object is done using the Validation::factory method:

```
$post = Validation::factory($_POST);
```

The `$post` object will be used for the rest of this tutorial. This tutorial will show you how to validate the registration of a new user.

Provided Rules

Kohana provides a set of useful rules in the Valid class:

Rule name	Function
Valid::not_empty	Value must be a non-empty value
Valid::regex	Match the value against a regular expression
Valid::min_length	Minimum number of characters for value
Valid::max_length	Maximum number of characters for value
Valid::exact_length	Value must be an exact number of characters
Valid::email	An email address is required
Valid::email_domain	Check that the domain of the email exists
Valid::url	Value must be a URL
Valid::ip	Value must be an IP address
Valid::phone	Value must be a phone number
Valid::credit_card	Require a credit card number
Valid::date	Value must be a date (and time)
Valid::alpha	Only alpha characters allowed
Valid::alpha_dash	Only alpha and hyphens allowed
Valid::alpha_numeric	Only alpha and numbers allowed
Valid::digit	Value must be an integer digit
Valid::decimal	Value must be a decimal or float value
Valid::numeric	Only numeric characters allowed
Valid::range	Value must be within a range
Valid::color	Value must be a valid HEX color
Valid::matches	Value matches another field value

Adding Rules

All validation rules are defined as a field name, a method or function (using the PHP callback syntax), and an array of parameters:

```
$object->rule($field, $callback, array($parameter1, $parameter2));
```

If no parameters are specified, the field value will be passed to the callback. The following two rules are equivalent:

```
$object->rule($field, 'not_empty');
$object->rule($field, 'not_empty', array(':value'));
```

Rules defined in the Valid class can be added by using the method name alone. The following three rules are equivalent:

```
$object->rule('number', 'phone');
$object->rule('number', array('Valid', 'phone'));
$object->rule('number', 'Valid::phone');
```

Binding Variables

The Validation class allows you to bind variables to certain strings so that they can be used when defining rules. Variables are bound by calling the Validation::bind method.

```
$object->bind(':model', $user_model);
// Future code will be able to use :model to reference the object
$object->rule('username', 'some_rule', array(':model'));
```

By default, the validation object will automatically bind the following values for you to use as rule parameters:

- `:validation` - references the validation object
- `:field` - references the field name the rule is for
- `:value` - references the value of the field the rule is for

Adding Errors

The Validation class will add an error for a field if any of the rules associated to it return `FALSE`. This allows many built in PHP functions to be used as rules, like `in_array`.

```
$object->rule('color', 'in_array', array(':value', array('red', 'green', 'blue')));
```

Rules added to empty fields will run, but returning `FALSE` will not automatically add an error for the field. In order for a rule to affect empty fields, you must add the error manually by calling the Validation::error method. In order to do this, you must pass the validation object to the rule.

```
$object->rule($field, 'the_rule', array(':validation', ':field'));

public function the_rule($validation, $field)
{
    if (something went wrong)
    {
        $validation->error($field, 'the_rule');
    }
}
```

`not_empty` and `matches` are the only rules that will run on empty fields and add errors by returning `FALSE`.

Example

To start our example, we will perform validation on a `$_POST` array that contains user registration information:

```
$post = Validation::factory($_POST);
```

Next we need to process the POST'ed information using Validation. To start, we need to add some rules:

```
$post
    ->rule('username', 'not_empty')
    ->rule('username', 'regex', array(':value', '/^[a-z_.]++$/iD'))
    ->rule('password', 'not_empty')
    ->rule('password', 'min_length', array(':value', '6'))
    ->rule('confirm',  'matches', array(':validation', 'confirm', 'password'))
    ->rule('use_ssl', 'not_empty');
```

Any existing PHP function can also be used a rule. For instance, if we want to check if the user entered a proper value for the SSL question:

```
$post->rule('use_ssl', 'in_array', array(':value', array('yes', 'no')));
```

Note that all array parameters must still be wrapped in an array! Without the wrapping array, `in_array` would be called as `in_array($value, 'yes', 'no')`, which would result in a PHP error.

Any custom rules can be added using a [PHP callback](http://php.net/manual/language.pseudo-types.php#language.types.callback]:

```
$post->rule('username', 'User_Model::unique_username');
```

The method `User_Model::unique_username()` would be defined similar to:

```
public static function unique_username($username)
{
    // Check if the username already exists in the database
    return ! DB::select(array(DB::expr('COUNT(username)'), 'total'))
        ->from('users')
        ->where('username', '=', $username)
        ->execute()
        ->get('total');
```

```
}
```

Custom rules allow many additional checks to be reused for multiple purposes. These methods will almost always exist in a model, but may be defined in any class.

A Complete Example

First, we need a View that contains the HTML form, which will be placed in `application/views/user/register.php`:

```
<?php echo Form::open() ?>
<?php if ($errors): ?>
<p class="message">Some errors were encountered, please check the details you entered.</p>
<ul class="errors">
<?php foreach ($errors as $message): ?>
    <li><?php echo $message ?></li>
<?php endforeach ?>
<?php endif ?>

<dl>
    <dt><?php echo Form::label('username', 'Username') ?></dt>
    <dd><?php echo Form::input('username', $post['username']) ?></dd>

    <dt><?php echo Form::label('password', 'Password') ?></dt>
    <dd><?php echo Form::password('password') ?></dd>
    <dd class="help">Passwords must be at least 6 characters long.</dd>
    <dt><?php echo Form::label('confirm', 'Confirm Password') ?></dt>
    <dd><?php echo Form::password('confirm') ?></dd>

    <dt><?php echo Form::label('use_ssl', 'Use extra security?') ?></dt>
    <dd><?php echo Form::select('use_ssl', array('yes' => 'Always', 'no' => 'Only when necessar
    <dd class="help">For security, SSL is always used when making payments.</dd>
</dl>

<?php echo Form::submit(NULL, 'Sign Up') ?>
<?php echo Form::close() ?>
```

This example uses the Form helper extensively. Using Form instead of writing HTML ensures that all of the form inputs will properly handle input that includes HTML characters. If you prefer to write the HTML yourself, be sure to use HTML::chars to escape user input.

Next, we need a controller and action to process the registration, which will be placed in `application/classes/controller/user.php`:

```
class Controller_User extends Controller {

    public function action_register()
    {
        $user = Model::factory('user');

        $post = Validation::factory($_POST)
            ->rule('username', 'not_empty')
            ->rule('username', 'regex', array(':value', '/^[a-z_.]++$/iD'))
            ->rule('username', array($user, 'unique_username'))

            ->rule('password', 'not_empty')
            ->rule('password', 'min_length', array(':value', 6))
            ->rule('confirm',  'matches', array(':validation', ':field', 'password'))

            ->rule('use_ssl', 'not_empty')
            ->rule('use_ssl', 'in_array', array(':value', array('yes', 'no')));

        if ($post->check())
        {
```

```
            // Data has been validated, register the user
            $user->register($post);

            // Always redirect after a successful POST to prevent refresh warnings
            $this->request->redirect('user/profile');
        }

        // Validation failed, collect the errors
        $errors = $post->errors('user');

        // Display the registration form
        $this->response->body(View::factory('user/register'))
            ->bind('post', $post)
            ->bind('errors', $errors);
    }

}
```

We will also need a user model, which will be placed in `application/classes/model/user.php`:

```
class Model_User extends Model {

    public function register($array)
    {
        // Create a new user record in the database
        $id = DB::insert(array_keys($array))
            ->values($array)
            ->execute();

        // Save the new user id to a cookie
        cookie::set('user', $id);

        return $id;
    }

}
```

That is it, we have a complete user registration example that properly checks user input!

Discuss security of cookies, like changing the encryption key in the config.

Not sure why I'm linking to this: http://kohanaframework.org/guide/security.cookies

Discuss database security.

How to avoid injection, etc.

Not sure why I'm linking to this: http://kohanaframework.org/guide/security.database

Discuss using encryption, including setting the encryption key in config.

Changes that should happen when you deploy. (Production)

Security settings from: http://kohanaframework.org/guide/using.configuration

http://kerkness.ca/wiki/doku.php?id=setting_up_production_environment

Setting up a production environment

There are a few things you'll want to do with your application before moving into production.

1. See the Bootstrap page in the docs. This covers most of the global settings that would change between environments. As a general rule, you should enable caching and disable profiling (Kohana::init settings) for production sites. Route::cache can also help if you have a lot of routes.

2. Turn on APC or some kind of opcode caching. This is the single easiest performance boost you can make to PHP itself. The more complex your application, the bigger the benefit of using opcode caching.

```
    /**
     * Set the environment string by the domain (defaults to Kohana::DEVELOPMENT).
     */
    Kohana::$environment = ($_SERVER['SERVER_NAME'] !== 'localhost') ? Kohana::PRODUCTION : Koh
    /**
     * Initialise Kohana based on environment
     */
    Kohana::init(array(
        'base_url'   => '/',
        'index_file' => FALSE,
        'profile'    => Kohana::$environment !== Kohana::PRODUCTION,
        'caching'    => Kohana::$environment === Kohana::PRODUCTION,
    ));
```

index.php:

```
    /**
     * Execute the main request using PATH_INFO. If no URI source is specified,
     * the URI will be automatically detected.
     */
    $request = Request::instance($_SERVER['PATH_INFO']);

    // Attempt to execute the response
    $request->execute();

    if ($request->send_headers()->response)
    {
        // Get the total memory and execution time
        $total = array(
          '{memory_usage}' => number_format((memory_get_peak_usage() - KOHANA_START_MEMORY) / 1
          '{execution_time}' => number_format(microtime(TRUE) - KOHANA_START_TIME, 5).' seconds

        // Insert the totals into the response
        $request->response = str_replace(array_keys($total), $total, $request->response);
    }


    /**
     * Display the request response.
     */
    echo $request->response;
```

Tutorials

Tutorials in this guide

Tutorials written elsewhere

Ellisgl's KO3 tutorial on dealtaker.com:

1. Install and Basic Usage
2. Views
3. Controllers
4. Models
5. Subrequests
6. Routes
7. Helpers
8. Modules
9. Vendor Libraries

Simple example of controller model and view working together.

Friendly Error Pages

By default Kohana 3 doesn't have a method to display friendly error pages like that seen in Kohana 2; In this short guide you will learn how it is done.

Prerequisites

You will need `'errors' => TRUE` passed to `Kohana::init`. This will convert PHP errors into exceptions which are easier to handle.

1. An Improved Exception Handler

Our custom exception handler is self explanatory.

```
class Kohana_Exception extends Kohana_Kohana_Exception {

    public static function handler(Exception $e)
    {
        if (Kohana::DEVELOPMENT === Kohana::$environment)
        {
            parent::handler($e);
        }
        else
        {
            try
            {
                Kohana::$log->add(Log::ERROR, parent::text($e));

                $attributes = array
                (
                    'action'  => 500,
                    'message' => rawurlencode($e->getMessage())
                );

                if ($e instanceof HTTP_Exception)
                {
                    $attributes['action'] = $e->getCode();
                }

                // Error sub-request.
                echo Request::factory(Route::get('error')->uri($attributes))
                ->execute()
                ->send_headers()
                ->body();
            }
            catch (Exception $e)
            {
                // Clean the output buffer if one exists
                ob_get_level() and ob_clean();

                // Display the exception text
                echo parent::text($e);

                // Exit with an error status
                exit(1);
            }
        }
    }
}
```

If we are in the development environment then pass it off to Kohana otherwise:

- Log the error

- Set the route action and message attributes.
- If a `HTTP_Exception` was thrown, then override the action with the error code.
- Fire off an internal sub-request.

The action will be used as the HTTP response code. By default this is: 500 (internal server error) unless a `HTTP_Response_Exception` was thrown.

So this:

```
throw new HTTP_Exception_404(':file does not exist', array(':file' => 'Gaia'));
```

would display a nice 404 error page, where:

```
throw new Kohana_Exception('Directory :dir must be writable',
            array(':dir' => Debug::path(Kohana::$cache_dir)));
```

would display an error 500 page.

The Route

```
Route::set('error', 'error/<action>(/<message>)', array('action' => '[0-9]++', 'message' => '.+
->defaults(array(
    'controller' => 'error_handler'
));
```

2. The Error Page Controller

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

    $this->template->page = URL::site(rawurldecode(Request::$initial->uri()));

    // Internal request only!
    if (Request::$initial !== Request::$current)
    {
        if ($message = rawurldecode($this->request->param('message')))
        {
            $this->template->message = $message;
        }
    }
    else
    {
        $this->request->action(404);
    }

    $this->response->status((int) $this->request->action());
}
```

1. Set a template variable "page" so the user can see what they requested. This is for display purposes only.
2. If an internal request, then set a template variable "message" to be shown to the user.

3. Otherwise use the 404 action. Users could otherwise craft their own error messages, eg:
   `error/404/email%20your%20login%20information%20to%20hacker%40google.com`

   ```
   public function action_404()
   {
       $this->template->title = '404 Not Found';

       // Here we check to see if a 404 came from our website. This allows the
       // webmaster to find broken links and update them in a shorter amount of time.
       if (isset ($_SERVER['HTTP_REFERER']) AND strstr($_SERVER['HTTP_REFERER'], $_SERVER['SE
       {
           // Set a local flag so we can display different messages in our template.
           $this->template->local = TRUE;
       }
   ```

```
        // HTTP Status code.
        $this->response->status(404);
    }

    public function action_503()
    {
        $this->template->title = 'Maintenance Mode';
    }

    public function action_500()
    {
        $this->template->title = 'Internal Server Error';
    }
```

You will notice that each example method is named after the HTTP response code and sets the request response code.

3. Conclusion

So that's it. Now displaying a nice error page is as easy as:

```
throw new HTTP_Exception_503('The website is down');
```

http://kerkness.ca/wiki/doku.php?id=routing:static_pages

http://kerkness.ca/wiki/doku.php?id=routing:multi-language_with_a_route

Clean URLs

Removing `index.php` from your urls.

To keep your URLs clean, you will probably want to be able to access your app without having `/index.php/` in the URL. There are two steps to remove `index.php` from the URL.

1. Edit the bootstrap file
2. Set up rewriting

1. Configure Bootstrap

The first thing you will need to change is the `index_file` setting of Kohana::init to false:

```
Kohana::init(array(
    'base_url'   => '/myapp/',
    'index_file' => FALSE,
));
```

This change will make it so all of the links generated using URL::site, URL::base, and HTML::anchor will no longer include "index.php" in the URL. All generated links will start with `/myapp/` instead of `/myapp/index.php/`.

2. URL Rewriting

Enabling rewriting is done differently, depending on your web server.

Rewriting will make it so urls will be passed to index.php.

Apache

Rename `example.htaccess` to only `.htaccess` and alter the `RewriteBase` line to match the `base_url` setting from your Kohana::init

```
RewriteBase /myapp/
```

The rest of the `.htaccess file` rewrites all requests through index.php, unless the file exists on the server (so your

css, images, favicon, etc. are still loaded like normal). In most cases, you are done!

Failed!

If you get a "Internal Server Error" or "No input file specified" error, try changing:

```
RewriteRule ^(?:application|modules|system)\b - [F,L]
```

Instead, we can try a slash:

```
RewriteRule ^(application|modules|system)/ - [F,L]
```

If that doesn't work, try changing:

```
RewriteRule .* index.php/$0 [PT]
```

To something more simple:

```
RewriteRule .* index.php [PT]
```

Still Failed!

If you are still getting errors, check to make sure that your host supports URL `mod_rewrite`. If you can change the Apache configuration, add these lines to the the configuration, usually `httpd.conf`:

```
<Directory "/var/www/html/myapp">
    Order allow,deny
    Allow from all
    AllowOverride All
</Directory>
```

You should also check your Apache logs to see if they can shed some light on the error.

NGINX

It is hard to give examples of nginx configuration, but here is a sample for a server:

```
location / {
    index     index.php index.html index.htm;
    try_files $uri index.php;
}

location = index.php {
    include       fastcgi.conf;
    fastcgi_pass  127.0.0.1:9000;
    fastcgi_index index.php;
}
```

If you are having issues getting this working, enable debug level logging in nginx and check the access and error logs.

Sharing Kohana

Because Kohana follows a [front controller] pattern, which means that all requests are sent to `index.php`, the filesystem is very configurable. Inside of `index.php` you can change the `$application`, `$modules`, and `$system` paths.

There is a security check at the top of every Kohana file to prevent it from being accessed without using the front controller. Also, the `.htaccess` file should protect those folders as well. Moving the application, modules, and system directories to a location that cannot be accessed vie web can add another layer of security, but is optional.

The `$application` variable lets you set the directory that contains your application files. By default, this is `application`. The `$modules` variable lets you set the directory that contains module files. The `$system` variable lets you set the directory that contains the default Kohana files. You can move these three directories anywhere.

For instance, by default the directories are set up like this:

```
www/
    index.php
    application/
    modules/
    system/
```

You could move the directories out of the web root so they look like this:

```
application/
modules/
system/
www/
    index.php
```

Then you would need to change the settings in `index.php` to be:

```
$application = '../application';
$modules     = '../modules';
$system      = '../system';
```

Sharing system and modules

To take this a step further, we could point several kohana apps to the same system and modules folders. For example (and this is just an example, you could arrange these anyway you want):

```
apps/
    foobar/
        application/
        www/
    bazbar/
        application/
        www/
kohana/
    3.0.6/
    3.0.7/
    3.0.8/
modules/
```

And you would need to change the settings in `index.php` to be:

```
$application = '../application';
$system      = '../../../kohana/3.0.6';
$modules     = '../../../kohana/modules';
```

Using this method each app can point to a central copy of kohana, and you can add a new version, and quickly update your apps to point to the new version by editing their `index.php` files.

Making a template driven site.

http://kerkness.ca/wiki/doku.php?id=template-site:create_the_template

http://kerkness.ca/wiki/doku.php?id=template-site:extending_the_template_controller

http://kerkness.ca/wiki/doku.php?id=template-site:basic_page_controller

Creating a New Application

The following examples assume that your web server is already set up, and you are going to create a new application at http://localhost/gitorial/.

Using your console, change to the empty directory `gitorial` and run `git init`. This will create the bare structure for a new git repository.

Next, we will create a submodule for the `system` directory. Go to http://github.com/kohana/core and copy the "Clone URL":

Now use the URL to create the submodule for `system`:

```
git submodule add git://github.com/kohana/core.git system
```

This will create a link to the current development version of the next stable release. The development version should almost always be safe to use, have the same API as the current stable download with bugfixes applied.

Now add whatever submodules you need. For example, if you need the Database module:

```
git submodule add git://github.com/kohana/database.git modules/database
```

After submodules are added, they must be initialized:

```
git submodule init
```

Now that the submodules are added, you can commit them:

```
git commit -m 'Added initial submodules'
```

Next, create the application directory structure. This is the bare minimum required:

```
mkdir -p application/classes/{controller,model}
mkdir -p application/{config,views}
mkdir -m 0777 -p application/{cache,logs}
```

If you run `find application` you should see this:

```
application
application/cache
application/config
application/classes
application/classes/controller
application/classes/model
application/logs
application/views
```

We don't want git to track log or cache files, so add a `.gitignore` file to each of the directories. This will ignore all non-hidden files:

```
echo '[^.]*' > application/{logs,cache}/.gitignore
```

Git ignores empty directories, so adding a `.gitignore` file also makes sure that git will track the directory, but not the files within it.

Now we need the `index.php` and `bootstrap.php` files:

```
wget https://github.com/kohana/kohana/raw/3.2/master/index.php --no-check-certificate
wget https://github.com/kohana/kohana/raw/3.2/master/application/bootstrap.php --no-check-certi
```

Commit these changes too:

```
git add application
git commit -m 'Added initial directory structure'
```

That's all there is to it. You now have an application that is using Git for versioning.

Adding Submodules

To add a new submodule complete the following steps:

1. run the following code - git submodule add repository path for each new submodule e.g.:

   ```
   git submodule add git://github.com/shadowhand/sprig.git modules/sprig
   ```

2. then init and update the submodules:

```
git submodule init
git submodule update
```

Updating Submodules

At some point you will probably also want to upgrade your submodules. To update all of your submodules to the latest `HEAD` version:

```
git submodule foreach 'git checkout 3.2/master && git pull origin 3.2/master'
```

To update a single submodule, for example, `system`:

```
cd system
git checkout 3.2/master
git pull origin 3.2/master
cd ..
git add system
git commit -m 'Updated system to latest version'
```

If you want to update a single submodule to a specific commit:

```
cd modules/database
git pull origin 3.2/master
git checkout fbfdea919028b951c23c3d99d2bc1f5bbeda0c0b
cd ../..
git add database
git commit -m 'Updated database module'
```

Note that you can also check out the commit at a tagged official release point, for example:

```
git checkout v3.2.0
```

Simply run `git tag` without arguments to get a list of all tags.

Removing Submodules

To remove a submodule that is no longer needed complete the following steps:

1. open .gitmodules and remove the reference to the to submodule It will look something like this:

   ```
   [submodule "modules/auth"]
   path = modules/auth
   url = git://github.com/kohana/auth.git
   ```

2. open .git/config and remove the reference to the to submodule\

   ```
   [submodule "modules/auth"]
   url = git://github.com/kohana/auth.git
   ```

3. run git rm --cached path/to/submodule, e.g.

   ```
   git rm --cached modules/auth
   ```

Note: Do not put a trailing slash at the end of path. If you put a trailing slash at the end of the command, it will fail.

Updating Remote Repository URL

During the development of a project, the source of a submodule may change for any reason (you've created your own fork, the server URL changed, the repository name or path changed, etc...) and you'll have to update those changes. To do so, you'll need to perform the following steps:

1. edit the .gitmodules file, and change the URL for the submodules which changed.

2. in your source tree's root run:

   ```
   git submodule sync
   ```

3. run `git init` to update the project's repository configuration with the new URLs:

```
git submodule init
```

And it's done, now you can continue pushing and pulling your submodules with no problems.

Source: http://jtrancas.wordpress.com/2011/02/06/git-submodule-location/

Auth

User authentication and authorization is provided by the auth module.

The auth module is included with the Kohana 3.0 install, but needs to be enabled before you can use it. To enable, open your `application/bootstrap.php` file and modify the call to Kohana::modules by including the auth module like so:

```
Kohana::modules(array(
    ...
    'auth' => MODPATH.'auth',
    ...
));
```

Next, you will then need to configure the auth module.

The auth module provides the Auth::File driver for you. There is also an auth driver included with the ORM module.

As your application needs change you may need to find another driver or develop your own.

Configuration

The default configuration file is located in `MODPATH/auth/config/auth.php`. You should copy this file to `APPPATH/config/auth.php` and make changes there, in keeping with the cascading filesystem.

Config merging allows these default configuration settings to apply if you don't overwrite them in your application configuration file.

Name	Type	Default	Description
driver	`string`	file	The name of the auth driver to use.
hash_method	`string`	sha256	The hashing function to use on the passwords.
hash_key	`string`	NULL	The key to use when hashing the password.
lifetime	`int`	1209600	The time (*in seconds*) that the user's session is valid.
session_type	`string`	Session::$default	The type of session to use when storing the auth user.
session_key	`string`	auth_user	The name of the session variable used to save the user.

Log in and out

The auth module provides methods to help you log users in and out of your application.

Log in

The Auth::login method handles the login.

```
// Handled from a form with inputs with names email / password
$post = $this->request->post();
$success = Auth::instance()->login($post['email'], $post['password']);

if ($success)
{
    // Login successful, send to app
}
else
{
```

```
    // Login failed, send back to form with error message
}
```

Logged in User

There are two ways to check if a user is logged in. If you just need to check if the user is logged in use Auth::logged_in.

```
if (Auth::instance()->logged_in())
{
    // User is logged in, continue on
}
else
{
    // User isn't logged in, redirect to the login form.
}
```

You can also get the logged in user object by using Auth::get_user. If the user is null, then no user was found.

```
$user = Auth::instance()->get_user();

// Check for a user (NULL if not user is found)
if ($user !== null)
{
     // User is found, continue on
}
else
{
    // User was not found, redirect to the login form
}
```

Log out

The Auth::logout method will take care of logging out a user.

```
Auth::instance()->logout();
// Redirect the user back to login page
```

File Driver

The Auth::File driver is included with the auth module.

Below are additional configuration options that can be set for this driver.

Name	Type	Default	Description
users	`array`	array()	A user => password (*hashed*) array of all the users in your application

Forcing Login

Auth_File::force_login allows you to force a user login without a password.

```
// Force the user with a username of admin to be logged into your application
Auth::instance()->force_login('admin');
$user = Auth::instance()->get_user();
```

Developing Drivers

Real World Example

Sometimes the best way to learn is to jump right in and read the code from another module. The ORM module comes with an auth driver you can learn from.

We will be developing an `example` driver. In your own driver you will substitute `example` with your driver name.

This example file would be saved at `APPPATH/classes/auth/example.php` (or `MODPATH` if you are creating a module).

Quick Example

First we will show you a quick example and then break down what is going on.

```
class Auth_Example extends Auth
{
    protected function _login($username, $password, $remember)
    {
        // Do username/password check here
    }

    public function password($username)
    {
        // Return the password for the username
    }

    public function check_password($password)
    {
        // Check to see if the logged in user has the given password
    }

    public function logged_in($role = NULL)
    {
        // Check to see if the user is logged in, and if $role is set, has all roles
    }

    public function get_user($default = NULL)
    {
        // Get the logged in user, or return the $default if a user is not found
    }
}
```

Extending Auth

All drivers must extend the Auth class.

```
class Auth_Example extends Auth
```

Abstract Methods

The `Auth` class has 3 abstract methods that must be defined in your new driver.

```
abstract protected function _login($username, $password, $remember);

abstract public function password($username);

abstract public function check_password($user);
```

Extending Functionality

Given that every auth system is going to check if users exist and if they have roles or not you will more than likely have to change some default functionality.

Here are a few functions that you should pay attention to.

```
public function logged_in($role = NULL)

public function get_user($default = NULL)
```

Activating the Driver

After you create your driver you will want to use it. It is a easy as setting the `driver` configuration option to the name of your driver (in our case `example`).

About Kohana Cache

Kohana_Cache provides a common interface to a variety of caching engines. Cache_Tagging is supported where available natively to the cache system. Kohana Cache supports multiple instances of cache engines through a grouped singleton pattern.

Supported cache engines

- APC (Cache_Apc)
- File (Cache_File)
- Memcached (Cache_Memcache)
- Memcached-tags (Cache_Memcachetag)
- SQLite (Cache_Sqlite)
- Wincache

Introduction to caching

Caching should be implemented with consideration. Generally, caching the result of resources is faster than reprocessing them. Choosing what, how and when to cache is vital. PHP APC is one of the fastest caching systems available, closely followed by Memcached. SQLite and File caching are two of the slowest cache methods, however usually faster than reprocessing a complex set of instructions.

Caching engines that use memory are considerably faster than file based alternatives. But memory is limited whereas disk space is plentiful. If caching large datasets, such as large database result sets, it is best to use file caching.

[!!] Cache drivers require the relevant PHP extensions to be installed. APC, eAccelerator, Memecached and Xcache all require non-standard PHP extensions.

What the Kohana Cache module does (and does not do)

This module provides a simple abstracted interface to a wide selection of popular PHP cache engines. The caching API provides the basic caching methods implemented across all solutions, memory, network or disk based. Basic key / value storing is supported by all drivers, with additional tagging and garbage collection support where implemented or required.

Kohana Cache does not provide HTTP style caching for clients (web browsers) and/or proxies (*Varnish*, *Squid*). There are other Kohana modules that provide this functionality.

Choosing a cache provider

Getting and setting values to cache is very simple when using the *Kohana Cache* interface. The hardest choice is choosing which cache engine to use. When choosing a caching engine, the following criteria must be considered:

1. **Does the cache need to be distributed?** This is an important consideration as it will severely limit the options available to solutions such as Memcache when a distributed solution is required.
2. **Does the cache need to be fast?** In almost all cases retrieving data from a cache is faster than execution. However generally memory based caching is considerably faster than disk based caching (see table below).
3. **How much cache is required?** Cache is not endless, and memory based caches are subject to a considerably more limited storage resource.

Driver	Storage	Speed	Tags	Distributed	Automatic Garbage Collection	Notes
APC	**Memory**	Excellent	No	No	Yes	Widely available PHP opcode caching solution, improves php execution performance
Wincache	**Memory**	Excellent	No	No	Yes	Windows variant of APC
File	**Disk**	Poor	No	No	No	Marginally faster than execution

Memcache (tag)	**Memory**	Good	No (yes)	Yes	Yes	Generally fast distributed solution, but has a speed hit due to variable network latency and serialization
Sqlite	**Disk**	Poor	Yes	No	No	Marginally faster than execution

It is possible to have hybrid cache solutions that use a combination of the engines above in different contexts. This is supported with *Kohana Cache* as well

Minimum requirements

- Kohana 3.0.4
- PHP 5.2.4 or greater

Kohana Cache configuration

Kohana Cache uses configuration groups to create cache instances. A configuration group can use any supported driver, with successive groups using multiple instances of the same driver type.

The default cache group is loaded based on the `Cache::$default` setting. It is set to the `file` driver as standard, however this can be changed within the `/application/boostrap.php` file.

```
// Change the default cache driver to memcache
Cache::$default = 'memcache';

// Load the memcache cache driver using default setting
$memcache = Cache::instance();
```

Group settings

Below are the default cache configuration groups for each supported driver. Add to- or override these settings within the `application/config/cache.php` file.

Name	Required	Description
driver	**YES**	(*string*) The driver type to use
default_expire	**NO**	(*string*) The driver type to use

```
'file'  => array
(
    'driver'             => 'file',
    'cache_dir'          => APPPATH.'cache/.kohana_cache',
    'default_expire'     => 3600,
),
```

Memcache & Memcached-tag settings

Name	Required	Description
driver	**YES**	(*string*) The driver type to use
servers	**YES**	(*array*) Associative array of server details, must include a **host** key. (see *Memcache server configuration* below)
compression	**NO**	(*boolean*) Use data compression when caching

Memcache server configuration

Name	Required	Description
host	**YES**	(*string*) The host of the memcache server, i.e. **localhost**; or **127.0.0.1**; or **memcache.domain.tld**
port	**NO**	(*integer*) Point to the port where memcached is listening for connections. Set this parameter to 0 when using UNIX domain sockets. Default **11211**
persistent	**NO**	(*boolean*) Controls the use of a persistent connection. Default **TRUE**
weight	**NO**	(*integer*) Number of buckets to create for this server which in turn control its probability of it being selected. The probability is relative to the total weight of all servers. Default **1**
		(*integer*) Value in seconds which will be used for connecting to the daemon. Think twice

timeout	**NO**	(*integer*) Value in seconds which will be used for connecting to the daemon. Think twice before changing the default value of 1 second - you can lose all the advantages of caching if your connection is too slow. Default **1**
retry_interval	**NO**	(*integer*) Controls how often a failed server will be retried, the default value is 15 seconds. Setting this parameter to -1 disables automatic retry. Default **15**
status	**NO**	(*boolean*) Controls if the server should be flagged as online. Default **TRUE**
failure_callback	**NO**	(*callback*) Allows the user to specify a callback function to run upon encountering an error. The callback is run before failover is attempted. The function takes two parameters, the hostname and port of the failed server. Default **NULL**

```
'memcache' => array
(
    'driver'             => 'memcache',
    'default_expire'     => 3600,
    'compression'        => FALSE,              // Use Zlib compression
                                                (can cause issues with integers)
    'servers'            => array
    (
        array
        (
            'host'             => 'localhost',  // Memcache Server
            'port'             => 11211,        // Memcache port number
            'persistent'       => FALSE,        // Persistent connection
        ),
    ),
),
'memcachetag' => array
(
    'driver'             => 'memcachetag',
    'default_expire'     => 3600,
    'compression'        => FALSE,              // Use Zlib compression
                                                (can cause issues with integers)
    'servers'            => array
    (
        array
        (
            'host'             => 'localhost',  // Memcache Server
            'port'             => 11211,        // Memcache port number
            'persistent'       => FALSE,        // Persistent connection
        ),
    ),
),
```

APC settings

```
'apc'      => array
(
    'driver'             => 'apc',
    'default_expire'     => 3600,
),
```

SQLite settings

```
'sqlite'   => array
(
    'driver'             => 'sqlite',
    'default_expire'     => 3600,
    'database'           => APPPATH.'cache/kohana-cache.sql3',
    'schema'             => 'CREATE TABLE caches(id VARCHAR(127) PRIMARY KEY,
                                 tags VARCHAR(255), expiration INTEGER, cache TEXT)',
),
```

File settings

```
'file'    => array
(
    'driver'             => 'file',
    'cache_dir'          => 'cache/.kohana_cache',
    'default_expire'     => 3600,
)
```

Wincache settings

```
'wincache' => array
(
    'driver'             => 'wincache',
    'default_expire'     => 3600,
),
```

Override existing configuration group

The following example demonstrates how to override an existing configuration setting, using the config file in `/application/config/cache.php`.

```
<?php defined('SYSPATH') or die('No direct script access.');
return array
(
    // Override the default configuration
    'memcache'   => array
    (
        'driver'         => 'memcache',  // Use Memcached as the default driver
        'default_expire' => 8000,        // Overide default expiry
        'servers'        => array
        (
            // Add a new server
            array
            (
                'host'       => 'cache.domain.tld',
                'port'       => 11211,
                'persistent' => FALSE
            )
        ),
        'compression'    => FALSE
    )
);
```

Add new configuration group

The following example demonstrates how to add a new configuration setting, using the config file in `/application/config/cache.php`.

```
<?php defined('SYSPATH') or die('No direct script access.');
return array
(
    // Override the default configuration
    'fastkv'   => array
    (
        'driver'         => 'apc',  // Use Memcached as the default driver
        'default_expire' => 1000,   // Overide default expiry
    )
);
```

Kohana Cache usage

Kohana_Cache provides a simple interface allowing getting, setting and deleting of cached values. Two interfaces included in *Kohana Cache* additionally provide *tagging* and *garbage collection* where they are supported by the

respective drivers.

Getting a new cache instance

Creating a new *Kohana Cache* instance is simple, however it must be done using the Cache::instance method, rather than the traditional `new` constructor.

```
// Create a new instance of cache using the default group
$cache = Cache::instance();
```

The default group will use whatever is set to Cache::$default and must have a corresponding configuration definition for that group.

To create a cache instance using a group other than the *default*, simply provide the group name as an argument.

```
// Create a new instance of the memcache group
$memcache = Cache::instance('memcache');
```

If there is a cache instance already instantiated then you can get it directly from the class member.

[!!] Beware that this can cause issues if you do not test for the instance before trying to access it.

```
// Check for the existance of the cache driver
if (isset(Cache::$instances['memcache']))
{
     // Get the existing cache instance directly (faster)
     $memcache = Cache::$instances['memcache'];
}
else
{
     // Get the cache driver instance (slower)
     $memcache = Cache::instance('memcache');
}
```

Setting and getting variables to and from cache

The cache library supports scalar and object values, utilising object serialization where required (or not supported by the caching engine). This means that the majority or objects can be cached without any modification.

[!!] Serialisation does not work with resource handles, such as filesystem, curl or socket resources.

Setting a value to cache

Setting a value to cache using the Cache::set method can be done in one of two ways; either using the Cache instance interface, which is good for atomic operations; or getting an instance and using that for multiple operations.

The first example demonstrates how to quickly load and set a value to the default cache instance.

```
// Create a cachable object
$object = new stdClass;

// Set a property
$object->foo = 'bar';

// Cache the object using default group (quick interface) with default time (3600 seconds)
Cache::instance()->set('foo', $object);
```

If multiple cache operations are required, it is best to assign an instance of Cache to a variable and use that as below.

```
// Set the object using a defined group for a defined time period (30 seconds)
$memcache = Cache::instance('memcache');
$memcache->set('foo', $object, 30);
```

Setting a value with tags

Certain cache drivers support setting values with tags. To set a value to cache with tags using the following interface.

```
// Get a cache instance that supports tags
$memcache = Cache::instance('memcachetag');

// Test for tagging interface
if ($memcache instanceof Cache_Tagging)
{
     // Set a value with some tags for 30 seconds
     $memcache->set('foo', $object, 30, array('snafu', 'stfu', 'fubar'));
}
// Otherwise set without tags
else
{
     // Set a value for 30 seconds
     $memcache->set('foo', $object, 30);
}
```

It is possible to implement custom tagging solutions onto existing or new cache drivers by implementing the Cache_Tagging interface. Kohana_Cache only applies the interface to drivers that support tagging natively as standard.

Getting a value from cache

Getting variables back from cache is achieved using the Cache::get method using a single key to identify the cache entry.

```
// Retrieve a value from cache (quickly)
$object = Cache::instance()->get('foo');
```

In cases where the requested key is not available or the entry has expired, a default value will be returned (**NULL** by default). It is possible to define the default value as the key is requested.

```
// If the cache key is available (with default value set to FALSE)
if ($object = Cache::instance()->get('foo', FALSE))
{
     // Do something
}
else
{
     // Do something else
}
```

Getting values from cache using tags

It is possible to retrieve values from cache grouped by tag, using the Cache::find method with drivers that support tagging.

[!!] The **Memcachetag** driver does not support the `Cache::find($tag)` interface and will throw an exception.

```
// Get an instance of cache
$cache = Cache::instance('memcachetag');

// Wrap in a try/catch statement to gracefully handle memcachetag
try
{
     // Find values based on tag
     return $cache->find('snafu');
}
catch (Cache_Exception $e)
{
     // Handle gracefully
     return FALSE;
}
```

Deleting values from cache

Deleting variables is very similar to the getting and setting methods already described. Deleting operations are split into three categories:

- **Delete value by key**. Deletes a cached value by the associated key.
- **Delete all values**. Deletes all caches values stored in the cache instance.
- **Delete values by tag**. Deletes all values that have the supplied tag. This is only supported by Memcached-Tag and Sqlite.

Delete value by key

To delete a specific value by its associated key:

```
// If the cache entry for 'foo' is deleted
if (Cache::instance()->delete('foo'))
{
     // Cache entry successfully deleted, do something
}
```

By default a `TRUE` value will be returned. However a `FALSE` value will be returned in instances where the key did not exist in the cache.

Delete all values

To delete all values in a specific instance:

```
// If all cache items where deleted successfully
if (Cache::instance()->delete_all())
{
      // Do something
}
```

It is also possible to delete all cache items in every instance:

```
// For each cache instance
foreach (Cache::$instances as $group => $instance)
{
     if ($instance->delete_all())
     {
          var_dump('instance : '.$group.' has been flushed!');
     }
}
```

Delete values by tag

Some of the caching drivers support deleting by tag. This will remove all the cached values that are associated with a specific tag. Below is an example of how to robustly handle deletion by tag.

```
// Get cache instance
$cache = Cache::instance();

// Check for tagging interface
if ($cache instanceof Cache_Tagging)
{
      // Delete all entries by the tag 'snafu'
      $cache->delete_tag('snafu');
}
```

Garbage Collection

Garbage Collection (GC) is the cleaning of expired cache entries. For the most part, caching engines will take care of garbage collection internally. However a few of the file based systems do not handle this task and in these circumstances it would be prudent to garbage collect at a predetermined frequency. If no garbage collection is executed, the resource storing the cache entries will eventually fill and become unusable.

When not automated, garbage collection is the responsibility of the developer. It is prudent to have a GC probability value that dictates how likely the garbage collection routing will be run. An example of such a system is demonstrated below.

```
// Get a cache instance
```

```
$cache_file = Cache::instance('file');

// Set a GC probability of 10%
$gc = 10;

// If the GC probability is a hit
if (rand(0,99) <= $gc and $cache_file instanceof Cache_GarbageCollect)
{
     // Garbage Collect
     $cache_file->garbage_collect();
}
```

Interfaces

Kohana Cache comes with two interfaces that are implemented where the drivers support them:

- **Cache_Tagging for tagging support on cache entries**
  - Cache_MemcacheTag
  - Cache_Sqlite
- **Cache_GarbageCollect for garbage collection with drivers without native support**
  - Cache_File
  - Cache_Sqlite

When using interface specific caching features, ensure that code checks for the required interface before using the methods supplied. The following example checks whether the garbage collection interface is available before calling the `garbage_collect` method.

```
// Create a cache instance
$cache = Cache::instance();

// Test for Garbage Collection
if ($cache instanceof Cache_GarbageCollect)
{
     // Collect garbage
     $cache->garbage_collect();
}
```

Using Codebench

The contents of this page are taken (with some minor changes) from http://www.geertdedeckere.be/article/introducing-codebench and are copyright Geert De Deckere.

For a long time I have been using a quick-and-dirty `benchmark.php` file to optimize bits of PHP code, many times regex-related stuff. The file contained not much more than a gettimeofday function wrapped around a `for` loop. It worked, albeit not very efficiently. Something more solid was needed. I set out to create a far more usable piece of software to aid in the everlasting quest to squeeze every millisecond out of those regular expressions.

Codebench Goals

Benchmark multiple regular expressions at once

Being able to compare the speed of an arbitrary amount of regular expressions would be tremendously useful. In case you are wonderingâ€”yes, I had been writing down benchmark times for each regex, uncommenting them one by one. You get the idea. Those days should be gone forever now.

Benchmark multiple subjects at once

What gets overlooked too often when testing and optimizing regular expressions is the fact that speed can vastly differ depending on the subjects, also known as input or target strings. Just because your regular expression matches, say, a valid email address quickly, does not necessarily mean it will quickly realize when an invalid email is provided. I plan to write a follow-up article with hands-on regex examples to demonstrate this point. Anyway, Codebench allows you to create an array of subjects which will be passed to each benchmark.

Make it flexible enough to work for all PCRE functions

Initially I named the module â€œRegexbenchâ€. I quickly realized, though, it would be flexible enough to benchmark all kinds of PHP code, hence the change to â€œCodebenchâ€. While tools specifically built to help profiling PCRE functions, like preg_match or preg_replace, definitely have their use, more flexibility was needed here. You should be able to compare all kinds of constructions like combinations of PCRE functions and native PHP string functions.

Create clean and portable benchmark cases

Throwing valuable benchmark data away every time I needed to optimize another regular expression had to stop. A clean file containing the complete set of all regex variations to compare, together with the set of subjects to test them against, would be more than welcome. Moreover, it would be easy to exchange benchmark cases with others.

Visualize the benchmarks

Obviously providing a visual representation of the benchmark results, via simple graphs, would make interpreting them easier. Having not to think about Internet Explorer for once, made writing CSS a whole lot more easy and fun. It resulted in some fine graphs which are fully resizable.

Below are two screenshots of Codebench in action. `Valid_Color` is a class made for benchmarking different ways to validate hexadecimal HTML color values, e.g. `#FFF`. If you are interested in the story behind the actual regular expressions, take a look at this topic in the Kohana forums.

 **Benchmarking seven ways to validate HTML color values**

 **Collapsable results per subject for each method**

Working with Codebench

Codebench is included in Kohana 3, but if you need you can download it from GitHub. Be sure Codebench is activated in your `application/bootstrap.php`.

Creating your own benchmarks is just a matter of creating a class that extends the Codebench class. The class should go in `classes/bench` and the class name should have the `Bench_` prefix. Put the code parts you want to compare into separate methods. Be sure to prefix those methods with `bench_`, other methods will not be benchmarked. Glance at the files in `modules/codebench/classes/bench/` for more examples.

Here is another short example with some extra explanations.

```
// classes/bench/ltrimdigits.php
class Bench_LtrimDigits extends Codebench {

    // Some optional explanatory comments about the benchmark file.
    // HTML allowed. URLs will be converted to links automatically.
    public $description = 'Chopping off leading digits: regex vs ltrim.';

    // How many times to execute each method per subject.
    // Total loops = loops * number of methods * number of subjects
    public $loops = 100000;

    // The subjects to supply iteratively to your benchmark methods.
    public $subjects = array
    (
        '123digits',
        'no-digits',
    );

    public function bench_regex($subject)
    {
        return preg_replace('/^\d+/', '', $subject);
    }

    public function bench_ltrim($subject)
    {
```

```
        return ltrim($subject, '0..9');
    }
}
```

And the winner isâ€¦ ltrim. Happy benchmarking!

Database

Kohana 3.0 comes with a robust module for working with databases. By default, the database module supports drivers for MySQL and PDO, but new drivers can be made for other database servers.

The database module is included with the Kohana 3.0 install, but needs to be enabled before you can use it. To enable, open your `application/bootstrap.php` file and modify the call to Kohana::modules by including the database module like so:

```
Kohana::modules(array(
    ...
    'database' => MODPATH.'database',
    ...
));
```

Next, you will then need to configure the database module to connect to your database.

Once that is done then you can make queries and use the results.

The database module also provides a config driver (for storing configuration in the database) and a session driver.

Configuration

The default config file is located in `MODPATH/database/config/database.php`. You should copy this file to `APPPATH/config/database.php` and make changes there, in keeping with the cascading filesystem.

The database configuration file contains an array of configuration groups. The structure of each database configuration group, called an "instance", looks like this:

```
string INSTANCE_NAME => array(
    'type'         => string DATABASE_TYPE,
    'connection'   => array CONNECTION_ARRAY,
    'table_prefix' => string TABLE_PREFIX,
    'charset'      => string CHARACTER_SET,
    'profiling'    => boolean QUERY_PROFILING,
),
```

Understanding each of these settings is important.

INSTANCE_NAME
: Connections can be named anything you want, but you should always have at least one connection called "default".

DATABASE_TYPE
: One of the installed database drivers. Kohana comes with "mysql" and "pdo" drivers. Drivers must extend the Database class.

CONNECTION_ARRAY
: Specific driver options for connecting to your database. (Driver options are explained below.)

TABLE_PREFIX
: Prefix that will be added to all table names by the query builder. Prepared statements will **not** use the table prefix.

QUERY_PROFILING
: Enables profiling of database queries. This is useful for seeing how many queries each page is using, and which are taking the longest. You must enable the profiler the view these stats.

Example

The example file below shows 2 MySQL connections, one local and one remote.

```
return array
(
```

```
    'default' => array
    (
        'type'       => 'mysql',
        'connection' => array(
            'hostname'   => 'localhost',
            'username'   => 'dbuser',
            'password'   => 'mypassword',
            'persistent' => FALSE,
            'database'   => 'my_db_name',
        ),
        'table_prefix' => '',
        'charset'      => 'utf8',
        'profiling'    => TRUE,
    ),
    'remote' => array(
        'type'       => 'mysql',
        'connection' => array(
            'hostname'   => '55.55.55.55',
            'username'   => 'remote_user',
            'password'   => 'mypassword',
            'persistent' => FALSE,
            'database'   => 'my_remote_db_name',
        ),
        'table_prefix' => '',
        'charset'      => 'utf8',
        'profiling'    => TRUE,
    ),
);
```

Connections and Instances

Each configuration group is referred to as a database instance. Each instance can be accessed by calling Database::instance. If you don't provide a parameter, the default instance is used.

```
// This would connect to the database defined as 'default'
$default = Database::instance();

// This would connect to the database defined as 'remote'
$remote  = Database::instance('remote');
```

To disconnect the database, simply destroy the object:

```
unset($default)

// Or

unset(Database::$instances['default']);
```

If you want to disconnect all of the database instances at once:

```
Database::$instances = array();
```

Connection Settings

Every database driver has different connection settings.

MySQL

A MySQL database can accept the following options in the `connection` array:

Type	Option	Description	Default value
`string`	hostname	Hostname of the database	`localhost`
`integer`	port	Port number	`NULL`
`string`	socket	UNIX socket	`NULL`

string	username	Database username	NULL
string	password	Database password	NULL
boolean	persistent	Persistent connections	FALSE
string	database	Database name	kohana

PDO

A PDO database can accept these options in the `connection` array:

Type	Option	Description	Default value
string	dsn	PDO data source identifier	localhost
string	username	Database username	NULL
string	password	Database password	NULL
boolean	persistent	Persistent connections	FALSE

If you are using PDO and are not sure what to use for the `dsn` option, review PDO::__construct.

Making Queries

There are two different ways to make queries. The simplest way to make a query is to use Database_Query, via DB::query, to manually create queries. These queries are called prepared statements and allow you to set query parameters which are automatically escaped. The second way to make a query is by building the query using method calls. This is done using the query builder.

All queries are run using the `execute` method, which accepts a Database object or instance name. See Database_Query::execute for more information.

Query Builder

Creating queries dynamically using objects and methods allows queries to be written very quickly in an agnostic way. Query building also adds identifier (table and column name) quoting, as well as value quoting.

At this time, it is not possible to combine query building with prepared statements.

Select

Each type of database query is represented by a different class, each with their own methods. For instance, to create a SELECT query, we use DB::select which is a shortcut to return a new Database_Query_Builder_Select object:

```
$query = DB::select();
```

Query Builder methods return a reference to itself so that method chaining may be used. Select queries ussually require a table and they are referenced using the `from()` method. The `from()` method takes one parameter which can be the table name (string), an array of two strings (table name and alias), or an object (See Subqueries in the Advanced Queries section below).

```
$query = DB::select()->from('users');
```

Limiting the results of queries is done using the `where()`, `and_where()` and `or_where()` methods. These methods take three parameters: a column, an operator, and a value.

```
$query = DB::select()->from('users')->where('username', '=', 'john');
```

Multiple `where()` methods may be used to string together multiple clauses connected by the boolean operator in the method's prefix. The `where()` method is a wrapper that just calls `and_where()`.

```
$query = DB::select()->from('users')->where('username', '=', 'john')->or_where('username', '=',
```

You can use any operator you want. Examples include `IN`, `BETWEEN`, `>`, `=<`, `!=`, etc. Use an array for operators that require more than one value.

```
$query = DB::select()->from('users')->where('logins', '<=', 1);
```

```
$query = DB::select()->from('users')->where('logins', '>', 50);

$query = DB::select()->from('users')->where('username', 'IN', array('john','mark','matt'));

$query = DB::select()->from('users')->where('joindate', 'BETWEEN', array($then, $now));
```

By default, DB::select will select all columns (`SELECT * ...`), but you can also specify which columns you want returned by passing parameters to DB::select:

```
$query = DB::select('username', 'password')->from('users')->where('username', '=', 'john');
```

Now take a minute to look at what this method chain is doing. First, we create a new selection object using the DB::select method. Next, we set table(s) using the `from()` method. Last, we search for a specific records using the `where()` method. We can display the SQL that will be executed by casting the query to a string:

```
echo Debug::vars((string) $query);
// Should display:
// SELECT `username`, `password` FROM `users` WHERE `username` = 'john'
```

Notice how the column and table names are automatically escaped, as well as the values? This is one of the key benefits of using the query builder.

Select - AS (column aliases)

It is also possible to create `AS` aliases when selecting, by passing an array as each parameter to DB::select:

```
$query = DB::select(array('username', 'u'), array('password', 'p'))->from('users');
```

This query would generate the following SQL:

```
SELECT `username` AS `u`, `password` AS `p` FROM `users`
```

Select - DISTINCT

Unique column values may be turned on or off (default) by passing TRUE or FALSE, respectively, to the `distinct()` method.

```
$query = DB::select('username')->distinct(TRUE)->from('posts');
```

This query would generate the following SQL:

```
SELECT DISTINCT `username` FROM `posts`
```

Select - LIMIT & OFFSET

When querying large sets of data, it is often better to limit the results and page through the data one chunk at a time. This is done using the `limit()` and `offset()` methods.

```
$query = DB::select()->from(`posts`)->limit(10)->offset(30);
```

This query would generate the following SQL:

```
SELECT * FROM `posts` LIMIT 10 OFFSET 30
```

Select - ORDER BY

Often you will want the results in a particular order and rather than sorting the results, it's better to have the results returned to you in the correct order. You can do this by using the order_by() method. It takes the column name and an optional direction string as the parameters. Multiple `order_by()` methods can be used to add additional sorting capability.

```
$query = DB::select()->from(`posts`)->order_by(`published`, `DESC`);
```

This query would generate the following SQL:

```
SELECT * FROM `posts` ORDER BY `published` DESC
```

For a complete list of methods available while building a select query see Database_Query_Builder_Select.

Insert

To create records into the database, use DB::insert to create an INSERT query, using `values()` to pass in the data:

```
$query = DB::insert('users', array('username', 'password'))->values(array('fred', 'p@5sW0Rd'));
```

This query would generate the following SQL:

```
INSERT INTO `users` (`username`, `password`) VALUES ('fred', 'p@5sW0Rd')
```

For a complete list of methods available while building an insert query see Database_Query_Builder_Insert.

Update

To modify an existing record, use DB::update to create an UPDATE query:

```
$query = DB::update('users')->set(array('username' => 'jane'))->where('username', '=', 'john');
```

This query would generate the following SQL:

```
UPDATE `users` SET `username` = 'jane' WHERE `username` = 'john'
```

For a complete list of methods available while building an update query see Database_Query_Builder_Update.

Delete

To remove an existing record, use DB::delete to create a DELETE query:

```
$query = DB::delete('users')->where('username', 'IN', array('john', 'jane'));
```

This query would generate the following SQL:

```
DELETE FROM `users` WHERE `username` IN ('john', 'jane')
```

For a complete list of methods available while building a delete query see Database_Query_Builder_Delete.

Advanced Queries

Joins

Multiple tables can be joined using the `join()` and `on()` methods. The `join()` method takes two parameters. The first is either a table name, an array containing the table and alias, or an object (subquery or expression). The second parameter is the join type: LEFT, RIGHT, INNER, etc.

The `on()` method sets the conditions for the previous `join()` method and is very similar to the `where()` method in that it takes three parameters; left column (name or object), an operator, and the right column (name or object). Multiple `on()` methods may be used to supply multiple conditions and they will be appended with an 'AND' operator.

```
// This query will find all the posts related to "smith" with JOIN
$query = DB::select('authors.name', 'posts.content')->from('authors')->join('posts')->on('autho
```

This query would generate the following SQL:

```
SELECT `authors`.`name`, `posts`.`content` FROM `authors` JOIN `posts` ON (`authors`.`id` = `po
```

If you want to do a LEFT, RIGHT or INNER JOIN you would do it like this `join('colum_name', 'type_of_join')`:

```
// This query will find all the posts related to "smith" with LEFT JOIN
$query = DB::select()->from('authors')->join('posts', 'LEFT')->on('authors.id', '=', 'posts.aut
```

This query would generate the following SQL:

```
SELECT `authors`.`name`, `posts`.`content` FROM `authors` LEFT JOIN `posts` ON (`authors`.`id`
```

When joining multiple tables with similar column names, it's best to prefix the columns with the table name or table alias to avoid errors. Ambiguous column names should also be aliased so that they can be referenced easier.

Database Functions

Eventually you will probably run into a situation where you need to call `COUNT` or some other database function within your query. The query builder supports these functions in two ways. The first is by using quotes within aliases:

```
$query = DB::select(array('COUNT("username")', 'total_users'))->from('users');
```

This looks almost exactly the same as a standard `AS` alias, but note how the column name is wrapped in double quotes. Any time a double-quoted value appears inside of a column name, **only** the part inside the double quotes will be escaped. This query would generate the following SQL:

```
SELECT COUNT(`username`) AS `total_users` FROM `users`
```

When building complex queries and you need to get a count of the total rows that will be returned, build the expression with an empty column list first. Then clone the query and add the COUNT function to one copy and the columns list to the other. This will cut down on the total lines of code and make updating the query easier.

```
$query = DB::select()->from('users')
    ->join('posts')->on('posts.username', '=', 'users.username')
    ->where('users.active', '=', TRUE)
    ->where('posts.created', '>=', $yesterday);

$total = clone $query;
$total->select(array('COUNT( DISTINCT "username")', 'unique_users'));
$query->select('posts.username')->distinct();
```

Aggregate Functions

Aggregate functions like `COUNT()`, `SUM()`, `AVG()`, etc. will most likely be used with the `group_by()` and possibly the `having()` methods in order to group and filter the results on a set of columns.

```
$query = DB::select('username', array('COUNT("id")', 'total_posts')
    ->from('posts')->group_by('username')->having('total_posts', '>=', 10);
```

This will generate the following query:

```
SELECT `username`, COUNT(`id`) AS `total_posts` FROM `posts` GROUP BY `username` HAVING `total_
```

Subqueries

Query Builder objects can be passed as parameters to many of the methods to create subqueries. Let's take the previous example query and pass it to a new query.

```
$sub = DB::select('username', array('COUNT("id")', 'total_posts')
    ->from('posts')->group_by('username')->having('total_posts', '>=', 10);

$query = DB::select('profiles.*', 'posts.total_posts')->from('profiles')
    ->join(array($sub, 'posts'), 'INNER')->on('profiles.username', '=', 'posts.username');
```

This will generate the following query:

```
SELECT `profiles`.*, `posts`.`total_posts` FROM `profiles` INNER JOIN
( SELECT `username`, COUNT(`id`) AS `total_posts` FROM `posts` GROUP BY `username` HAVING `tota
ON `profiles`.`username` = `posts`.`username`
```

Insert queries can also use a select query for the input values

```
$sub = DB::select('username', array('COUNT("id")', 'total_posts')
    ->from('posts')->group_by('username')->having('total_posts', '>=', 10);

$query = DB::insert('post_totals', array('username', 'posts'))->select($sub);
```

This will generate the following query:

```
INSERT INTO `post_totals` (`username`, `posts`)
SELECT `username`, COUNT(`id`) AS `total_posts` FROM `posts` GROUP BY `username` HAVING `total_
```

Boolean Operators and Nested Clauses

Multiple Where and Having clauses are added to the query with Boolean operators connecting each expression. The default operator for both methods is AND which is the same as the and_ prefixed method. The OR operator can be specified by prefixing the methods with or_. Where and Having clauses can be nested or grouped by post fixing either method with _open and then followed by a method with a _close.

```
$query = DB::select()->from('users')
    ->where_open()
        ->or_where('id', 'IN', $expired)
        ->and_where_open()
            ->where('last_login', '<=', $last_month)
            ->or_where('last_login', 'IS', NULL)
        ->and_where_close()
    ->where_close()
    ->and_where('removed','IS', NULL);
```

This will generate the following query:

```
SELECT * FROM `users` WHERE ( `id` IN (1, 2, 3, 5) OR ( `last_login` <= 1276020805 OR `last_log
```

Database Expressions

There are cases were you need a complex expression or other database functions, which you don't want the Query Builder to try and escape. In these cases, you will need to use a database expression created with DB::expr. **A database expression is taken as direct input and no escaping is performed.**

```
$query = DB::update('users')->set(array('login_count' => DB::expr('login_count + 1')))->where('
```

This will generate the following query, assuming `$id = 45`:

```
UPDATE `users` SET `login_count` = `login_count` + 1 WHERE `id` = 45
```

Another example to calculate the distance of two geographical points:

```
$query = DB::select(array(DB::expr('degrees(acos(sin(radians('.$lat.')) * sin(radians(`latitude
```

You must validate or escape any user input inside of DB::expr as it will obviously not be escaped it for you.

Executing

Once you are done building, you can execute the query using `execute()` and use the results.

```
$result = $query->execute();
```

To use a different database config group pass either the name or the config object to `execute()`.

```
$result = $query->execute('config_name')
```

Results

Execute

Once you have a query object built, either through a prepared statement or through the builder, you must then `execute()` the query and retrieve the results. Depending on the query type used, the results returned will vary.

Select

DB::select will return a Database_Result object which you can then iterate over. This example shows how you can iterate through the Database_Result using a foreach.

```
$results = DB::select()->from('users')->where('verified', '=', 0)->execute();
foreach($results as $user)
{
    // Send reminder email to $user['email']
    echo $user['email']." needs to verify his/her account\n";
}
```

Select - `as_object()` and `as_assoc()`

When iterating over a result set, the default type will be an associative array with the column names or aliases as the keys. As an option, before calling `execute()`, you can specify to return the result rows as an object by using the `as_object()` method. The `as_object()` method takes one parameter, the name of the class of your choice, but will default to TRUE which uses the `stdClass`. Here is the example again using `stdClass`.

```
$results = DB::select()->from('users')->where('verified', '=', 0)->as_object()->execute();
foreach($results as $user)
{
    // Send reminder email to $user->email
    echo $user->email." needs to verify his/her account\n";
}
```

The method `as_assoc()` will remove the object name and return the results set back to an associative array. Since this is the default, this method is seldom required.

Select - `as_array()`

Sometimes you will require the results as a pure array rather than as an object. The `Database_Result` method `as_array()` will return an array of all rows.

```
$results = DB::select('id', 'email')->from('users')->execute();
$users = $results->as_array();
foreach($users as $user)
{
    echo 'User ID: '.$user['id'];
    echo 'User Email: '.$user['email'];
}
```

It also accepts two parameters that can be very helpful: `$key` and `$value`. When passing a value to `$key` you will index the resulting array by the column specified.

```
$results = DB::select('id', 'email')->from('users')->execute();
$users = $results->as_array('id');
foreach($users as $id => $user)
{
    echo 'User ID: '.$id;
    echo 'User Email: '.$user['email'];
}
```

The second parameter, `$value`, will reference the column specified and return that value rather than the whole row. This is particularly useful when making `<select>` dropdowns.

```
$results = DB::select('id', 'name')->from('users')->execute();
$users = $results->as_array('id','name');
// Show a dropdown with all users in it.
echo Form::select('author', $users)
```

To return a non-associative array, leave `$key` as NULL and just pass a `$value`.

```
$results = DB::select('email')->from('users')->execute();
$users = $results->as_array(NULL, 'email');
foreach($users as $email)
{
    echo 'User Email: '.$email;
}
```

Select - `get()`

Sometime you only want a single value from a query. The `get()` method returns the value of the named column from the current row. The second parameter, `$default`, is used to supply a default value when the result is NULL.

```
$total_users = DB::select(array('COUNT("username")', 'total_users'))->from('users')->execute()-
```

Select - `cached()`

The mysql database driver returns a `Database_Result` that works with a MySQL Resource data type. Since this resource lives outside of PHP environment, it can't be serialized which means it also can't be cached. To get around this the `Database_Result` object has the `cached()` method that returns a `Database_Result_Cached` object of the result set. The `Database_Result_Cached` can be serialized and cached, but can take up more memory.

NOTE: Currently, the PDO diver always returns a class of `Database_Result_Cached`, so `cached()` just returns itself.

The `cached()` function doesn't actually do any caching, it simply returns the result in a way that can be serialized and cached. You will need to use the Cache Module or some other caching method.

Select - `count()`

The `Database_Result` object implements the `Countable` Interface. The method `count()` returns the total row count in the result set.

NOTE: This is the count of the current result set, not a count of how many records are in the database. This is important to point out especially when using `limit()` and `offset()` in your query.

For a complete list of methods available when working with a result set see Database_Result.

Insert

DB::insert returns an array of two values: the last insert id and the number of affected rows.

```
$insert = DB::insert('tools')
    ->columns(array('name', 'model', 'description'))
    ->values(array('Skil 3400 10" Table Saw', '3400', 'Powerful 15 amp motor; weighs just 54-po

list($insert_id, $affected_rows) = $insert->execute();
```

Update & Delete

DB::update and DB::delete both return the number of affected rows as an integer.

```
$rows_deleted = DB::delete('tools')->where('model', 'like', '3400')->execute();
```

Examples

Here are some "real world" examples of using the database library to construct your queries and use the results.

Examples of Prepared Statements

TODO: 4-6 examples of prepared statements of varying complexity, including a good bind() example.

Pagination and search/filter

In this example, we loop through an array of whitelisted input fields and for each allowed non-empty value we add it to the search query. We make a clone of the query and then execute that query to count the total number of results. The count is then passed to the Pagination class to determine the search offset. The last few lines search with Pagination's items_per_page and offset values to return a page of results based on the current page the user is on.

```
$query = DB::select()->from('users');
```

```
//only search for these fields
$form_inputs = array('first_name', 'last_name', 'email');
foreach ($form_inputs as $name)
{
    $value = Arr::get($_GET, $name, FALSE);
    if ($value !== FALSE AND $value != '')
    {
        $query->where($name, 'like', '%'.$value.'%');
    }
}

//copy the query & execute it
$pagination_query = clone $query;
$count = $pagination_query->select('COUNT("*") AS mycount')->execute()->get('mycount');

//pass the total item count to Pagination
$config = Kohana::$config->load('pagination');
$pagination = Pagination::factory(array(
    'total_items' => $count,
    'current_page'   => array('source' => 'route', 'key' => 'page'),
    'items_per_page' => 20,
    'view'           => 'pagination/pretty',
    'auto_hide'      => TRUE,
));
$page_links = $pagination->render();

//search for results starting at the offset calculated by the Pagination class
$query->order_by('last_name', 'asc')
    ->order_by('first_name', 'asc')
    ->limit($pagination->items_per_page)
    ->offset($pagination->offset);
$results = $query->execute()->as_array();
```

Having

TODO: example goes here

We could use more examples on this page.

ORM

Kohana 3.x includes a powerful Object Relational Mapping (ORM) module that uses the active record pattern and database introspection to determine a model's column information. ORM is integrated tightly with the Validation library.

The ORM allows for manipulation and control of data within a database as though it was a PHP object. Once you define the relationships ORM allows you to pull data from your database, manipulate the data in any way you like, and then save the result back to the database without the use of SQL. By creating relationships between models that follow convention over configuration, much of the repetition of writing queries to create, read, update, and delete information from the database can be reduced or entirely removed. All of the relationships can be handled automatically by the ORM library and you can access related data as standard object properties.

ORM is included with the Kohana 3.x install but needs to be enabled before you can use it. In your `application/bootstrap.php` file modify the call to Kohana::modules and include the ORM modules.

Getting started

Before we use ORM, we must enable the modules required

```
Kohana::modules(array(
    ...
    'database' => MODPATH.'database',
    'orm' => MODPATH.'orm',
    ...
));
```

The database module is requried for the ORM module to work. Of course the database module has to be configured to use an existing database.

You can now create your models and use ORM.

Creating your Model

To create a model for the table `members` in your database, create the file `application/classes/model/member.php` with the following syntax:

```
class Model_Member extends ORM
{
    ...
}
```

(this should provide more examples)

Overriding the Table name

If you wish to change the database table that a model uses, just override the `$_table_name` variable like this:

```
protected $_table_name = 'strange_tablename';
```

Changing the primary key

ORM assumes each model (and database table) has an `id` column that is indexed and unique. If your primary key column isn't named `id`, that's fine - just override the `$_primary_key` variable like this:

```
protected $_primary_key = 'strange_pkey';
```

Use a non-default database

For each model, you can define which database configuration ORM will run queries on. If you override the `$_db_group` variable in your model, ORM will connect to that database. Example:

```
protected $_db_group = 'alternate';
```

Basic Usage

Load a new model instance

To create a new `Model_User` instance, you can do one of two things:

```
$user = ORM::factory('user');
// Or
$user = new Model_User();
```

Inserting

To insert a new record into the database, create a new instance of the model:

```
$user = ORM::factory('user');
```

Then, assign values for each of the properties;

```
$user->first_name = 'Trent';
$user->last_name = 'Reznor';
$user->city = 'Mercer';
$user->state = 'PA';
```

Insert the new record into the database by running ORM::save:

```
$user->save();
```

ORM::save checks to see if a value is set for the primary key (`id` by default). If the primary key is set, then ORM will execute an `UPDATE` otherwise it will execute an `INSERT`.

Finding an object

To find an object you can call the ORM::find method or pass the id into the ORM constructor:

```
// Find user with ID 20
$user = ORM::factory('user')
    ->where('id', '=', 20)
    ->find();
// Or
$user = ORM::factory('user', 20);
```

Check that ORM loaded a record

Use the ORM::loaded method to check that ORM successfully loaded a record.

```
if ($user->loaded())
{
    // Load was successful
}
else
{
    // Error
}
```

Updating and Saving

Once an ORM model has been loaded, you can modify a model's properties like this:

```
$user->first_name = "Trent";
$user->last_name = "Reznor";
```

And if you want to save the changes you just made back to the database, just run a `save()` call like this:

```
$user->save();
```

Deleting

To delete an object, you can call the ORM::delete method on a loaded ORM model.

```
$user = ORM::factory('user', 20);
$user->delete();
```

Relationships

Kohana ORM supports four types of object relationships: `belongs_to`, `has_many`, `has_many "through"` and `has_one`. The `has_many "through"` relationship can be used to function like Active Record's `has_many_and_belongs_to` relationship type.

belongs_to

A `belongs_to` relation should be used when you have one model that belongs to another. For example, a `Child` model belongs_to a `Parent` or a `Flag` model `belongs_to` a `Country`.

This is the base `belongs_to` relationship:

```
protected $_belongs_to = array(
    '[alias name]' => array(
```

```
        'model'       => '[model name]',
        'foreign_key' => '[column]',
    ),
);
```

You can omit any or all of the keys/values in the array on the right, in which case defaults are used:

```
protected $_belongs_to = array('[alias name]' => array());
```

The **alias name** is what is used to access the related model in your code. If you had a `Post` model that belonged to a `User` model and wished to use the default values of the `belongs_to` configuration then your code would look like this:

```
protected $_belongs_to = array('user' => array());
```

To access the user model, you would use `$post->user`. Since we're using the defaults above, the alias name will be used for the model name, and the foreign key in the posts table will be the alias name followed by `_id`, in this case it would be `user_id`. (You can change the `_id` suffix by modifying the `$foreign_key_suffix` variable in the model.)

Let's say your `Post` database table schema doesn't have a `user_id` column but instead has an `author_id` column which is a foreign key for a record in the `User` table. You could use code like this:

```
protected $_belongs_to = array(
    'user' => array(
        'foreign_key' => 'author_id',
    ),
);
```

If you wanted access a post's author by using code like `$post->author` then you would simply need to change the alias and add the `model` index:

```
protected $_belongs_to = array(
    'author' => array(
        'model'       => 'user',
    ),
);
```

has_many

The standard `has_many` relationship will likely fall on the other side of a `belongs_to` relationship. In the above examples, a post belongs to a user. From the user's perspective, a user has many posts. A has_many relationship is defined below:

```
protected $_has_many = array(
    '[alias name]' => array(
        'model'       => '[model name]',
        'foreign_key' => '[column]',
    ),
);
```

Again, you can omit all keys in the right array to use the defaults:

```
protected $_has_many = array('[alias name]' => array());
```

For our user and post example, this would look like the following in the user model:

```
protected $_has_many = array('posts' => array());
```

Using the above, the posts could be access using `$user->posts->find_all()`. Notice the `find_all()` used in this example. With `belongs_to` and `has_one` relationship types, the model is already loaded with necessary data. For `has_many` relationships, however, you may want to limit the number of results or add additional conditions to the SQL query; you can do so prior to the `find_all()`.

The model name used by default will be the singular name of the alias using the `inflector` class. In this case, `posts` uses `post` as the model name. The foreign key used by default is the owner model's name followed by `_id`. In this case, the foreign key will be `user_id` and it must exist in the posts table as before.

Let's assume now you want to access the posts using the name `stories` instead, and are still using the `author_id` key as in the `belongs_to` example. You would define your has_many relationship as:

```
protected $_has_many = array(
    'stories' => array(
        'model'       => 'post',
        'foreign_key' => 'author_id',
    ),
);
```

has_one

A `has_one` relationship is almost identical to a `has_many` relationship. In a `has_one` relationship, there can be 1 and only 1 relationship (rather than 1 or more in a has_many). If a user can only have one post or story, rather than many then the code would look like this:

```
protected $_has_one = array(
    'story' => array(
        'model'       => 'post',
        'foreign_key' => 'author_id',
    ),
);
```

has_many "through"

A `has_many "through"` relationship is used for many-to-many relationships. For instance, let's assume now we have an additional model, called `Category`. Posts may belong to more than one category, and each category may have more than one post. To link them together, an additional table is needed with columns for a `post_id` and a `category_id` (sometimes called a pivot table). We'll name the model for this `Post_Category` and the corresponding table `categories_posts`.

To define the `has_many` "through" relationship, the same syntax for standard has_many relationships is used with the addition of a 'through' parameter. Let's assume we're working with the Post model:

```
protected $_has_many = array(
    'categories' => array(
        'model'   => 'category',
        'through' => 'categories_posts',
    ),
);
```

In the Category model:

```
protected $_has_many = array(
    'posts' => array(
        'model'   => 'post',
        'through' => 'categories_posts',
    ),
);
```

To access the categories and posts, you simply use `$post->categories->find_all()` and `$category->posts->find_all()`

Methods are available to check for, add, and remove relationships for many-to-many relationships. Let's assume you have a $post model loaded, and a $category model loaded as well. You can check to see if the $post is related to this $category with the following call:

```
$post->has('categories', $category);
```

The first parameter is the alias name to use (in case your post model has more than one relationship to the category model) and the second is the model to check for a relationship with.

Assuming you want to add the relationship (by creating a new record in the categories_posts table), you would simply do:

```
$post->add('categories', $category);
```

To remove:

```
$post->remove('categories', $category);
```

Validation

ORM models are tightly integrated with the Validation library and the module comes with a very flexible ORM_Validation_Exception that helps you quickly handle validation errors from basic CRUD operations.

Defining Rules

Validation rules are defined in the `ORM::rules()` method. This method returns the array of rules to be added to the Validation object like so:

```
public function rules()
{
    return array(
        'username' => array(
            // Uses Valid::not_empty($value);
            array('not_empty'),
            // Calls Some_Class::some_method('param1', 'param2');
            array('Some_Class::some_method', array('param1', 'param2')),
            // Calls A_Class::a_method($value);
            array(array('A_Class', 'a_method')),
            // Calls the lambda function and passes the field value and the validation object
            array(function($value, Validation $object)
            {
                $object->error('some_field', 'some_error');
            }, array(':value', ':validation')),
        ),
    );
}
```

Bound Values

ORM will automatically bind the following values with `Validation::bind()`:

- **:field** - The name of the field the rule is being applied to.
- **:value** - The value of the field the rule is being applied to.
- **:model** - The instance of the model that is being validated.

Automatic Validation

All models automatically validate their own data when `ORM::save()`, `ORM::update()`, or `ORM::create()` is called. Because of this, you should always expect these methods to throw an ORM_Validation_Exception when the model's data is invalid.

```
public function action_create()
{
    try
    {
        $user = ORM::factory('user');
        $user->username = 'invalid username';
        $user->save();
    }
    catch (ORM_Validation_Exception $e)
    {
        $errors = $e->errors();
    }
}
```

Handling Validation Exceptions

The ORM_Validation_Exception will give you access to the validation errors that were encountered while trying to save a model's information. The `ORM_Validation_Exception::errors()` method works very similarly to `Validation::errors()`. Not passing a first parameter will return the name of the rules that failed. But unlike `Validate::errors()`, the first parameter of `ORM_Validation_Exception::errors()` is a directory path. The model's ORM::$_object_name will be appended to the directory in order to form the message file for `Validation::errors()` to use. The second parameter is identical to that of `Validation::errors()`.

In the below example, the error messages will be defined in `application/messages/models/user.php`

```
public function action_create()
{
    try
    {
        $user = ORM::factory('user');
        $user->username = 'invalid username';
        $user->save();
    }
    catch (ORM_Validation_Exception $e)
    {
        $errors = $e->errors('models');
    }
}
```

External Validation

Certain forms contain information that should not be validated by the model, but by the controller. Information such as a CSRF token, password verification, or a CAPTCHA should never be validated by a model. However, validating information in multiple places and combining the errors to provide the user with a good experience is often quite tedius. For this reason, the ORM_Validation_Exception is built to handle multiple Validation objects and namespaces the array of errors automatically for you. `ORM::save()`, `ORM::update()`, and `ORM::create()` all take an optional first parameter which is a Validation object to validate along with the model.

```
public function action_create()
{
    try
    {
        $user = ORM::factory('user');
        $user->username = $_POST['username'];
        $user->password = $_POST['password'];

        $extra_rules = Validation::factory($_POST)
            ->rule('password_confirm', 'matches', array(
                ':validation', ':field', 'password'
            ));

        // Pass the extra rules to be validated with the model
        $user->save($extra_rules);
    }
    catch (ORM_Validation_Exception $e)
    {
        $errors = $e->errors('models');
    }
}
```

Because the validation object was passed as a parameter to the model, any errors found in that check will be namespaced into a sub-array called `_external`. The array of errors would look something like this:

```
array(
    'username'  => 'This field cannot be empty.',
    '_external' => array(
        'password_confirm' => 'The values you entered in the password fields did not match.',
    ),
```

```
);
```

This ensures that errors from multiple validation objects and models will never overwrite each other.

The power of the ORM_Validation_Exception can be leveraged in many different ways to merge errors from related models. Take a look at the list of Examples for some great use cases.

Filters

Filters in ORM work much like they used to when they were part of the Validate class in 3.0.x however they have been modified to match the flexible syntax of Validation rules in 3.1.x. Filters run as soon as the field is set in your model and should be used to format the data before it is inserted into the Database.

Define your filters the same way you define rules, as an array returned by the `ORM::filters()` method like the following:

```
public function filters()
{
    return array(
        'username' => array(
            array('trim'),
        ),
        'password' => array(
            array(array($this, 'hash_password')),
        ),
        'created_on' => array(
            array('Format::date', array(':value', 'Y-m-d H:i:s')),
        ),
    );
}
```

When defining filters, you may use the parameters `:value`, `:field`, and `:model` to refer to the field value, field name, and the model instance respectively.

Examples

- Simple: Basic, one table model examples.
- Validation: Full example of creating a user account and handling validation errors.

@TODO:

The following is a sample list of examples that might be useful. Don't feel limited by this list, or consider these required. Items on the list can be combined, split up, removed or added to. All contribution are appreciated.

- Examples of changing things like $_table_name, $_labels, with, etc.
- Example of a one to one relationship.
- Example of one to many
- Example of many to many.

Validation Example

This example will create user accounts and demonstrate how to handle model and controller validation. We will create a form, process it, and display any errors to the user. We will be assuming that the Model_User class contains a method called `hash_password` that is used to turn the plaintext passwords into some kind of hash. The implementation of the hashing methods are beyond the scope of this example and should be provided with the Authentication library you decide to use.

SQL schema

```
CREATE TABLE IF NOT EXISTS `members` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL,
```

```
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Model

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

class Model_Member extends ORM {

    public function rules()
    {
        return array(
            'username' => array(
                array('not_empty'),
                array('min_length', array(':value', 4)),
                array('max_length', array(':value', 32)),
                array(array($this, 'username_available')),
            ),
            'password' => array(
                array('not_empty'),
            ),
        );
    }

    public function filters()
    {
        return array(
            'password' => array(
                array(array($this, 'hash_password')),
            ),
        );
    }

    public function username_available($username)
    {
        // There are simpler ways to do this, but I will use ORM for the sake of the example
        return ORM::factory('member', array('username' => $username))->loaded();
    }

    public function hash_password($password)
    {
        // Do something to hash the password
    }
}
```

HTML Form

Please forgive my slightly ugly form. I am trying not to use any modules or unrelated magic. :)

```
<form action="<?= URL::site('/members'); ?>" method="post" accept-charset="utf-8">
    <label for="username">Username:</label>
    <input id="username" type="text" name="username" value="<?= Arr::get($values, 'username');
    <label for="username" class="error"><?= Arr::get($errors, 'username'); ?>

    <label for="password">Password:</label>
    <input id="password" type="password" name="password" value="<?= Arr::get($values, 'password
    <label for="password" class="error"><?= Arr::get($errors, 'password'); ?>

    <label for="password_confirm">Repeat Password:</label>
    <input id="password_confirm" type="password" name="_external[password_confirm]" value="<?=
    <label for="password_confirm" class="error"><?= Arr::path($errors, '_external.password_conf

    <button type="submit">Create</button>
```

```
</form>
```

Controller

Remember that the `password` will be hashed as soon as it is set in the model, for this reason, it is impossible to validate it's length or the fact that it matches the `password_confirm` field. The model should not care about validating the `password_confirm` field, so we add that logic to the controller and simply ask the model to bundle the errors into one tidy array. Read the filters section to understand how those work.

```
public function action_create()
{
    $view = View::factory('members/create')
        ->set('values', $_POST)
        ->bind('errors', $errors);

    if ($_POST)
    {
        $member = ORM::factory('member')
            // The ORM::values() method is a shortcut to assign many values at once
            ->values($_POST, array('username', 'password'));

        $external_values = array(
            // The unhashed password is needed for comparing to the password_confirm field
            'password' => Arr::get($_POST, 'password'),
        // Add all external values
        ) + Arr::get($_POST, '_external', array());
        $extra = Validation::factory($external_values)
            ->rule('password_confirm', 'matches', array(':validation', ':field', 'password'));

        try
        {
            $member->save($extra);
            // Redirect the user to his page
            $this->request->redirect('members/'.$member->id);
        }
        catch (ORM_Validation_Exception $e)
        {
            $errors = $e->errors('models');
        }
    }

    $this->response->body($view);
}
```

Messages

application/messages/models/member.php

```
return array(
    'username' => array(
        'not_empty' => 'You must provide a username.',
        'min_length' => 'The username must be at least :param2 characters long.',
        'max_length' => 'The username must be less than :param2 characters long.',
        'username_available' => 'This username is not available.',
    ),
    'password' => array(
        'not_empty' => 'You must provide a password.',
    ),
);
```

application/messages/models/member/_external.php

```
return array(
    'password_confirm' => array(
```

```
        'matches' => 'The password fields did not match.',
    ),
);
```

Upgrading

Table aliases

ORM will now alias the main table in a query to the model's singular object name. i.e. Prior to 3.2 ORM set the from table like so:

```
$this->_db_builder->from($this->_table_name);
```

As of 3.2 it is now aliased like so:

```
$this->_db_builder->from(array($this->_table_name, $this->_object_name));
```

If you have a model `Model_Order` then when building a query use the alias like so:

```
$model->where('order.id', '=', $id);
```

Userguide

The userguide module provides documentation viewing including browsing the source code comments.

Give tips on how to most effectively use the user guide. How to navigate around, how things are organized, etc.

How the Userguide works

The userguide uses Markdown for the documentation. Both the userguide pages, and the in code comments for the API browser are written in markdown.

Userguide pages

Userguide pages are in the module they apply to, in `guide/<module>`. For example, documentation for Kohana is in `system/guide/kohana` and documentation for orm is in `modules/orm/guide/orm`, database is in `modules/database/guide/database`, etc.

Each module has an index page at `guide/<module>/index.md`.

Each module's menu is at `guide/<module>/menu.md`.

All other pages are are in `guide/<module>` and can be organized in subfolders and named however you want.

For more info on how to make your module have userguide pages, see Adding your module.

Images

Any images used in the userguide pages must be in `media/guide/<module>/`. For example, if a page has `![Image Title](hello-world.jpg)` the image would be located at `media/guide/<module>/hello-world.jpg`. Images for the ORM module are in `modules/orm/media/guide/orm`, and images for the Kohana docs are in `system/media/guide/kohana`.

API browser

The API browser is generated from the actual source code. The descriptions for classes, constants, properties, and methods is extracted from the comments and parsed in Markdown. For example if you look in the comment for Kohana_Core::init you can see a markdown list and table. These are parsed and show correctly in the API browser. `@param`, `@uses`, `@throws`, `@returns` and other tags are parsed as well.

TODO: give more specific details on how to comment your classes, constants, methods, etc. including package and how it relates to the api module.

Contributing

Kohana is community driven, and we rely on community contributions for the documentation.

Guidelines

Documentation should use complete sentences, good grammar, and be as clear as possible. Use lots of example code, but make sure the examples follow the Kohana conventions and style.

Try to commit often, with each commit only changing a file or two, rather than changing a ton of files and commiting it all at once. This will make it easier to offer feedback and merge your changes. Make sure your commit messages are clear and descriptive. Bad: "Added docs", Good: "Added initial draft of hello world tutorial", Bad: "Fixed typos", Good: "Fixed typos on the query builder page"

If you feel a menu needs to be rearranged or a module needs new pages, please open a bug report to discuss it.

Quick Method

To quickly point out something that needs improvement, report a bug report.

If you want to contribute some changes, you can do so right from your browser without even knowing git!

First create an account on Github.

You will need to fork the module for the area you want to improve. For example, to improve the ORM documentation fork http://github.com/kohana/orm. To improve the Kohana documentation, fork http://github.com/kohana/core, etc. So, find the module you want to improve and click on the Fork button in the top right.

The files that make the User Guide portion are found in `guide/<module>/`, and the API browser portion is made from the comments in the source code itself. Navigate to one of the files you want to change and click the edit button in the top right of the file viewer.

Make the changes and add a **detailed commit message**. Repeat this for as many files as you want to improve. (Note that you can't preview what the changes will look unless you actually test it locally.)

After you have made your changes, send a pull request so your improvements can be reviewed to be merged into the official documentation.

Once your pull request has been accepted, you can delete your repository if you want. Your commit will have been copied to the official branch.

If you know git

Short version: Fork the module whose docs you wish to improve (e.g. `git://github.com/kohana/orm.git` or `git://github.com/kohana/core.git`), checkout the `3.2/develop` branch (for the 3.2 docs!), make changes, and then send a pull request.

Long version: (This still assumes you at least know your way around git, especially how submodules work.)

1. Fork the specific repo you want to contribute to on github. (For example go to http://github.com/kohana/core and click the fork button.)
2. Now you need to add your fork as a "git remote" to your application and ensure you are on the right branch. An example for the ORM module and 3.2 docs:

```
cd my-kohana-app/modules/orm

# add your repository as a new remote
git remote add <your name> git://github.com/<your name>/orm.git
```

```
# Get the correct branch
git checkout 3.2/develop
```

3. Now go into the repo of the area of docs you want to contribute to and add your forked repo as a new remote, and push to it.

```
cd my-kohana-app/modules/orm

# Make some changes to the docs
nano file.md

# Commit your changes - Use a descriptive commit message! If there is a redmine ticket for
git commit -a -m "Corrected a typo in the ORM docs. Fixes #12345."

# make sure we are up to date with the latest changes
git merge origin/3.2/develop

# Now push your changes to your fork.
git push <your name> 3.2/develop
```

4. Finally, Send a pull request on github.

Configuration

The userguide has the following config options, available in `config/userguide.php`.

```
return array
(
    // Enable the API browser.  TRUE or FALSE
    'api_browser'  => TRUE,

    // Enable these packages in the API browser.  TRUE for all packages, or a string of comma s
    // Example: 'api_packages' => 'Kohana,Kohana/Database,Kohana/ORM,None',
    'api_packages' => TRUE,

);
```

You can enable or disabled the entire api browser, or limit it to only show certain packages. To disable a module from showing pages in the userguide, simply change that modules `enabled` option using the cascading filesystem. For example:

```
`application/config/userguide.php`

return array
(
    'modules' => array
    (
        'kohana' => array
        (
            'enabled' => FALSE,
        ),
        'database' => array
        (
            'enabled' => FALSE,
        )
    )
)
```

Using this you could make the userguide only show your modules and your classes in the api browser, if you wanted to host your own documentation on your site. Feel free to change the styles and views as well, but be sure to give credit where credit is due!

Adding your module to the userguide

Making your module work with the userguide is simple.

First, copy this config and place in it `<module>/config/userguide.php`, replacing anything in `<>` with the appropriate things:

```
return array
(
    // Leave this alone
    'modules' => array(

        // This should be the path to this modules userguide pages, without the 'guide/'. Ex: '
        '<modulename>' => array(

            // Whether this modules userguide pages should be shown
            'enabled' => TRUE,

            // The name that should show up on the userguide index page
            'name' => '<Module Name>',

            // A short description of this module, shown on the index page
            'description' => '<Description goes here.>',

            // Copyright message, shown in the footer for this module
            'copyright' => '&copy; 2010â€“2011 <Your Name>',
        )
    )
);
```

Next, create a folder in your module directory called `guide/<modulename>` and create `index.md` and `menu.md`. All userguide pages use Markdown. The index page is what is shown on the index of your module, the menu is what shows up in the side column. The menu should be formatted like this:

```
## [Module Name]()
 - [Page name](page-path)
 - [This is a Category](category)
     - [Sub Page](category/sub-page)
     - [Another](category/another)
         - [Sub sub page](category/another/sub-page)
 - Categories do not have to be a link to a page
     - [Etcetera](etc)
```

Page paths are relative to `guide/<modulename>`. So `[Page name](path-path)` would look for `guide/<modulename>/page-name.md` and `[Another](category/another)` would look for `guide/<modulename>/page-name.md`. The guide pages can be named or arranged any way you want within that folder (with the exception of `menu.md` and `index.md`). The breadcrumbs and page titles are pulled from the `menu.md file`, not the file names or paths. You can have items that are not pages (a category that doesn't have a corresponding page). To link to the `index.md` page, you should have an empty link, e.g. `[Module Name]()`. Do not include `.md` in your links.

Window size: 1044 x 774
Viewport size: 1044 x 635