CakePHP: The Manual

Preface

Section 1

Audience

This manual is written for people who want to build web applications faster and more enjoyably. CakePHP aims to assist PHP users of all levels to create robust, maintainable applications quickly and easily.

This manual expects a basic knowledge of PHP and HTML. A familiarity with the Model-View-Controller programming pattern is helpful, but we will cover that along the way for those new to MVC. While this work will attempt to aid the reader in configuring and troubleshooting their web server, a full coverage of such issues is outside the scope of this manual.

Rudimentary knowledge in first aid (CPR), basic survival skills, dating & relationships is also generally recommended, though also outside the scope of this document.

Section 2

CakePHP is Free

CakePHP is free. You don't have to pay for it, you can use it any way you want. CakePHP is developed under the MIT License. CakePHP is an Open Source project, this means you have full access to the source code. The best place to get the most recent version is at the CakePHP web site (http://www.cakephp.org). You can also browse the latest and greatest code there.

Section 3

Community

CakePHP is developed by a hard working community of people. They come from different countries all over the world and joined together to create the CakePHP framework for the benefit of the widest audience possible. For more information about Cake's active developer and user communities, visit http://www.cakephp.org.

Our IRC channel is always filled with knowledgable, friendly Bakers. If you're stuck on a bit of code, need a listening ear, or want to start an argument about coding conventions, drop on by: we'd love to hear from you. Visit us at #cakephp on irc.freenode.com.

CakePHP: The Manual

Introduction to CakePHP

Section 1

What is CakePHP?

CakePHP is a free open-source rapid development framework for PHP. Its a structure of libraries, classes and run-time infrastructure for programmers creating web applications originally inspired by the Ruby on Rails framework. Our primary goal is to enable you to work in a structured and rapid manner - without loss of flexibility.

Section 2

Why CakePHP?

CakePHP has several features that make it a great choice as a framework for developing applications swiftly and with the least amount of hassle. Here are a few in no particular order:

1. Active, friendly community
2. Flexible Licensing
3. Compatibility with PHP4 and PHP5
4. Integrated CRUD for database interaction and simplified queries
5. Application Scaffolding
6. Model View Controller (MVC) Architecture
7. Request dispatcher with good looking, custom URLs
8. Built-in Validation
9. Fast and flexible templating (PHP syntax, with helpers)
10. View Helpers for AJAX, Javascript, HTML Forms and more
11. Security, Session, and Request Handling Components
12. Flexible access control lists
13. Data Sanitization
14. Flexible View Caching
15. Works from any web site subdirectory, with little to no Apache configuration involved

Section 3

History of CakePHP

In 2005, Michal Tatarynowicz wrote a minimal version of a Rapid Application Framework in PHP. He found that it was the start of a very good framework. Michal published the framework under the MIT license, dubbing it Cake, and opened it up to a community of developers, who now maintain Cake under the name CakePHP.

CakePHP: The Manual

Basic Concepts

Section 1

Introduction

This chapter is a short, casual introduction to MVC concepts as they are implemented in Cake. If you're new to MVC (Model View Controller) patterns, this chapter is definitely for you. We begin with a discussion of general MVC concepts, work our way into the specific application of MVC in CakePHP, and show some simple examples of CakePHP using the MVC pattern.

Section 2

The MVC Pattern

Model-View-Controller is a software design pattern that helps you logically separate your code, make it more reusable, maintainable, and generally better. Model View Controller was first described by the author group Gang of Four. Dean Helman wrote (an extract from Objective Toolkit Pro white paper):

> "The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. MVC was originally developed to map the traditional input, processing, output roles into the GUI realm.
>
> Input -> Processing -> Output
>
> Controller -> Model -> View
>
> "The user input, the modeling of the external world, and the visual feedback to the user are separated and handled by model, view port and controller objects. The controller interprets mouse and keyboard inputs from the user and maps these user actions into commands that are sent to the model and/or view port to effect the appropriate change. The model manages one or more data elements, responds to queries about its state, and responds to instructions to change state. The view port manages a rectangular area of the display and is responsible for presenting data to the user through a combination of graphics and text."

In Cake terms, the Model represents a particular database table/record, and it's relationships to other tables and records. Models also contain data validation rules, which are applied when model data is inserted or updated. The View represents Cake's view files, which are regular HTML files embedded with PHP code. Cake's Controller handles requests from the server. It takes user input (URL and POST data), applies business logic, uses Models to read and write data to and from databases and other sources, and lastly, sends output data to the appropriate view file.

To make it as easy as possible to organize your application, Cake uses this pattern not only to manage how objects interact within your application, but also how files are stored, which is detailed next.

Section 3

Overview of the Cake File Layout

When you unpack Cake on your server you will find three main folders -

```
app
cake
vendors
```

The **cake** folder is where the core libraries for Cake lay and you generally won't ever need to touch it.

The **app** folder is where your application specific folders and files will go. The separation between the **cake** folder and the **app**
folder make it possible for you to have many app folders sharing a single set of Cake libraries. This also makes it easy to update CakePHP: you just download the latest version of Cake and overwrite your current core libraries. No need to worry about overwriting something you wrote for your app.

You can use the **vendors**
directory to keep third-party libraries in. You will learn more about vendors later, but the basic idea is that you can access classes you've placed in the vendors directory using Cake's **vendor()** function.

Let's look at the entire file layout:

```
/app
    /config          - Contains config files for your database, ACL, etc.

    /controllers     - Controllers go here
        /components  - Components go here

    /index.php       - Allows you to deploy cake with /app as the DocumentRoot

    /models          - Models go here

    /plugins         - Plugins go here

    /tmp             - Used for caches and logs

    /vendors         - Contains third-party libaries for this application

    /views           - Views go here
        /elements    - Elements, little bits of views, go here
        /errors      - Your custom error pages go here
        /helpers     - Helpers go here
        /layouts     - Application layout files go here
        /pages       - Static views go here

    /webroot         - The DocumentRoot for the application
        /css
        /files
        /img
        /js

/cake                - Cake's core libraries. Don't edit any files here.

index.php

/vendors             - Used for server-wide third-party libraries.

VERSION.txt          - Let's you know what version of Cake you're using.
```

CakePHP: The Manual

Installing CakePHP

Section 1

Introduction

So now you know everything there is to know about the structure and purpose of all the CakePHP libraries, or you have skipped to this part because you don't care about that stuff and just want to start playing. Either way, you're ready to get your hands dirty.

This chapter will describe what must be installed on the server, different ways to configure your server, downloading and installing CakePHP, bringing up the default CakePHP page, and some troubleshooting tips just in case everything does not go as planned.

Section 2

Requirements

In order use CakePHP you must first have a server that has all the required libraries and programs to run CakePHP:

Server Requirements

Here are the requirements for setting up a server to run CakePHP:

1. An HTTP server (like Apache) with the following enabled: sessions, mod_rewrite (not absolutely necessary but preferred)
2. PHP 4.3.2 or greater. Yes, CakePHP works great in either PHP 4 or 5.
3. A database engine (right now, there is support for MySQL, PostgreSQL and a wrapper for ADODB).

Section 3

Installing CakePHP

Getting the most recent stable version

There are a few ways you can secure a copy of CakePHP: getting a stable release from CakeForge, grabbing a nightly build, or getting a fresh version of code from SVN.

To download a stable version of code, check out the files section of the CakePHP project at CakeForge by going to http://cakeforge.org/projects/cakephp/.

To grab a nightly, download one from
http://cakephp.org/downloads/index/nightly. These nightly releases are stable, and often include the bug fixes between stable releases.

To grab a fresh copy from our SVN repository, use your favorite SVN client and connect to https://svn.cakephp.org/repo/trunk/cake/ and choose the version you're after.

Unpacking

Now that you've downloaded the most recent release, place that compressed package on your web server in the webroot. Now you need to unpack the CakePHP package. There are two ways to do this, using a development setup, which allows you to easily view many CakePHP applications under a single domain, or using the production setup, which allows for a single CakePHP application on the domain.

Section 4

Setting Up CakePHP

The first way to setup CakePHP is generally only recommended for development environments because it is less secure. The second way is considered more secure and should be used in a production environment.

NOTE: **/app/tmp** must be writable by the user that your web server runs as.

Development Setup

For development we can place the whole Cake installation directory inside the specified DocumentRoot like this:

```
/wwwroot
    /cake
        /app
        /cake
        /vendors
        .htaccess
        index.php
```

In this setup the wwwroot folder acts as the web root so your URLs will look something like th is (if you're also using mod_rewrite):

```
www.example.com/cake/controllerName/actionName/param1/param2
```

Production Setup

In order to utilize a production setup, you will need to have the rights to change the Documen tRoot on your server. Doing so, makes the whole domain act as a single CakePHP application.

The production setup uses the following layout:

```
../path_to_cake_install
    /app
        /config
        /controllers
        /models
        /plugins
        /tmp
        /vendors
        /views
        /webroot <-- This should be your new DocumentRoot
        .htaccess
        index.php
    /cake
    /vendors
    .htaccess
    index.php
```

Suggested Production httpd.conf

```
DocumentRoot /path_to_cake/app/webroot
```

In this setup the webroot directory is acting as the web root so your URLs might look like thi s (if you're using mod_rewrite):

```
http://www.example.com/controllerName/actionName/param1/param2
```

Advanced Setup: Alternative Installation Options

There are some cases where you may wish to place Cake's directories on different places on dis k. This may be due to a shared host restriction, or maybe you just want a few of your apps to share the same Cake libraries.

There are three main parts to a Cake application:

1. The core CakePHP libraries - Found in **/cake**
2. Your application code (e.g. controllers, models, layouts and views) - Found in **/app**
3. Your application webroot files (e.g. images, javascript and css) - Found in **/app/webroot**

Each of these directories can be located anywhere on your file system, with the exception of t he webroot, which needs to be accessible by your web server. You can even move the **webroot** folder out of the **app** folder as long as you tell Cake where you've put it.

To configure your Cake installation, you'll need to make some changes to **/app/webroot/index.php** (as it is distributed in Cake). There are three constants that you'll need to edit: ROOT, APP_DIR, and CAKE_CORE_INCLUDE_PATH.

1. ROOT should be set to the path of the directory that contains your **app** folder.
2. APP_DIR should be set to the path of your **app** folder.
3. CAKE_CORE_INCLUDE_PATH should be set to the path of your Cake libraries folder.

/app/webroot/index.php (partial, comments removed)

```
if (!defined('ROOT'))
{
    define('ROOT', dirname(dirname(dirname(__FILE__))));
}

if (!defined('APP_DIR'))
{
    define ('APP_DIR', basename(dirname(dirname(__FILE__))));
}

if (!defined('CAKE_CORE_INCLUDE_PATH'))
{
    define('CAKE_CORE_INCLUDE_PATH', ROOT);
}
```

An example might help illustrate this better. Imagine that I wanted to set up Cake to work wit h the following setup:

1. I want my Cake libraries shared with other applications, and placed in **/usr/lib/cake**.
2. My Cake webroot directory needs to be **/var/www/mysite/**.
3. My application files will be stored in **/home/me/mysite**.

Here's what the file setup looks like:

```
/home
    /me
        /mysite                 <-- Used to be /cake_install/app
            /config
            /controllers
            /models
            /plugins
            /tmp
            /vendors
            /views
            index.php
/var
    /www
        /mysite                 <-- Used to be /cake_install/app/webroot
            /css
            /files
            /img
            /js
            .htaccess
            css.php
            favicon.ico
            index.php
/usr
    /lib
        /cake                   <-- Used to be /cake_install/cake
            /cake
                /config
                /docs
                /libs
                /scripts
                app_controller.php
                app_model.php
                basics.php
                bootstrap.php
                dispatcher.php
            /vendors
```

Given this type of setup, I would need to edit my webroot index.php file (which should be at / var/www/mysite/index.php, in this example) to look like the following:

It is recommended to use the 'DS' constant rather than slashes to delimit file paths. This pre vents any 'missing file' errors you might get as a result of using the wrong delimiter, and it makes your code more portable.

```
if (!defined('ROOT'))
{
    define('ROOT', DS.'home'.DS.'me');
}

if (!defined('APP_DIR'))
{
    define ('APP_DIR', 'mysite');
}

if (!defined('CAKE_CORE_INCLUDE_PATH'))
{
    define('CAKE_CORE_INCLUDE_PATH', DS.'usr'.DS.'lib'.DS.'cake');
}
```

Section 5

Configuring Apache and mod_rewrite

While CakePHP is built to work with mod_rewrite out of the box, we've noticed that a few users struggle with getting everything to play nicely on their systems. Here are a few things you might try to get it running correctly:

1. Make sure that an .htaccess override is allowed: in your httpd.conf, you should have a section that defines a section for each Directory on your server. Make sure the **AllowOverride** is set to **All** for the correct Directory.
2. Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.con f.
3. For some reason or another, you might have obtained a copy of CakePHP without the needed .htac cess files. This sometimes happens because some operating systems treat files that start with '.' as hidden, and don't copy them. Make sure yo ur copy of CakePHP is from the downloads section of the site or our SVN repository.
4. Make sure you are loading up mod_rewrite correctly! You should see something like **LoadModule rewrite_module libexec/httpd/mod_rewrite.so** and **AddModule mod_rewrite.c** in your httpd.conf.
5. If you are installing Cake into a user directory (http://example.com/~myusername/), you'll nee d to modify the .htaccess file in the base directory of your Cake installation. Just add the line "`RewriteBase /~myusername/`".
6. If for some reason your URLS are suffixed with a long, annoying session ID (http://example.com/posts/?CAKEPHP=4kgj577sgabvnmhjgkdiuy1956if6ska), you might also add "`php_flag session.trans_id off`" to the .htaccess file at the root of your installation as well.

Section 6

Make Sure It's Working

Alright, lets see this baby in action. Depending on which setup you used, you should point you r browser to http://www.example.com or http://www.example.com/cake. At this point, you'll be presented with CakePHP's default home, a nd a message that tells you the status of your current database connection.

Congratulations! You are ready to create your first Cake-based application.

CakePHP: The Manual

Configuration

Section 1

Database Configuration

Your **app/config/database.php**
file is where your database configuration all takes place. A fresh install doesn't have a **database.php**, so you'll need to make a copy of **database.php.default**. Once you've made a copy and renamed it you'll see the following:

app/config/database.php

```
var $default = array('driver'   => 'mysql',
                     'connect'  => 'mysql_connect',
                     'host'     => 'localhost',
                     'login'    => 'user',
                     'password' => 'password',
                     'database' => 'project_name',
                     'prefix'   => '');
```

Replace the information provided by default with the database connection information for your application.

One note about the prefix key: the string you enter there will be prepended to any SQL call that Cake makes to your database when working with tables. You define it here once so you don't have to specify it in other places. It also allows you to follow Cake's table naming conventions if you're on a host that only gives you a single database. Note: for HABTM join tables, you only add the prefix once: prefix_apples_bananas, not prefix_apples_prefix_bananas.

CakePHP supports the following database drivers:

1. mysql
2. postgres
3. sqlite
4. pear-**drivername** (so you might enter pear-mysql, for example)
5. adodb-**drivername**

The 'connect' key in the **$default**
connection allows you to specify whether or not the database connection will be treated as persistent or not. Read the comments in the database.php.default file for help on specifying connection types for your database setup.

Your database tables should also follow the following conventions:

1. Table names used by Cake should consist of English words in plural, like "users", "authors" or "articles". Note that corresponding models have singular names.
2. Your tables must have a primary key named 'id'.
3. If you plan to relate tables, use foreign keys that look like: 'article_id'. The table name is singular, followed by an underscore, followed by 'id'.

4. If you include a 'created' and/or 'modified' column in your table, Cake will automatically populate the field when appropriate.

You'll also notice that there is a $test connection setting included in the database.php file. Fill out this configuration (or add other similarly formatted configurations) and use it in your application by placing something like:

```
var $useDbConfig = 'test';
```

Inside one of your models. You can add any number of additional connection settings in this manner.

Section 2

Global Configuration

CakePHP's global configuration can be found in **app/config/core.php**. While we really dislike configuration files, it just had to be done. There are a few things you can change here, and the notes on each of these settings can be found within the comments of the **core.php** file.

DEBUG: Set this to different values to help you debug your application as you build it. Specifying this setting to a non-zero value will force Cake to print out the results of pr( ) and debug( ) function calls, and stop flash messages from forwarding automatically. Setting it to 2 or higher will result in SQL statements being printed at the bottom of the page.

Also when in debug mode (where DEBUG is set to 1 or higher), Cake will render certain generated error pages, i.e. "Missing Controller," "Missing Action," etc. In production mode, however (where DEBUG is set to 0), Cake renders the "Not Found" page, which can be overridden in **app/views/errors/error404.thtml**.

CAKE_SESSION_COOKIE: Change this value to the name of the cookie you'd like to use for user sessions in your Cake app.

CAKE_SECURITY: Change this value to indicate your preferred level of sessions checking. Cake will timeout sessions, generate new session ids, and delete old session files based on the settings you provide here. The possible values are:

1. high: sessions time out after 10 minutes of inactivity, and session id's are regenerated on each request
2. medium: sessions time out after 20 minutes of inactivity
3. low: sessions time out after 30 minutes of inactivity

CAKE_SESSION_SAVE: Specify how you'd like session data saved. Possible values are:

1. cake: Session data is saved in tmp/ inside your Cake installation
2. php: Session data saved as defined in php.ini
3. database: Session data saved to database connection defined by the 'default' key.

Section 3

Routes Configuration

"Routing" is a pared-down pure-PHP mod_rewrite-alike that can map URLs to controller/action/params and back. It was added to Cake to make pretty URLs more configurable and to divorce us from the mod_rewrite requirement. Using mod_rewrite, however, will make your address bar look much more tidy.

Routes are individual rules that map matching URLs to specific controllers and actions. Routes are configured in the **app/config/routes.php** file. They are set-up like this:

Route Pattern

```
<?php
$Route->connect (
    'URL',
    array('controller'=>'controllername',
    'action'=>'actionname', 'firstparam')
);
?>
```

Where:

1. **URL** is the regular expression Cake URL you wish to map,
2. **controllername** is the name of the controller you wish to invoke,
3. **actionname** is the name of the controller's action you wish to invoke,
4. and **firstparam** is the value of the first parameter of the action you've specified.

Any parameters following **firstparam** will also be passed as parameters to the controller action.

The following example joins all the urls in /blog to the BlogController. The default action will be BlogController::index().

Route Example

```
<?php
$Route->connect ('/blog/:action/*', array('controller'=>'Blog', 'action'=>'index'));
?>
```

A URL like /blog/history/05/june can then be handled like this:

Route Handling in a Controller

```
<?php
class BlogController extends AppController
{
 function history ($year, $month=null)
 {
   // .. Display appropriate content
 }
}
?>
```

The 'history' from the URL was matched by :action from the Blog's route. URL elements matched by * are passed to the active controller's handling method as parameters, hence the $year and $month. Called with URL /blog/history/05, history() would only be passed one parameter, 05.

The following example is a default CakePHP route used to set up a route for PagesController::display('home'). Home is a view which can be overridden by creating the file **/app/views/pages/home.thtml**.

Setting the Default Route

```
<?php
$Route->connect ('/', array('controller'=>'Pages', 'action'=>'display', 'home'));
?>
```

Section 4

Advanced Routing Configuration: Admin Routing and Webservices

There are some settings in /app/config/core.php you can take advantage of in order to organize your application and craft URLs that make the most sense to you and your users.

The first of these is admin routing. If your application has a ProductsController as well as a NewsController, you might want to set up some special URLs so users with administrative privileges can access special actions in those controllers. To keep the URLs nice and easy to read, some people prefer /admin/products/add and /admin/news/post to something like /products/adminAdd and /news/adminPost.

To enable this, first, uncomment the CAKE_ADMIN line in your /app/config/core.php file. The default value of CAKE_ADMIN is 'admin', but you can change it to whatever you like. Remember this string, because you'll need to prepend it to your administrative actions in your controller. So, admin actions in this case would be named admin_actionName(). Here's some examples of desired URLs and possible CAKE_ADMIN and controller action settings:

```
/admin/products/add          CAKE_ADMIN = 'admin'
                             name of action in ProductsController = 'admin_add()'

/superuser/news/post         CAKE_ADMIN = 'superuser'
                             name of action in NewsController = 'superuser_post()'

/admin/posts/delete          CAKE_ADMIN = 'admin'
                             name of action in PostsController = 'admin_delete()'
```

Using admin routes allows you to keep your logic organized while making the routing very easy to accomplish.

Please note that enabling admin routes or using them does not enable any sort of authentication or security. You'll need implement those yourself.

Similarly, you can enable Cake's webservices routing to make easier there as well. Have a controller action you'd like to expose as a webservice? First, set WEBSERVICES in /app/config/core.php to 'on'. This enables some automatic routing somewhat similar to admin routing, except that a certain set of route prefixes are enabled:

1. rss
2. xml
3. rest
4. soap
5. xmlrpc

What this does is allows you to provide an alternate views that will automatically be available at /rss/controllerName/actionName or /soap/controllerName/actionName. This allows you to create a single action that can have two views: one for normal HTML viewiers, and another for webservices users. By doing this, you can easily allow much of the functionality of your application to be available via webservices.

For example, let's say I have some logic in my application that tells users who is on the phone in my office. I already have a HTML view for this data, but I want to offer it in XML so it can be used in a desktop widget or handheld application. First I need to enable Cake's webservice routing:

/app/config/core.php (partial)

```
/**
 *  The define below is used to turn cake built webservices
 *  on or off. Default setting is off.
 */
```

```
    define('WEBSERVICES', 'on');
```

Next, I can structure the logic in my controller just as I normally would:

messages_controller.php

```
<?php
class PhonesController extends AppController
{
    function doWhosOnline()
    {
        // this action is where we do all the work of seeing who's on the phone...

        // If I wanted this action to be available via Cake's xml webservices route,
        // I'd need to include a view at /app/views/posts/xml/do_whos_online.thtml.
        // Note: the default view used here is at /app/views/layouts/xml/default.thtml.

        // If a user requests /phones/doWhosOnline, they will get an HTML version.
        // If a user requests /xml/phones/doWhosOnline, they will get the XML version.
    }
}
?>
```

Section 5

(Optional) Custom Inflections Configuration

Cake's naming conventions can be really nice - you can name your model Box, your controller Boxes, and everything just works out. There are occasions (especially for our non-english speaking friends) where you may run into situations where Cake's inflector (the class that pluralizes, singularizes, camelCases, and under_scores) might not work as you'd like. If Cake won't recognize your Foci or Fish, editing the custom inflections configuration file is where you'll need to go.

Found at /app/config/inflections.php is a list of Cake variables you can use to adjust the pluralization, singularization of classnames in Cake, along with defininig terms that shouldn't be inflected at all (like Fish and Deer, for you outdoorsman cakers) along with irregularities.

Follow the notes inside the file to make adjustments, or use the examples in the file by uncommenting them. You may need to know a little regex before diving in.

CakePHP: The Manual

Scaffolding

Section 1

Cake's Scaffolding is Pretty Cool

So cool that you'll want to use it in production apps. Now, we think its cool, too, but please realize that scaffolding is... well... just scaffolding. It's a bunch of stuff you throw up real quick during the beginning of a project in order to get started. It isn't meant to be completely flexible. So, if you find yourself really wanting to customize your logic and your views, its time to pull your scaffolding down in order to write some code.

Scaffolding is a great way of getting the early parts of developing a web application started. Early database schemas are volatile and subject to change, which is perfectly normal in the early part of the design process. This has a downside: a web developer hates creating forms that never will see real use. To reduce the strain on the developer, scaffolding has been included in Cake. Scaffolding analyzes your database tables and creates standard lists with add, delete and edit buttons, standard forms for editing and standard views for inspecting a single item in the database. To add scaffolding to your application, in the controller, add the **$scaffold** variable:

```
<?php
class CategoriesController extends AppController
{
    var $scaffold;
}
?>
```

One important thing to note about scaffold: it expects that any field name that ends with **_id** is a foreign key to a table which has a name that precedes the underscore. So, for example, if you have nested categories, you'd probably have a column called **parent_id**. With this release, it would be best to call this parentid. Also, when you have a foreign key in your table (e.g. titles table has **category_id**), and you have associated your models appropriately (see Understanding Associations, 6.2), a select box will be automatically populated with the rows from the foreign table (category) in the show/edit/new views. To set which field in the foreign table is shown, set the **$displayField** variable in the foreign model. To continue our example of a category having a title:

```
<?php
class Title extends AppModel
{
    var $name = 'Title';

    var $displayField = 'title';
}
?>
```

Section 2

Customizing Scaffold Views

If you're looking for something a little different in your scaffolded views, you can create them yourself. We still don't recommend using this technique for production applications, but such a customization may be extremely useful for prototyping iterations.

If you'd like to change your scaffolding views, you'll need to supply your own:

Custom Scaffolding Views for a Single Controller

```
Custom scaffolding views for a PostsController should be placed like so:

/app/views/posts/scaffold/index.scaffold.thtml
/app/views/posts/scaffold/show.scaffold.thtml
/app/views/posts/scaffold/edit.scaffold.thtml
/app/views/posts/scaffold/new.scaffold.thtml
```

Custom Scaffolding Views for an Entire Application

```
Custom scaffolding views for all controllers should be placed like so:

/app/views/scaffold/index.scaffold.thtml
/app/views/scaffold/show.scaffold.thtml
/app/views/scaffold/edit.scaffold.thtml
/app/views/scaffold/new.scaffold.thtml
```

If you find yourself wanting to change the controller logic at this point, it's time to take the scaffolding down from your application and start building it.

One feature you might find helpful is Cake's code generator: Bake. Bake allows you to generate a coded version of scaffolded code you can then move on to modify and customize as your application requires.

CakePHP: The Manual

Models

Section 1

What is a model?

What does it do? It separates domain logic from the presentation, isolating application logic.

A model is generally an access point to the database, and more specifically, to a certain table in the database. By default, each model uses the table who's name is plural of its own, i.e. a 'User' model uses the 'users' table. Models can also contain data validation rules, association information, and methods specific to the table it uses. Here's what a simple User model might look like in Cake:

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

```
<?php

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

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

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

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

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

    // You can also include your own functions:
    function makeInactive($uid)
    {
        //Put your own logic here...
    }
}

?>
```

Section 2

Model Functions

From a PHP view, models are classes extending the AppModel class. The AppModel class is originally defined in the cake/ directory, but should you want to create your own, place it in **app/app_model.php**. It should contain methods that are shared between two or more models. It itself extends the Model class which is a standard Cake library defined in **cake/libs/model.php**.

While this section will treat most of the often-used functions in Cake's Model, it's important to remember to use http://api.cakephp.org for a full reference.

User-Defined Functions

An example of a table-specific method in the model is a pair of methods for hiding/unhiding posts in a blog.

Example Model Functions

```
<?php
class Post extends AppModel
{
   var $name = 'Post';

   function hide ($id=null)
   {
      if ($id)
      {
          $this->id = $id;
          $this->saveField('hidden', '1');
      }
   }

   function unhide ($id=null)
   {
      if ($id)
      {
          $this->id = $id;
          $this->saveField('hidden', '0');
      }
   }
}
?>
```

Retrieving Your Data

Below are a few of the standard ways of getting to your data using a model:

- **findAll**
- string *$conditions*
- array *$fields*
- string *$order*
- int *$limit*
- int *$page*
- int *$recursive*

Returns the specified fields up to $limit records matching $conditions (if any), start listing from page $page (default is page 1). $conditions should look like they would in an SQL statement: $conditions = "race = 'wookie' AND thermal_detonators > 3", for example.

When the $recursive option is set to an integer value greater than 1, the findAll() operation will make an effort to return the models associated to the ones found by the findAll(). If your property has many owners who in turn have many contracts, a recursive findAll() on your Property model will return those associated models.

- **find**
- string *$conditions*
- array *$fields*
- string *$order*
- int *$recursive*

Returns the specified (or all if not specified) fields from the first record that matches $conditions.

When the $recursive option is set to an integer value between 1 and 3, the find() operation will make an effort to return the models associated to the ones found by the find(). The recursive find can go up to three levels deep. If your property has many owners who in turn have many contracts, a recursive find() on your Property model will return up to three levels deep of associated models.

- **findAllBy<fieldName>**
- string *$value*

These magic functions can be used as a shortcut to search your tables for a row given a certain field, and a certain value. Just tack on the name of the field you wish to search, and CamelCase it. Examples (as used in a Controller) might be:

```
$this->Post->findByTitle('My First Blog Post');
$this->Author->findByLastName('Rogers');
$this->Property->findAllByState('AZ');
$this->Specimen->findAllByKingdom('Animalia');
```

The returned result is an array formatted just as would be from find() or findAll().

- **findNeighbours**
- string *$conditions*
- array *$field*
- string *$value*

Returns an array with the neighboring models (with only the specified fields), specified by $field and $value, filtered by the SQL conditions, $conditions.

This is useful in situations where you want 'Previous' and 'Next' links that walk users through some ordered sequence through your model entries. It only works for numeric and date based fields.

```
class ImagesController extends AppController
{
    function view($id)
    {
        // Say we want to show the image...

        $this->set('image', $this->Image->find("id = $id");

        // But we also want the previous and next images...

        $this->set('neighbours', $this->Image->findNeighbours(null, 'id', $id);

    }
}
```

This gives us the full $image['Image'] array, along with $neighbours['prev']['Image']['id'] and $neighbours['next']['Image']['id'] in our view.

- **field**
- string *$name*
- string *$conditions*
- string *$order*

Returns as a string a single field from the first record matched by **$conditions** as ordered by **$order**.

- **findCount**
- string *$conditions*

Returns the number of records that match the given conditions.

- **generateList**
- string *$conditions*
- string *$order*
- int *$limit*
- string *$keyPath*
- string *$valuePath*

This function is a shortcut to getting a list of key value pairs - especially handy for creating a html select tag from a list of your models. Use the $conditions, $order, and $limit parameters just as you would for a findAll() request. The $keyPath and $valuePath are where you tell the model where to find the keys and values for your generated list. For example, if you wanted to generate a list of roles based on your Role model, keyed by their integer ids, the full call might look something like:

```
$this->set(
    'Roles',
    $this->Role->generateList(null, 'role_name ASC', null, '{n}.Role.id',
'{n}.Role.role_name')
);

//This would return something like:
array(
    '1' => 'Account Manager',
    '2' => 'Account Viewer',
    '3' => 'System Manager',
    '4' => 'Site Visitor'
);
```

- **read**
- string *$fields*
- string *$id*

Use this function to get the fields and their values from the currently loaded record, or the record specified by $id.

Please note that read() operations will only fetch the first level of associated models regardless of the value of $recursive in the model. To gain additional levels, use find() or findAll().

- **query**
- string *$query*

- **execute**
- string *$query*

Custom SQL calls can be made using the model's query() and execute() methods. The difference between the two is that query() is used to make custom SQL queries (the results of which are returned), and execute() is used to make custom SQL commands (which require no return value).

Custom Sql Calls with query()

```
<?php
class Post extends AppModel
{
    var $name = 'Post';

    function posterFirstName()
    {
        $ret = $this->query("SELECT first_name FROM posters_table
                                WHERE poster_id = 1");
        $firstName = $ret[0]['first_name'];
        return $firstName;
```

```
    }
}
?>
```

Complex Find Conditions (using arrays)

Most of the model's finder calls involve passing sets of conditions in one way or another. The simplest approach to this is to use a WHERE clause snippet of SQL, but if you need more control, you can use arrays. Using arrays is clearer and easier to read, and also makes it very easy to build queries. This syntax also breaks out the elements of your query (fields, values, operators, etc.) into discreet, manipulatable parts. This allows Cake to generate the most efficient query possible, ensure proper SQL syntax, and properly escape each individual part of the query.

At it's most basic, an array-based query looks like this:

Basic find conditions array usage example:

```
$conditions = array("Post.title" => "This is a post");

//Example usage with a model:
$this->Post->find($conditions);
```

The structure is fairly self-explanatory: it will find any post where the title matches the string "This is a post". Note that we could have used just "title" as the field name, but when building queries, it is good practice to always specify the model name, as it improves the clarity of the code, and helps prevent collisions in the future, should you choose to change your schema. What about other types of matches? These are equally simple. Let's say we wanted to find all the posts where the title is **not** "This is a post":

```
array("Post.title" => "<> This is a post")
```

All that was added was '<>' before the expression. Cake can parse out any valid SQL comparison operator, including match expressions using LIKE, BETWEEN, or REGEX, as long as you leave a space between the operator an the expression or value. The one exception here is IN (...)-style matches. Let's say you wanted to find posts where the title was in a given set of values:

```
array("Post.title" => array("First post", "Second post", "Third post"))
```

Adding additional filters to the conditions is as simple as adding additional key/value pairs to the array:

```
array
(
    "Post.title"   => array("First post", "Second post", "Third post"),
    "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
)
```

By default, Cake joins multiple conditions with boolean AND; which means, the snippet above would only match posts that have been created in the past two weeks, **and** have a title that matches one in the given set. However, we could just as easily find posts that match either condition:

```
array
("or" =>
    array
    (
        "Post.title" => array("First post", "Second post", "Third post"),
        "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
    )
)
```

Cake accepts all valid SQL boolean operations, including AND, OR, NOT, XOR, etc., and they can be upper or lower case, whichever you prefer. These conditions are also infinitely nestable. Let's say you had a

hasMany/belongsTo relationship between Posts and Authors, which would result in a LEFT JOIN on the find done on Post. Let's say you wanted to find all the posts that contained a certain keyword **or** were created in the past two weeks, but you want to restrict your search to posts written by Bob:

```
array
("Author.name" => "Bob", "or" => array
    (
        "Post.title" => "LIKE %magic%",
        "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks")
    )
)
```

Saving Your Data

To save data to your model, you need to supply it with the data you wish to save. The data handed to the save() method should be in the following form:

```
Array
(
    [ModelName] => Array
        (
            [fieldname1] => 'value'
            [fieldname2] => 'value'
        )
)
```

In order to get your data posted to the controller in this manner, it's easiest just to use the HTML Helper to do this, because it creates form elements that are named in the way Cake expects. You don't need to use it however: just make sure your form elements have names that look like **data[Modelname][fieldname]**. Using $html->input('Model/fieldname') is the easiest, however.

Data posted from forms is automatically formatted like this and placed in **$this->data** inside of your controller, so saving your data from web forms is a snap. An edit function for a property controller might look something like the following:

```
function edit($id)
{

   //Note: The property model is automatically loaded for us at $this->Property.

   // Check to see if we have form data...
   if (empty($this->data))
   {
        $this->Property->id = $id;
        $this->data = $this->Property->read();//populate the form fields with the current
row
   }
   else
   {
      // Here's where we try to save our data. Automagic validation checking
      if ($this->Property->save($this->data['Property']))
      {
         //Flash a message and redirect.
         $this->flash('Your information has been saved.',
                     '/properties/view/'.$this->data['Property']['id'], 2);
      }
      //if some fields are invalid or save fails the form will render
   }
}
```

Notice how the save operation is placed inside a conditional: when you try to save data to your model, Cake automatically attempts to validate your data using the rules you've provided. To learn more about data validation, see Chapter "Data Validation". If you do not want save() to try to validate your data, use **save($data,**

false).

Other useful save functions:

- **del**
- string *$id*
- boolean *$cascade*

Deletes the model specified by $id, or the current id of the model.

If this model is associated to other models, and the 'dependent' key has been set in the association array, this method will also delete those associated models if $cascade is set to true.

Returns true on success.

- **saveField**
- string *$name*
- string *$value*

Used to save a single field value.

- **getLastInsertId**

Returns the ID of the most recently created record.

Model Callbacks

We've added some model callbacks that allow you to sneak in logic before or after certain model operations. To gain this functionality in your applications, use the parameters provided and override these functions in your Cake models.

- **beforeFind**
- string *$conditions*

The beforeFind() callback is executed just before a find operation begins. Place any pre-find logic in this method. When you override this in your model, return **true** when you want the find to execute, and **false** when you want it to abort.

- **afterFind**
- array *$results*

Use this callback to modify results that have been returned from a find operation, or perform any other post-find logic. The parameter for this function is the returned results from the model's find operation, and the return value is the modified results.

- **beforeValidate**

Use this callback to modify model data before it is validated. It can also be used to add additional, more complex validation rules, using **Model::invalidate()**. In this context, model data is accessible via **$this->data**. This function must also return **true**, otherwise save() execution will abort.

- **beforeSave**

Place any pre-save logic in this function. This function executes immediately after model data has been validated (assuming it validates, otherwise the save() call aborts, and this callback will not execute), but before

the data is saved. This function should also return **true** if you want the save operation to continue, and **false** if you want to abort.

One usage of beforeSave might be to format time data for storage in a specifc database engine:

```
// Date/time fields created by HTML Helper:
// This code would be seen in a view

$html->dayOptionTag('Event/start');
$html->monthOptionTag('Event/start');
$html->yearOptionTag('Event/start');
$html->hourOptionTag('Event/start');
$html->minuteOptionTag('Event/start');

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/

// Model callback functions used to stitch date
// data together for storage
// This code would be seen in the Event model:

function beforeSave()
{
    $this->data['Event']['start'] = $this->_getDate('Event', 'start');

    return true;
}

function _getDate($model, $field)
{
    return date('Y-m-d H:i:s', mktime(
        intval($this->data[$model][$field . '_hour']),
        intval($this->data[$model][$field . '_min']),
        null,
        intval($this->data[$model][$field . '_month']),
        intval($this->data[$model][$field . '_day']),
        intval($this->data[$model][$field . '_year'])));
}
```

- **afterSave**

Place any logic that you want to be executed after every save in this callback method.

- **beforeDelete**

Place any pre-deletion logic in this function. This function should return **true** if you want the deletion to continue, and **false** if you want to abort.

- **afterDelete**

Place any logic that you want to be executed after every deletion in this callback method.

Section 3

Model Variables

When creating your models, there are a number of special variables you can set in order to gain access to Cake functionality:

$primaryKey

If this model relates to a database table, and the table's primary key is not named 'id', use this variable to tell Cake the name of the primary key.

$recursive

This sets the number of levels you wish Cake to fetch associated model data in find() and findAll() operations.

Imagine you have Groups which have many Users which in turn have many Articles.

Model::recursive options

$recursive = 0 Cake fetches Group data
$recursive = 1 Cake fetches a Group and its associated Users
$recursive = 2 Cake fetches a Group, its associated Users, and the Users' associated Articles

$transactional

Tells Cake whether or not to enable transactions for this model (i.e. begin/commit/rollback). Set to a boolean value. Only available for supporting databases.

$useTable

If the database table you wish to use isn't the plural form of the model name (and you don't wish to change the table name), set this variable to the name of the table you'd like this model to use.

$validate

An array used to validate the data passed to this model. See Chapter "Data Validation".

$useDbConfig

Remember the database settings you can configure in **/app/config/database.php**? Use this variable to switch between them - just use the name of the database connection variable you've created in your database configuration file. The default is, you guessed it, 'default'.

Section 4

Associations

Introduction

One of the most powerful features of CakePHP is the relational mapping provided by the model. In CakePHP, the links between tables are handled through associations. Associations are the glue between related logical units.

There are four types of associations in CakePHP:

1. hasOne
2. hasMany
3. belongsTo
4. hasAndBelongsToMany

When associations between models have been defined, Cake will automagically fetch models related to the model you are working with. For example, if a Post model is related to an Author model using a hasMany

association, making a call to $this->Post->findAll() in a controller will fetch Post records, as well as all the Author records they are related to.

To use the association correctly it is best to follow the CakePHP naming conventions (see Appendix "Cake Conventions"). If you use CakePHP's naming conventions, you can use scaffolding to visualize your application data, because scaffolding detects and uses the associations between models. Of course you can always customize model associations to work outside of Cake's naming conventions, but we'll save those tips for later. For now, let's just stick to the conventions. The naming conventions that concern us here are the foreign keys, model names, and table names.

Here's a review of what Cake expects for the names of these different elements: (see Appendix "Cake Conventions" for more information on naming)

1. Foreign Keys: [singular model name]_id. For example, a foreign key in the "authors" table pointing back to the Post a given Author belongs to would be named "post_id".

2. Table Names: [plural object name]. Since we'd like to store information about blog posts and their authors, the table names are "posts" and "authors", respectively.

3. Model Names: [CamelCased, singular form of table name]. The model name for the "posts" table is "Post", and the model name for the "authors" table is "Author".

CakePHP's scaffolding expects your associations to be in the same order as your colums. So if I have an Article that belongsTo three other models (Author, Editor, and Publisher), I would need three keys: author_id, editor_id, and publisher_id. Scaffolding would expect your associations in the same order as the keys in the table (e.g. first Author, second Editor, lastly Publisher).

In order to illustrate how some of these associations work, let's continue using the blog application as an example. Imagine that we're going to create a simple user management system for the blog. I suppose it goes without saying we'll want to keep track of Users, but we'd also like each user to have an associated Profile (User hasOne Profile). Users will also be able to create comments and remain associated to them (User hasMany Comments). Once we have the user system working, we'll move to allowing Posts to be related to Tag objects using the hasAndBelongsToMany relationship (Post hasAndBelongsToMany Tags).

Defining and Querying with hasOne

In order to set up this association, we'll assume that you've already created the User and Profile models. To define the hasOne assocation between them, we'll need to add an array to the models to tell Cake how they relate. Here's how that looks like:

/app/models/user.php hasOne

```
<?php
class User extends AppModel
{
    var $name = 'User';
    var $hasOne = array('Profile' =>
                        array('className'    => 'Profile',
                              'conditions'   => '',
                              'order'        => '',
                              'dependent'    =>  true,
                              'foreignKey'   => 'user_id'
                        )
                    );
}
?>
```

The $hasOne array is what Cake uses to build the association between the User and Profile models. Each key in the array allows you to further configure the association:

1. className (required): the classname of the model you'd like to associate

   For our example, we want to specify the 'Profile' model class name.

2. conditions: SQL condition fragments that define the relationship

   We could use this to tell Cake to only associate a Profile that has a green header, if we wished. To define conditions like this, you'd specify a SQL conditions fragment as the value for this key: "Profile.header_color = 'green'".

3. order: the ordering of the associated models

   If you'd like your associated models in a specific order, set the value for this key using an SQL order predicate: "Profile.name ASC", for example.

4. dependent: if set to true, the associated model is destroyed when this one is.

   For example, if the "Cool Blue" profile is associated to "Bob", and I delete the user "Bob", the profile "Cool Blue" will also be deleted.

5. foreignKey: the name of the foreign key that points to the associated model.

   This is here in case you're working with a database that doesn't follow Cake's naming conventions.

Now, when we execute find() or findAll() calls using the Profile model, we should see our associated User model there as well:

```
$user = $this->User->read(null, '25');
print_r($user);

//output:

Array
(
    [User] => Array
        (
            [id] => 25
            [first_name] => John
            [last_name] => Anderson
            [username] => psychic
            [password] => c4k3roxx
        )

    [Profile] => Array
        (
            [id] => 4
            [name] => Cool Blue
            [header_color] => aquamarine
            [user_id] = 25
        )
)
```

Defining and Querying with belongsTo

Now that a User can see its Profile, we'll need to define an association so Profile can see its User. This is done in Cake using the belongsTo assocation. In the Profile model, we'd do the following:

/app/models/profile.php belongsTo

```
<?php
class Profile extends AppModel
{
    var $name = 'Profile';
    var $belongsTo = array('User' =>
                           array('className'  => 'User',
                                 'conditions' => '',
                                 'order'      => '',
                                 'foreignKey' => 'user_id'
                           )
                     );
}
?>
```

The $belongsTo array is what Cake uses to build the association between the User and Profile models. Each key in the array allows you to further configure the association:

1. className (required): the classname of the model you'd like to associate

   For our example, we want to specify the 'User' model class name.

2. conditions: SQL condition fragments that define the relationship

   We could use this to tell Cake to only associate a User that is active. You would do this by setting the value of the key to be "User.active = '1'", or something similar.

3. order: the ordering of the associated models

   If you'd like your associated models in a specific order, set the value for this key using an SQL order predicate: "User.last_name ASC", for example.

4. foreignKey: the name of the foreign key that points to the associated model.

   This is here in case you're working with a database that doesn't follow Cake's naming conventions.

Now, when we execute find() or findAll() calls using the Profile model, we should see our associated User model there as well:

```
$profile = $this->Profile->read(null, '4');
print_r($profile);

//output:

Array
(

    [Profile] => Array
        (
            [id] => 4
            [name] => Cool Blue
            [header_color] => aquamarine
            [user_id] = 25
        )

    [User] => Array
        (
            [id] => 25
            [first_name] => John
            [last_name] => Anderson
            [username] => psychic
            [password] => c4k3roxx
        )
```

```
)
```

Defining and Querying with hasMany

Now that User and Profile models are associated and working properly, let's build our system so that User records are associated to Comment records. This is done in the User model like so:

/app/models/user.php hasMany

```
<?php
class User extends AppModel
{
    var $name = 'User';
    var $hasMany = array('Comment' =>
                         array('className'     => 'Comment',
                               'conditions'    => 'Comment.moderated = 1',
                               'order'         => 'Comment.created DESC',
                               'limit'         => '5',
                               'foreignKey'    => 'user_id',
                               'dependent'     => true,
                               'exclusive'     => false,
                               'finderQuery'   => ''
                         )
                );

    // Here's the hasOne relationship we defined earlier...
    var $hasOne = array('Profile' =>
                        array('className'    => 'Profile',
                              'conditions'   => '',
                              'order'        => '',
                              'dependent'    =>  true,
                              'foreignKey'   => 'user_id'
                        )
                );
}
?>
```

The $hasMany array is what Cake uses to build the association between the User and Comment models. Each key in the array allows you to further configure the association:

1. className (required): the classname of the model you'd like to associate

   For our example, we want to specify the 'Comment' model class name.

2. conditions: SQL condition fragments that define the relationship

   We could use this to tell Cake to only associate a Comment that has been moderated. You would do this by setting the value of the key to be "Comment.moderated = 1", or something similar.

3. order: the ordering of the associated models

   If you'd like your associated models in a specific order, set the value for this key using an SQL order predicate: "Comment.created DESC", for example.

4. limit: the maximum number of associated models you'd like Cake to fetch.

   For this example, we didn't want to fetch *all* of the user's comments, just five.

5. foreignKey: the name of the foreign key that points to the associated model.

   This is here in case you're working with a database that doesn't follow Cake's naming conventions.

6. dependent: if set to true, the associated model is destroyed when this one is.

   For example, if the "Cool Blue" profile is associated to "Bob", and I delete the user "Bob", the profile "Cool Blue" will also be deleted.

7. exclusive: If set to true, all the associated objects are deleted in one SQL statement without having their beforeDelete callback run.

   Good for use for simpler associations, because it can be much faster.

8. finderQuery: Specify a complete SQL statement to fetch the association.

   This is a good way to go for complex associations that depends on multiple tables. If Cake's automatic assocations aren't working for you, here's where you customize it.

Now, when we execute find() or findAll() calls using the User model, we should see our associated Comment models there as well:

```
$user = $this->User->read(null, '25');
print_r($user);

//output:

Array
(
    [User] => Array
        (
            [id] => 25
            [first_name] => John
            [last_name] => Anderson
            [username] => psychic
            [password] => c4k3roxx
        )

    [Profile] => Array
        (
            [id] => 4
            [name] => Cool Blue
            [header_color] => aquamarine
            [user_id] = 25
        )

    [Comment] => Array
        (
            [0] => Array
                (
                    [id] => 247
                    [user_id] => 25
                    [body] => The hasMany assocation is nice to have.
                )

            [1] => Array
                (
                    [id] => 256
                    [user_id] => 25
                    [body] => The hasMany assocation is really nice to have.
                )

            [2] => Array
                (
                    [id] => 269
                    [user_id] => 25
                    [body] => The hasMany assocation is really, really nice to have.
                )

            [3] => Array
```

```
                (
                    [id] => 285
                    [user_id] => 25
                    [body] => The hasMany assocation is extremely nice to have.
                )

            [4] => Array
                (
                    [id] => 286
                    [user_id] => 25
                    [body] => The hasMany assocation is super nice to have.
                )

        )
)
```

While we won't document the process here, it would be a great idea to define the "Comment belongsTo User" association as well, so that both models can see each other. Not defining assocations from both models is often a common gotcha when trying to use scaffolding.

Defining and Querying with hasAndBelongsToMany

Now that you've mastered the simpler associations, let's move to the last assocation: hasAndBelongsToMany (or HABTM). This last one is the hardest to wrap your head around, but it is also one of the most useful. The HABTM association is useful when you have two Models that are linked together with a join table. The join table holds the individual rows that are related to each other.

The difference between hasMany and hasAndBelongsToMany is that with hasMany, the associated model is not shared. If a User hasMany Comments, it is the *only* user associated to those comments. With HABTM, the associated models are shared. This is great for what we're about to do next: associate Post models to Tag models. When a Tag belongs to a Post, we don't want it to be 'used up', we want to continue to associate it to other Posts as well.

In order to do this, we'll need to set up the correct tables for this association. Of course you'll need a "tags" table for you Tag model, and a "posts" table for your posts, but you'll also need to create a join table for this association. The naming convention for HABTM join tables is [plural model name1]_[plural model name2], where the model names are in alphabetical order:

HABTM Join Tables: Sample models and their join table names

1. Posts and Tags: posts_tags
2. Monkeys and IceCubes: ice_cubes_monkeys
3. Categories and Articles: articles_categories

HABTM join tables need to at least consist of the two foreign keys of the models they link. For our example, "post_id" and "tag_id" is all we'll need.

Here's what the SQL dumps will look like for our Posts HABTM Tags example:

```
--
-- Table structure for table `posts`
--

CREATE TABLE `posts` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `user_id` int(10) default NULL,
  `title` varchar(50) default NULL,
  `body` text,
```

```
  `created` datetime default NULL,
  `modified` datetime default NULL,
  `status` tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (`id`)
) TYPE=MyISAM;

-- --------------------------------------------------------

--
-- Table structure for table `posts_tags`
--

CREATE TABLE `posts_tags` (
  `post_id` int(10) unsigned NOT NULL default '0',
  `tag_id` int(10) unsigned NOT NULL default '0',
  PRIMARY KEY  (`post_id`,`tag_id`)
) TYPE=MyISAM;

-- --------------------------------------------------------

--
-- Table structure for table `tags`
--

CREATE TABLE `tags` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `tag` varchar(100) default NULL,
  PRIMARY KEY  (`id`)
) TYPE=MyISAM;
```

With our tables set up, let's define the association in the Post model:

/app/models/post.php hasAndBelongsToMany

```
<?php
class Post extends AppModel
{
    var $name = 'Post';
    var $hasAndBelongsToMany = array('Tag' =>
                               array('className'    => 'Tag',
                                     'joinTable'    => 'posts_tags',
                                     'foreignKey'   => 'post_id',
                                     'associationForeignKey'=> 'tag_id',
                                     'conditions'   => '',
                                     'order'        => '',
                                     'limit'        => '',
                                     'unique'       => true,
                                     'finderQuery'  => '',
                                     'deleteQuery'  => '',
                               )
                               );
}
?>
```

The $hasAndBelongsToMany array is what Cake uses to build the association between the Post and Tag models. Each key in the array allows you to further configure the association:

1. className (required): the classname of the model you'd like to associate

   For our example, we want to specify the 'Tag' model class name.

2. joinTable: this is here for a database that doesn't adhere to Cake's naming conventions. If your table doesn't look like [plural model1]_[plural model2] in lexical order, you can specify the name of your table here.

3. foreignKey: the name of the foreign key in the join table that points to the current model.

This is here in case you're working with a database that doesn't follow Cake's naming conventions.

4. associationForeignKey: the name of the foreign key that points to the associated model.
5. conditions: SQL condition fragments that define the relationship

   We could use this to tell Cake to only associate a Tag that has been approved. You would do this by setting the value of the key to be "Tag.approved = 1", or something similar.
6. order: the ordering of the associated models

   If you'd like your associated models in a specific order, set the value for this key using an SQL order predicate: "Tag.tag DESC", for example.
7. limit: the maximum number of associated models you'd like Cake to fetch.

   Used to limit the number of associated Tags to be fetched.
8. unique: If set to true, duplicate associated objects will be ignored by accessors and query methods.

   Basically, if the associations are distinct, set this to true. That way the Tag "Awesomeness" can only be assigned to the Post "Cake Model Associations" once, and will only show up once in result arrays.
9. finderQuery: Specify a complete SQL statement to fetch the association.

   This is a good way to go for complex associations that depends on multiple tables. If Cake's automatic assocations aren't working for you, here's where you customize it.
10. deleteQuery: A complete SQL statement to be used to remove assocations between HABTM models.

    If you don't like the way Cake is performing deletes, or your setup is customized in some way, you can change the way deletion works by supplying your own query here.

Now, when we execute find() or findAll() calls using the Post model, we should see our associated Tag models there as well:

```
$post = $this->Post->read(null, '2');
print_r($post);

//output:

Array
(
    [Post] => Array
        (
            [id] => 2
            [user_id] => 25
            [title] => Cake Model Associations
            [body] => Time saving, easy, and powerful.
            [created] => 2006-04-15 09:33:24
            [modified] => 2006-04-15 09:33:24
            [status] => 1
        )

    [Tag] => Array
        (
            [0] => Array
                (
                    [id] => 247
                    [tag] => CakePHP
                )
```

```
            [1] => Array
                (
                    [id] => 256
                    [tag] => Powerful Software
                )
        )
)
```

Saving Related Model Data

One important thing to remember when working with associated models is that saving model data should always be done by the corresponding Cake model. If you are saving a new Post and its associated Comments, then you would use both Post and Comment models during the save operation.

If neither of the associated models exists in the system yet (for example, you want to save a new Post and a related Comment at the same time), you'll need to first save the primary, or parent model. To get an idea of how this works, let's imagine that we have an action in our PostsController that handles the saving of a new Post and a related Comment. The example action shown below will assume that you've posted a single Post and a single Comment.

/app/controllers/posts_controller.php (partial)

```
function add()
{
    if (!empty($this->data))
    {
        //We can save the Post data:
        //it should be in $this->data['Post']

        $this->Post->save($this->data);

        //Now, we'll need to save the Comment data
        //But first, we need to know the ID for the
        //Post we just saved...

        $post_id = $this->Post->getLastInsertId();

        //Now we add this information to the save data
        //and save the comment.

        $this->data['Comment']['post_id'] = $post_id;

        //Because our Post hasMany Comments, we can access
        //the Comment model through the Post model:

        $this->Post->Comment->save($this->data);

    }
}
```

If, however, the parent model already exists in the system (for example, adding a Comment to an existing Post), you need to know the ID of the parent model before saving. You could pass this ID as a URL parameter, or as a hidden element in a form...

/app/controllers/posts_controller.php (partial)

```
//Here's how it would look if the URL param is used...
function addComment($post_id)
{
    if (!empty($this->data))
    {
        //You might want to make the $post_id data more safe,
        //but this will suffice for a working example..
```

```
        $this->data['Comment']['post_id'] = $post_id;

        //Because our Post hasMany Comments, we can access
        //the Comment model through the Post model:

        $this->Post->Comment->save($this->data);
    }
}
```

If the ID was passed as a hidden element in the form, you might want to name the field (if you're using the HtmlHelper) so it ends up in the posted data where it needs to be:

```
If the ID for the post is at $post['Post']['id']...

<?php echo $html->hidden('Comment/post_id', array('value' => $post['Post']['id'])); ?>
```

Done this way, the ID for the parent Post model can be accessed at `$this->data['Comment']['post_id']`, and is all ready for a simple `$this->Post->Comment->save($this->data)` call.

These same basic techniques will work if you're saving multiple child models, just place those save() calls in a loop (and remember to clear the model information using Model::create()).

In summary, if you're saving associated data (for belongsTo, hasOne, and hasMany relations), the main point is getting the ID of the parent model and saving it to the child model.

Saving hasAndBelongsToMany Relations

Saving models that are associated by hasOne, belongsTo, and hasMany is pretty simple: you just populate the foreign key field with the ID of the associated model. Once that's done, you just call the save() method on the model, and everything gets linked up correctly.

With hasAndBelongsToMany, its a bit trickier, but we've gone out of our way to make it as simple as possible. In keeping along with our example, we'll need to make some sort of form that relates Tags to Posts. Let's now create a form that creates posts, and associates them to an existing list of Tags.

You might actually like to create a form that creates new tags and associates them on the fly - but for simplicity's sake, we'll just show you how to associate them and let you take it from there.

When you're saving a model on its own in Cake, the tag name (if you're using the Html Helper) looks like 'Model/field_name'. Let's just start out with the part of the form that creates our post:

/app/views/posts/add.thtml Form for creating posts

```
<h1>Write a New Post</h1>
<table>
    <tr>
        <td>Title:</td>
        <td><?php echo $html->input('Post/title')?></td>
    </tr>
    <tr>
        <td>Body:<td>
        <td><?php echo $html->textarea('Post/body')?></td>
    </tr>
    <tr>
        <td colspan="2">
            <?php echo $html->hidden('Post/user_id',
array('value'=>$this->controller->Session->read('User.id')))?>
            <?php echo $html->hidden('Post/status' , array('value'=>'0'))?>
            <?php echo $html->submit('Save Post')?>
        </td>
```

```
    </tr>
</table>
```

The form as it stands now will just create Post records. Let's add some code to allow us to bind a given Post to one or many Tags:

/app/views/posts/add.thtml (Tag association code added)

```
<h1>Write a New Post</h1>
<table>
    <tr>
        <td>Title:</td>
        <td><?php echo $html->input('Post/title')?></td>
    </tr>
    <tr>
        <td>Body:</td>
        <td><?php echo $html->textarea('Post/body')?></td>
    </tr>
    <tr>
        <td>Related Tags:</td>
        <td><?php echo $html->selectTag('Tag/Tag', $tags, null, array('multiple' =>
'multiple')) ?>
        </td>
    </tr>
    <tr>
        <td colspan="2">
            <?php echo $html->hidden('Post/user_id',
array('value'=>$this->controller->Session->read('User.id')))?>
            <?php echo $html->hidden('Post/status' , array('value'=>'0'))?>
            <?php echo $html->submit('Save Post')?>
        </td>
    </tr>
</table>
```

In order for a call to $this->Post->save() in the controller to save the links between this new Post and its associated Tags, the name of the field must be in the form "Tag/Tag" (the rendered name attribute would look something like 'data[ModelName][ModelName][]'). The submitted data must be a single ID, or an array of IDs of linked records. Because we're using a multiple select here, the submitted data for Tag/Tag will be an array of IDs.

The $tags variable here is just an array where the keys are the IDs of the possible Tags, and the values are the displayed names of the Tags in the multi-select element.

Changing Associations on the Fly using bindModel() and unbindModel()

You might occasionally wish to change model association information for exceptional situations when building your application. If your association settings in the model file are giving you too much (or not enough) information, you can use two model functions to bind and unbind model associations for your next find.

Let's set up a few models so we can see how bindModel() and unbindModel() work. We'll start with two models:

leader.php and follower.php

```
<?php

class Leader extends AppModel
{
    var $name = 'Leader';
```

```
    var $hasMany = array(
        'Follower' => array(
            'className' => 'Follower',
            'order'     => 'Follower.rank'
        )
    );
}

?>

<?php

class Follower extends AppModel
{
    var $name = 'Follower';
}

?>
```

Now, in a LeadersController, we can use the find() method in the Leader Model to come up with a Leader and its associated followers. As you can see above, the association array in the Leader model defines a "Leader hasMany Followers" relationship. For demonstration purposes, let's use unbindModel() to remove that association mid-controller.

leaders_controller.php (partial)

```
function someAction()
{
    //This fetches Leaders, and their associated Followers
    $this->Leader->findAll();

    //Let's remove the hasMany...
    $this->Leader->unbindModel(array('hasMany' => array('Follower')));

    //Now a using a find function will return Leaders, with no Followers
    $this->Leader->findAll();

    //NOTE: unbindModel only affects the very next find function.
    //An additional find call will use the configured association information.

    //We've already used findAll() after unbindModel(), so this will fetch
    //Leaders with associated Followers once again...
    $this->Leader->findAll();
}
```

The unbindModel() function works similarly with other associations: just change the name of the association type and model classname. The basic usage for unbindModel() is:

Generic unbindModel() example

```
$this->Model->unbindModel(array('associationType' => array('associatedModelClassName')));
```

Now that we've successfully removed an association on the fly, let's add one. Our as-of-yet unprincipled Leader needs some associated Principles. The model file for our Principle model is bare, except for the var $name statement. Let's associate some Principles to our Leader on the fly (but only for just the following find function call):

leaders_controller.php (partial)

```
funciton anotherAction()
{
    //There is no Leader hasMany Principles in the leader.php model file, so
    //a find here, only fetches Leaders.
```

```
    $this->Leader->findAll();

    //Let's use bindModel() to add a new association to the Principle model:
    $this->Leader->bindModel(
        array('hasMany' => array(
                'Principle' => array(
                    'className' => 'Principle'
                )
            )
        )
    );

    //Now that we're associated correctly, we can use a single find function
    //to fetch Leaders with their associated principles:
    $this->Leader->findAll();
}
```

The bindModel() function can be handy for creating new assocations, but it can also be useful if you want to change the sorting or other parameters in a given association on the fly.

There you have it. The basic usage for bindModel is to encapsulate a normal association array inside an array who's key is named after the type of assocation you are trying to create:

Generic bindModel() example

```
$this->Model->bindModel(
        array('associationName' => array(
                'associatedModelClassName' => array(
                    // normal association keys go here...
                )
            )
        )
    );
```

Please note that your tables will need to be keyed correctly (or association array properly configured) to bind models on the fly.

CakePHP: The Manual

Controllers

Section 1

What is a controller?

A controller is used to manage the logic for a certain section of your application. Most commonly, controllers are used to manage the logic for a single model. For example, if you were building a site that manages a video collection, you might have a VideoController and a RentalController managing your videos and rentals, respectively. **In Cake, controller names are always plural.**

Your application's controllers are classes that extend the Cake AppController class, which in turn extends a core Controller class. Controllers can include any number of actions: functions used in your web application to display views.

The AppController class can be defined in **/app/app_controller.php**
and it should contain methods that are shared between two or more controllers. It itself extends the Controller class which is a standard Cake library.

An action is a single functionality of a controller. It is run automatically by the Dispatcher if an incoming page request specifies it in the routes configuration. Returning to our video collection example, our VideoController might contain the view(), rent(), and search() actions. The controller would be found in /app/controllers/videos_controller.php and contain:

```
class VideosController extends AppController
{
    function view($id)
    {
        //action logic goes here..
    }

    function rent($customer_id, $video_id)
    {
        //action logic goes here..
    }

    function search($query)
    {
        //action logic goes here..
    }
}
```

You would be able to access these actions using the following example URLs:

```
http://www.example.com/videos/view/253
http://www.example.com/videos/rent/5124/0-235253
http://www.example.com/videos/search/hudsucker+proxy
```

But how would these pages look? You would need to define a view for each of these actions - check it out in the next chapter, but stay with me: the following sections will show you how to harness the power of the Cake controller and use it to your advantage. Specifically, you'll learn how to have your controller hand data to the view, redirect the user, and much more.

Section 2

Controller Functions

While this section will treat most of the often-used functions in Cake's Controller, it's important to remember to use http://api.cakephp.org for a full reference.

Interacting with your Views

- **set**
- string *$var*
- mixed *$value*

This function is the main way to get data from your controller to your view. You can use it to hand over anything: single values,

whole arrays, etc. Once you've used set(), the variable can be accessed in your view: doing set('color', 'blue') in your controller makes $color available in the view.

- **validateErrors**

Returns the number of errors generated by an unsuccessful save.

- **validate**

Validates the model data according to the model's validation rules. For more on validation, see Chapter "Data Validation".

- **render**
- string *$action*
- string *$layout*
- string *$file*

You may not often need this function, because render is automatically called for you at the end of each controller action, and the view named after your action is rendered. Alternatively, you can call this function to render the view at any point in the controller logic.

User Redirection

- **redirect**
- string *$url*

Tell your users where to go using this function. The URL passed here can be a Cake internal URL, or a fully qualified URL (http://...).

- **flash**
- string *$message*
- string *$url*
- int *$pause*

This function shows $message for $pause seconds inside of your flash layout (found in **app/views/layouts/flash.thtml**) then redirects the user to the specified $url.

Cake's `redirect()` and `flash()` functions do not include an `exit()` call. If you wish your application to halt after a `redirect()` or `flash()`, you'll need to include your own `exit()` call immediately after. You may also want to `return` rather than `exit()`, depending on your situation (for example, if you need some callbacks to execute).

Controller Callbacks

Cake controllers feature a number of callbacks you can use to insert logic before or after important controller functions. To utilize this functionality, declare these functions in your controller using the parameters and return values detailed here.

- **beforeFilter**

Called before every controller action. A useful place to check for active sessions and check roles.

- **afterFilter**

Called after every controller action.

- **beforeRender**

Called after controller logic, and just before a view is rendered.

Other Useful Functions

While these are functions part of Cake's Object class, they are also available inside the Controller:

- **requestAction**
- string *$url*
- array *$extra*

This function calls a controller's action from any location and returns the rendered view. The $url is a Cake URL (/controllername/actionname/params). If the $extra array includes a 'return' key, AutoRender is automatically set to true for the controller action.

You can use requestAction to get data from another controller action, or get a fully rendered view from a controller.

First, getting data from a controller is simple. You just use requestAction in the view where you need the data.

```
// Here is our simple controller:

class UsersController extends AppController
{
    function getUserList()
    {
        return $this->User->findAll();
    }
}
```

Imagine that we needed to create a simple table showing the users in the system. Instead of duplicating code in another controller, we can get the data from UsersController::getUserList() instead by using requestAction().

```
class ProductsController extends AppController
{
    function showUserProducts()
    {
        $this->set('users', $this->requestAction('/users/getUserList'));

        // Now the $users variable in the view will have the data from
        // UsersController::getUserList().
    }
}
```

If you have an often used element in your application that is not static, you might want to use requestAction() to inject it into your views. Let's say that rather than just passing the data from UsersController::getUserList, we actually wanted to render that action's view (which might consist of a table), inside another controller. This saves us from duplicating view code.

```
class ProgramsController extends AppController
{
    function viewAll()
    {
        $this->set('userTable', $this->requestAction('/users/getUserList', array('return')));

        // Now, we can echo out $userTable in this action's view to
        // see the rendered view that is also available at /users/getUserList.
    }
}
```

Please note that actions called using requestAction() are rendered using an empty layout - this way you don't have to worry about layouts getting rendered inside of layouts.

The requestAction() function is also useful in AJAX situations where a small element of a view needs to be populated before or during an AJAX update.

- **log**
- string *$message*
- int *$type = LOG_ERROR*

You can use this function to log different events that happen within your web application. Logs can be found inside Cake's **/tmp** directory.

If the $type is equal to the PHP constant LOG_DEBUG, the message is written to the log as a debug message. Any other type is written to the log as an error.

```
// Inside a controller, you can use log() to write entries:

$this->log('Mayday! Mayday!');

//Log entry:

06-03-28 08:06:22 Error: Mayday! Mayday!

$this->log("Look like {$_SESSION['user']} just logged in.", LOG_DEBUG);
```

```
//Log entry:

06-03-28 08:06:22 Debug: Looks like Bobby just logged in.
```

- **postConditions**
- array *$data*

A method to which you can pass $this->data, and it will pass back an array formatted as a model conditions array.

For example, if I have a person search form:

```
// app/views/people/search.thtml:

<?php echo $html->input('Person/last_name'); ?>
```

Submitting the form with this element would result in the following $this->data array:

```
Array
(
    [Person] => Array
        (
            [last_name] => Anderson
        )
)
```

At this point, we can use postConditions() to format this data to use in model:

```
// app/controllers/people_controller.php:

$conditions = $this->postConditions($this->data);

// Yields an array looking like this:

Array
(
    [Person.last_name] => Anderson
)

// Which can be used in model find operations:

$this->Person->findAll($conditions);
```

Section 3

Controller Variables

Manipulating a few special variables inside of your controller allows you to take advantage of some extra Cake functionality:

$name

PHP 4 doesn't like to give us the name of the current class in CamelCase. Use this variable to set the correct CamelCased name of your class if you're running into problems.

$uses

Does your controller use more than one model? Your FragglesController will automatically load $this->Fraggle, but if you want access to $this->Smurf as well, try adding something like the following to your controller:

```
var $uses = array('Fraggle','Smurf');
```

Please notice how you also need to include your Fraggle model in the $uses array, even though it was automatically available before.

$helpers

Use this variable to have your controller load helpers into its views. The HTML helper is automatically loaded, but you can use this variable to specify a few others:

```
var $helpers = array('Html','Ajax','Javascript');
```

Remember that you will need to include the HtmlHelper in the $helpers array if you intend to use it. It is normally available by default, but if you define $helpers without it, you'll get error messages in your views.

$layout

Set this variable to the name of the layout you would like to use for this controller.

$autoRender

Setting this to **false** will stop your actions from automatically rendering.

$beforeFilter

If you'd like a bit of code run every time an action is called (and before any of that action code runs), use $beforeFilter. This functionality is really nice for access control - you can check to see a user's permissions before any action takes place. Just set this variable using an array containing the controller action(s) you'd like to run:

```
class ProductsController extends AppController
{
    var $beforeFilter = array('checkAccess');

    function checkAccess()
    {
        //Logic to check user identity and access would go here....
    }

    function index()
    {
        //When this action is called, checkAccess() is called first.
    }
}
```

$components

Just like $helpers and $uses, this variable is used to load up components you will need:

```
var $components = array('acl');
```

Section 4

Controller Parameters

Controller parameters are available at **$this->params**
in your Cake controller. This variable is used to get data into the controller and provide access to information about the current request. The most common usage of $this->params is to get access to information that has been handed to the controller via POST or GET operations.

$this->data

Used to handle POST data sent from HTML Helper forms to the controller.

```
// A HTML Helper is used to create a form element

$html->input('User/first_name');

// When rendered as HTML it looks like:

<input name="data[User][first_name]" value="" type="text" />

// And when submitted to the controller via POST,
// shows up in $this->data['User']['first_name']

Array
(
    [data] => Array
        (
            [User] => Array
                (
                    [username] => mrrogers
                    [password] => myn3ighb0r
                    [first_name] => Mister
                    [last_name] => Rogers
                )

        )
)
```

$this->params['form']

Any POST data from any form is stored here, including information also found in $_FILES.

$this->params['bare']

Stores '1' if the current layout is bare, '0' if not.

$this->params['ajax']

Stores '1' if the current layout is ajax, '0' if not.

$this->params['controller']

Stores the name of the current controller handling the request. For example, if the URL /posts/view/1 was called, $this->params['controller'] would equal "posts".

$this->params['action']

Stores the name of the current action handling the request. For example, if the URL /posts/view/1 was called, $this->params['action'] would equal "view".

$this->params['pass']

Stores the GET query string passed with the current request. For example, if the URL /posts/view/?var1=3&var2=4 was called, $this->params['pass'] would equal "?var1=3&var2=4".

$this->params['url']

Stores the current URL requested, along with key-value pairs of get variables. For example, if the URL /posts/view/?var1=3&var2=4 was called, $this->params['url'] would look like this:

```
[url] => Array
        (
            [url] => posts/view
            [var1] => 3
            [var2] => 4
        )
```

CakePHP: The Manual

Views

Section 1

Views

A view is a page template, usually named after an action. For example, the view for **PostsController::add()** would be found at **/app/views/posts/add.thtml**. Cake views are quite simply PHP files, so you can use any PHP code inside them. Although most of your view files will contain HTML, a view could be any perspective on a certain set of data, be it XML, and image, etc.

In the view template file, you can use the data from the corresponding Model. This data is passed as an array called **$data**. Any data that you've handed to the view using set() in the controller is also now available in your view.

The HTML helper is available in every view by default, and is by far the most commonly used helper in views. It is very helpful in creating forms, including scripts and media, linking and aiding in data validation. Please see section 1 in Chapter "Helpers" for a discussion on the HTML helper.

Most of the functions available in the views are provided by Helpers. Cake comes with a great set of helpers (discussed in Chapter "Helpers"), and you can also include your own. Because views shouldn't contain much logic, there aren't many well used public functions in the view class. One that is helpful is renderElement(), which will be discussed in section 1.2.

Layouts

A layout contains all the presentational code that wraps around a view. Anything you want to see in all of your views should be placed in your layout.

Layout files are placed in **/app/views/layouts**. Cake's default layout can be overridden by placing a new default layout at **/app/views/layouts/default.thtml**. Once a new default layout has been created, controller view code is placed inside of the default layout when the page is rendered.

When you create a layout, you need to tell Cake where to place your controller view code: to do so, make sure your layout includes a place for **$content_for_layout** (and optionally, **$title_for_layout**). Here's an example of what a default layout might look like:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title><?php echo $title_for_layout?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
</head>
<body>

<!-- If you'd like some sort of menu to show up on all of your views, include it here -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Here's where I want my views to be displayed -->
<?php echo $content_for_layout ?>

<!-- Add a footer to each displayed page -->
<div id="footer">...</div>
```

```
</body>
</html>
```

To set the title for the layout, it's easiest to do so in the controller, using the $pageTitle controller variable.

```
class UsersController extends AppController
{
    function viewActive()
    {
        $this->pageTitle = 'View Active Users';
    }
}
```

You can create as many layouts as you wish for your Cake site, just place them in the app/views/layouts directory, and switch between them inside of your controller actions using the controller's **$layout** variable, or setLayout() function.

For example, if a section of my site included a smaller ad banner space, I might create a new layout with the smaller advertising space and specify it as the layout for all controller's actions using something like:

```
var $layout = 'default_small_ad';
```

Elements

Many applications have small blocks of presentational code that needs to be repeated from page to page, sometimes in different places in the layout. Cake can help you repeat parts of your website that need to be reused. These reusable parts are called Elements. Ads, help boxes, navigational controls, extra menus, and callouts are often implemented in Cake as elements. An Element is basically a mini-view that can be included in other Views.

Elements live in the **/app/views/elements/** folder, and have the **.thtml** filename extension.

The Element by default has no access to any data. To give it access to data, you send it in as a named parameter in an array.

Calling an Element without parameters

```
<?php echo $this->renderElement('helpbox'); ?>
```

Calling an Element passing a data array

```
<?php echo
$this->renderElement('helpbox', array("helptext" => "Oh, this text is very helpful."));
 ?>
```

Inside the Element file, all the passed variables are available as the names of the keys of the passed array (much like how set() in the controller works with the views). In the above example, the **/app/views/elements/helpbox.thtml** file can use the **$helptext** variable. Of course, it would be more useful to pass an array to the Element.

Elements can be used to make a View more readable, placing the rendering of repeating elements in its own file. They can also help you re-use content fragments in your website.

CakePHP: The Manual

Components

Section 1

Presentation

Components are used to aid controllers in specific situations. Rather than extend Cake's core libraries, special functionality can be made into components.

A guy named olle on the IRC channel once said: A component is a sharable little "controllerette". We find that this is a good definition. The main goal in fact is: reusability. Components are to controllers what helpers are to views. The main difference is that components encapsulate **business logic** whereas helpers encapsulate presentation logic. This point actually is very important, a common confusing for new Bakers when trying to achieve reusability: I'm trying to do X, should this be a component or a helper?! Well, the answer is really simple, what does X do? Does it do business logic or presentation logic, perhaps both? If it's business logic then it's a component. If it's presentation logic then it's a helper. If it's both, then..well it's both a component and a helper. An example of the later case would be an authentification system. You would want to login, logout, restrict access, and test permissions of a user to a ressource (an action: edit, add, del.. or a url), this is business logic, so this auth system should be a component. But you also want to add some entries to the main menu when the user is logged in, and this is presentation logic.

Section 2

Creating your own

To create a component, add a file in **app/controllers/components/ directory.**

Let us assume you created **foo.php**. Inside of the file you need to define a class that corresponds to the file name (appending the word 'Component' to the file name). So in our case you would create the following contents:

A simple component

```
class FooComponent extends Object
{
    var $someVar = null;
    var $controller = true;

    function startup(&$controller)
    {
        // This method takes a reference to the controller which is loading it.
        // Perform controller initialization here.
    }

    function doFoo()
    {
        $this->someVar = 'foo';
    }
}
```

Now, to use your component, you need to add the following code in your controller's definition:

```
var $components = array('Foo');
```

Inside of that controller you could now use:

```
$this->Foo->doFoo();
```

A component gets access to the controller that loaded it through the startup() method shown above. This method is called immediately after Controller::beforeFilter(). This allows you to set component properties in the beforeFilter method, which the component can act on in itâ€™s startup() method.

To use your models inside of your components, you can create a new instance like this:

```
$foo =& new Foo();
```

You can also use other components inside your component. You simply have to declare in your component which components you want to use. In the example below it is the session component.

```
var $components = array('Session');
```

Section 3

Making your components public

If you think your component can be helpful to others, add it to CakeForge. A component that becomes increasingly useful for the community may some day be included in the core distribution.

Also check the snippet archive for user committed components.

CakePHP: The Manual

Helpers

Section 1

Helpers

Helpers are meant to provide functions that are commonly needed in views to format and present data in useful ways.

HTML

Introduction

The HTML helper is one of Cake's ways of making development less monotonous and more rapid. The HTML helper has two main goals: to aid in the insertion of often-repeated sections of HTML code, and to aid in the quick-and-easy creation of web forms. The following sections will walk you through the most important functions in the helper, but remember http://api.cakephp.org should always be used as a final reference.

Many of the functions in the HTML helper make use of a HTML tag definition file called **tags.ini.php**. Cake's core configuration contains a tags.ini.php, but if you'd like to make some changes, create a copy of **/cake/config/tags.ini.php** and place it in your **/app/config/** folder. The HTML helper uses the tag definitions in this file to generate tags you request. Using the HTML helper to create some of your view code can be helpful, as a change to the tags.ini.php file will result in a site-wide cascading change.

Additionally, if AUTO_OUTPUT is set to true in your core config file for your application (**/app/config/core.php**), the helper will automatically output the tag, rather than returning the value. This has been done in an effort to assuage those who dislike short tags (<?= ?>) or lots of echo() calls in their view code. Functions that include the $return parameter allow you to force-override the settings in your core config. Set $return to true if you'd like the HTML helper to return the HTML code regardless of any AUTO_OUTPUT settings.

HTML helper functions also include a $htmlAttributes parameter, that allow you to tack on any extra attributes on your tags. For example, if you had a tag you'd like to add a class attribute to, you'd pass this as the $htmlAttribute value:

```
array('class'=>'someClass')
```

Inserting Well-Formatted elements

If you'd like to use Cake to insert well-formed and often-repeated elements in your HTML code, the HTML helper is great at doing that. There are functions in this helper that insert media, aid with tables, and there's even guiListTree which creates an unordered list based on a PHP array.

- **charset**
- string *$charset*
- boolean *$return*

This is used to generate a charset META-tag.

- **css**

- string *$path*
- string *$rel = 'stylesheet'*
- array *$htmlAttributes*
- boolean *$return = false*

Creates a link to a CSS stylesheet. The $rel parameter allows you to provide a rel= value for the tag.

- **image**
- string *$path*
- array *$htmlAttributes*
- boolean *$return = false*

Renders an image tag. The code returned from this function can be used as an input for the link() function to automatically create linked images.

- **link**
- string *$title*
- string *$url*
- array *$htmlAttributes*
- string *$confirmMessage = false*
- boolean *$escapeTitle = true*
- boolean *$return = false*

Use this function to create links in your view. $confirmMessage is used when you need a JavaScript confirmation message to appear once the link is clicked. For example, a link that deletes an object should probably have a "Are you sure?" type message to confirm the action before the link is activated. Set $escapeTitle to true if you'd like to have the HTML helper escape the data you handed it in the $title variable.

- **tableHeaders**
- array *$names*
- array *$tr_options*
- array *$th_options*

Used to create a formatted table header.

- **tableCells**
- array *$data*
- array *$odd_tr_options*
- array *$even_tr_options*

Used to create a formatted set of table cells.

- **guiListTree**
- array *$data*
- array *$htmlAttributes*
- string *$bodyKey = 'body'*
- string *$childrenKey = 'children'*
- boolean *$return = false*

Generates a nested unordered list tree from an array.

Forms and Validation

The HTML helper really shines when it comes to quickening your form code in your views. It generates all your form tags, automatically fills values back in during error situations, and spits out error messages. To help

illustrate, let's walk through a quick example. Imagine for a moment that your application has a Note model, and you want to create controller logic and a view to add and edit Note objects. In your NotesController, you would have an edit action that might look something like the following:

Edit Action inside of the NotesController

```
function edit($id)
{
   //First, let's check to see if any form data has been
   //submitted to the action.
   if (!empty($this->data['Note']))
   {
      //Here's where we try to validate the form data (see Chap. 12)
      //and save it
      if ($this->Note->save($this->data['Note']))
      {
         //If we've successfully saved, take the user
         //to the appropriate place
         $this->flash('Your information has been saved.', '/notes/edit/' . $id);
         exit();
      }
      else
      {

         //Generate the error messages for the appropriate fields
         //this is not really necessary as save already does this, but it is an example
         //call $this->Note->validates($this->data['Note']); if you are not doing a save
         //then use the method below to populate the tagErrorMsg() helper method
         $this->validateErrors($this->Note);

         //And render the edit view code
         $this->render();
      }
   }

   // If we haven't received any form data, get the note we want to edit, and hand
   // its information to the view
   $this->set('note', $this->Note->find("id = $id"));
   $this->render();
}
```

Once we've got our controller set up, let's look at the view code (which would be found in **app/views/notes/edit.thtml**). Our Note model is pretty simple at this point as it only contains an id, a submitter's id and a body. This view code is meant to display Note data and allow the user to enter new values and save that data to the model.

The HTML helper is available in all views by default, and can be accessed using **$html.**

Specifically, let's just look at the table where the guts of the form are found:

Edit View code (edit.thtml) sample

```
<!-- This tag creates our form tag -->

<?php echo $html->formTag('/notes/edit/' . $html->tagValue('Note/id')?>

<table cellpadding="10" cellspacing="0">
<tr>
   <td align="right">Body: </td>
   <td>

      <!-- Here's where we use the HTML helper to render the text
           area tag and its possible error message the $note
           variable was created by the controller, and contains
           the data for the note we're editing. -->
      <?php echo
```

```
      $html->textarea('Note/body', array('cols'=>'60', 'rows'=>'10'));
      ?>
      <?php echo $html->tagErrorMsg('Note/body',
      'Please enter in a body for this note.') ?>
   </td>
</tr>
<tr>
   <td></td>
   <td>

      <!-- We can also use the HTML helper to include
           hidden tags inside our table -->

      <?php echo $html->hiddenTag('Note/id')?>
      <?php echo $html->hiddenTag('note/submitter_id',
$this->controller->Session->read('User.id'))?>
   </td>
</tr>
</table>

<!-- And finally, the submit button-->
<?php echo $html->submit()?>

</form>
```

Most of the form tag generating functions (along with tagErrorMsg) require you to supply a $fieldName. This $fieldName lets Cake know what data you are passing so that it can save and validate the data correclty. The string passed in the $fieldName parameter is in the form "modelname/fieldname." If you were going to add a new title field to our Note, you might add something to the view that looked like this:

```
<?php echo $html->input('Note/title') ?>
<?php echo $html->tagErrorMsg('Note/title', 'Please supply a title for this note.')?>
```

Error messages displayed by the tagErrorMsg() function are wrapped in <div class="error_message"></div> for easy CSS styling.

Here are the form tags the HTML helper can generate (most of them are straightforward):

- **submit**
- string *$buttonCaption*
- array *$htmlAttributes*
- boolean *$return = false*

- **password**
- string *$fieldName*
- array *$htmlAttributes*
- boolean *$return = false*

- **textarea**
- string *$fieldName*
- array *$htmlAttributes*
- boolean *$return = false*

- **checkbox**
- string *$fieldName*
- array *$htmlAttributes*
- boolean *$return = false*

- **file**
- string *$fieldName*
- array *$htmlAttributes*

- boolean *$return = false*

- **hidden**
- string *$fieldName*
- array *$htmlAttributes*
- boolean *$return = false*

- **input**
- string *$fieldName*
- array *$htmlAttributes*
- boolean *$return = false*

- **radio**
- string *$fieldName*
- array *$options*
- array *$inbetween*
- array *$htmlAttributes*
- boolean *$return = false*

- **tagErrorMsg**
- string *$fieldName*
- string *$message*

The HTML helper also includes a set of functions that aid in creating date-related option tags. The $tagName parameter should be handled in the same way as the $fieldName parameter. Just provide the name of the field this date option tag is relevant to. Once the data is processed, you'll see it in your controller with the part of the date it handles concatenated to the end of the field name. For example, if my Note had a deadline field that was a date, and my dayOptionTag $tagName parameter was set to 'note/deadline', the day data would show up in the $params variable once the form has been submitted to a controller action:

```
$this->data['Note']['deadline_day']
```

You can then use this information to concatenate the time data in a format that is friendly to your current database configuration. This code would be placed just before you attempt to save the data, and saved in the $data array used to save the information to the model.

Concatenating time data before saving a model (excerpt from NotesController)

```
function edit($id)
   {
      //First, let's check to see if any form data has been submitted to the action.
      if (!empty($this->data['Note']))
      {

         //Concatenate time data for storage...
         $this->data['Note']['deadline'] =
            $this->data['Note']['deadline_year'] . "-" .
            $this->data['Note']['deadline_month'] . "-" .
            $this->data['Note']['deadline_day'];

         //Here's where we try to validate the form data (see Chap. 10) and save it
         if ($this->Note->save($this->data['Note']))
         {

         ...
```

1. dayOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)

2. yearOptionTag ($tagName, $value=null, $minYear=null, $maxYear=null, $selected=null,

$optionAttr=null)

3. monthOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
4. hourOptionTag ($tagName, $value=null, $format24Hours=false, $selected=null, $optionAttr=null)
5. minuteOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
6. meridianOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
7. dateTimeOptionTag ($tagName, $dateFormat= 'DMY', $timeFormat= '12', $selected=null, $optionAttr=null)

AJAX

The Cake Ajax helper utilizes the ever-popular Prototype and script.aculo.us libraries for Ajax operations and client side effects. In order to use this helper, you must have a current version of the JavaScript libraries from http://script.aculo.us placed in **/app/webroot/js/**. In addition, any views that plan to use the Ajax Helper will need to include those libraries.

Most of the functions in this helper expect a special $options array as a parameter. This array is used to specify different things about your Ajax operation. Here are the different values you can specify:

AjaxHelper $options Keys

```
/* General Options */

$options['url']         // The URL for the action you want to be called.

$options['frequency']   // The number of seconds between remoteTimer() or
                        // observeField() checks are made.

$options['update']      // The DOM ID of the element you wish to update with
                        // the results of an Ajax operation.

$options['with']        // The DOM ID of the form element you wish to serialize
                        // and send with an Ajax form submission.

$options['type']        // Either 'asynchronous' (default), or 'synchronous'.
                        // Allows you to pick between operation types.

/* Callbacks : JS code to be executed at certain
   times during the XMLHttpRequest process */

$options['loading']     // JS code to be executed when the remote document
                        // is being loaded with data by the browser.

$options['loaded']      // JS code to be executed when the browser has finished
                        // loading the remote document.

$options['interactive'] // JS code to be executed when the user can interact
                        // with the remote document, even though it has not
                        // finished loading.

$options['complete']    // JS code to be called when the XMLHttpRequest is
                        // complete.

$options['confirm']     // Text to be displayed in a confirmation dialog before
                        // a XMLHttpRequest action begins.

$options['condition']   // JS condition to be met before the XMLHttpRequest
                        // is initiated.

$options['before']      // JS code to be called before request is initiated.
```

```
$options['after']       // JS code to be called immediately after request was
                        // initiated and before 'loading'.
```

Here are the helper's functions for making Ajax in Cake quick and easy:

- **link**
- string *$title*
- string *$href*
- array *$options*
- boolean *$confirm*
- boolean *$escapeTitle*

Displays linked text $title, which retrieves the remote document at $options['url'] and updates the DOM element $options['update']. Callbacks can be used with this function.

- **remoteFunction**
- array *$options*

This function creates the JavaScript needed to make a remote call. It is primarily used as a helper for linkToRemote. This isn't often used unless you need to generate some custom scripting.

- **remoteTimer**
- array *$options*

Periodically calls the specified action at $options['url'], every $options['frequency'] seconds (default is 10). Usually used to update a specified div (specified by $options['update']) with the results of the remote call. Callbacks can be used with this function.

- **form**
- string *$action*
- string *$type*
- array *$options*

Returns a form tag that will submit to the action at $action using XMLHttpRequest in the background instead of the regular reload-required POST submission. The form data from this form will act just as a normal form data would (i.e. it will be available in $this->params['form']). The DOM element specified by $options['update'] will be updated with the resulting remote document. Callbacks can be used with this function.

- **observeField**
- string *$field_id*
- array *$options*

Observes the field with the DOM ID specified by $field_id (every $options['frequency'] seconds) and calls the action at $options['url'] when its contents have changed. You can update a DOM element with ID $options['update'] or specify a form element using $options['with'] as well. Callbacks can be used with this function.

- **observeForm**
- string *$form_id*
- array *$options*

Works the same as observeField(), only this observes all the elements in a given form.

- **autoComplete**
- string *$field*
- string *$url*

- array *$options*

Renders a text field with ID $field with autocomplete. The action at $url should be able to return the autocomplete terms: basically, your action needs to spit out an unordered list (
) with list items that are the auto complete terms. If you wanted an autocomplete field that retrieved the subjects of your blog posts, your controller action might look something like:

```
function autocomplete ()
{
    $this->set('posts',
        $this->Post->findAll(
            "subject LIKE '{$this->data['Post']['subject']}'")
        );
    $this->layout = "ajax";
}
```

And your view for the autocomplete() action above would look something like:

```
<ul>
<?php foreach($posts as $post): ?>
<li><?php echo $post['Post']['subject']; ?></li>
<?php endforeach; ?>
</ul>
```

The actual auto-complete field as it would look in a view would look like this:

```
<form action="/users/index" method="POST">
    <?php echo $ajax->autoComplete('Post/subject', '/posts/autoComplete')?>
    <?php echo $html->submit('View Post')?>
</form>
```

The autoComplete() function will use this information to render a text field, and some divs that will be used to show the autocomplete terms supplied by your action. You might also want to style the view with something like the following:

```
<style type="text/css">

div.auto_complete {
    position         :absolute;
    width            :250px;
    background-color :white;
    border           :1px solid #888;
    margin           :0px;
    padding          :0px;
}

li.selected { background-color: #ffb; }

</style>
```

- **drag**
- string *$id*
- array *$options*

Makes the DOM element with ID $id draggable. There are some additional things you can specify using $options:

```
// (The version numbers refer to script.aculo.us versions)

$options['handle']     // (v1.0) Sets whether the element should only be
                       // draggable by an embedded handle. The value must be
                       // an element reference or element id.

$options['handle']     // (V1.5) As above, except now the value may be a
                       // string referencing a CSS class value. The first
```

```
                        // child/grandchild/etc. element found within the
                        // element that has this CSS class value will be used
                        // as the handle.

$options['revert']      // (V1.0) If set to true, the element returns to its
                        // original position when the drags ends.

$options['revert']      // (V1.5) Revert can also be an arbitrary function
                        // reference, called when the drag ends.

$options['constraint'] // If set to horizontal or vertical, the drag will
                        // be constrained to take place only horizontally or
                        // vertically.
```

- **drop**
- string *$id*
- array *$options*

Makes the DOM element with ID $id drop-able. There are some additional things you can specify using $options:

```
$options['accept']      // Set accept to a string or a JavaScript array of
                        // strings describing CSS classes. The Droppable will
                        // only accept Draggables that have one or more of
                        // these CSS classes.

$options['containment'] // The droppable element will only accept the
                        // draggable element if it is contained in the given
                        // elements (or element ids). Can be a single element
                        // or a JS array of elements.

$options['overlap']     //If set to horizontal or vertical, the droppable
                        // will only react to a draggable element if its
                        //overlapping by more than 50% in the given direction.
```

- **dropRemote**
- string *$id*
- array *$options*
- array *$ajaxOptions*

Used to create a drop target that initiates a XMLHttpRequest when a draggable element is dropped on it. The $options are the same as in drop(), and the $ajaxOptions are the same as in link().

- **sortable**
- string *$id*
- array *$options*

Makes a list or group of floated objects (specified by DOM element ID $id) sortable. The $options array can configure your sorting as follows:

```
$options['tag']         // Sets the kind of tag (of the child elements of the
                        // container) that will be made sortable. For UL and
                        // OL containers, this is LI, you have to provide
                        // the tag kind for other sorts of child tags.
                        // Defaults to 'li'.

$options['only']        // Further restricts the selection of child elements
                        // to only encompass elements with the given CSS class
                        // (or, if you provide an array of strings, on any of
                        // the classes).

$options['overlap']     // Either vertical(default) or horizontal. For
                        // floating sortables or horizontal lists, choose
                        // horizontal. Vertical lists should use vertical.
```

```
$options['constraint']  // Restricts the movement of draggable elements,
                        // 'vertical' or 'horizontal'.

$options['containment'] // Enables dragging and dropping between Sortables.
                        // Takes an array of elements or element-ids (of the
                        // containers).

$options['handle']      // Makes the created draggable elemetns use handles,
                        // see the handle option on drag().
```

- **editor**
- string *$id*
- string *$url*
- array *$options*

Creates an in-place ajax editor using the element with the DOM id supplied as the first parameter. When implemented, the element will highlight on mouseOver, and will turn into a single text input field when clicked. The second parameter is the URL that the edited data should be sent to. The action should also return the updated contents of the element. Additional options for the in-place editor can be found on the Script.aculo.us wiki.

Javascript

The JavaScript helper is used to aid the developer in outputting well-formatted Javascript-related tags and data.

- **codeBlock**
- string *$string*

Used to return $script placed within JavaScript <script> tags.

- **link**
- string *$url*

Returns a JavaScript include tag pointing to the script referenced by $url.

- **linkOut**
- string *$url*

Same as link(), only the include tag assumes that the script referenced by $url is not hosted on the same domain.

- **escapeScript**
- string *$script*

Escapes carriage returns and single and double quotes for JavaScript code segments.

- **event**
- string *$object*
- string *$event*
- string *$observer*
- boolean *$useCapture*

Attaches an event to an element. Used with the Prototype library.

- **cacheEvents**

Caches JavaScript events created with event().

- **writeEvents**

Writes cached events cached with cacheEvents().

- **includeScript**
- string *$script*

Number

The Number helper includes a few nice functions for formatting numerical data in your views.

- **precision**
- mixed *$number*
- int *$precision = 3*

Returns $number formatted to the level of precision specified by $precision.

- **toReadableSize**
- int *$sizeInBytes*

Returns a human readable size, given the $size supplied in bytes. Basically, you pass a number of bytes in, and this function returns the appropriate human-readable value in KB, MB, GB, or TB.

- **toPercentage**
- mixed *$number*
- int *$precision = 2*

Returns the given number formatted as a percentage, limited to the precision specified in $precision.

Text

The Text Helper provides methods that a developer may need for outputting well formatted text to the browser.

- **highlight**
- string *$text*
- string *$highlighter = '\1'*

Returns $text, with every occurance or $phrase wrapped with the tags specified in $highlighter.

- **stripLinks**
- string *$text*

Returns $text, with all HTML links (<a href= ...) removed.

- **autoLinkUrls**
- string *$text*
- array *$htmlOptions*

Returns $text with URLs wrapped in corresponding <a> tags.

- **autoLinkEmails**
- string *$text*
- array *$htmlOptions*

Returns $text with email addresses wrapped in corresponding <a> tags.

- **autoLink**
- string *$text*

- array *$htmlOptions*

Returns $text with URLs and emails wrapped in corresponding <a> tags.

- **truncate**
- string *$text*
- int *$length*
- string *$ending = '...'*

Returns the first $length number of characters of $text followed by $ending ('...' by default).

- **excerpt**
- string *$text*
- string *$phrase*
- int *$radius = 100*
- string *$ending = '...'*

Extracts an excerpt from the $text, grabbing the $phrase with a number of characters on each side determined by $radius.

- **autoLinkEmails**
- string *$text*
- boolean *$allowHtml = false*

Text-to-html parser, similar to Textile or RedCloth, only with a little different syntax.

Time

The Time Helper provides methods that a developer may need for outputting Unix timestamps and/or datetime strings into more understandable phrases to the browser.

Dates can be provided to all functions as either valid PHP datetime strings or Unix timestamps.

- **fromString**
- string *$dateString*

Returns a UNIX timestamp, given either a UNIX timestamp or a valid strtotime() date string.

- **nice**
- string *$dateString*
- boolean *$return = false*

Returns a nicely formatted date string. Dates are formatted as "D, M jS Y, H:i", or 'Mon, Jan 1st 2005, 12:00'.

- **niceShort**
- string *$dateString*
- boolean *$return = false*

Formats date strings as specified in nice(), but ouputs "Today, 12:00" if the date string is today, or "Yesterday, 12:00" if the date string was yesterday.

- **isToday**
- string *$dateString*

Returns true if given datetime string is today.

- **daysAsSql**
- string *$begin*
- string *$end*
- string *$fieldName*
- boolean *$return = false*

Returns a partial SQL string to search for all records between two dates.

- **dayAsSql**
- string *$dateString*
- string *$fieldName*
- boolean *$return = false*

Returns a partial SQL string to search for all records between two times occurring on the same day.

- **isThisYear**
- string *$dateString*
- boolean *$return = false*

Returns true if given datetime string is within current year.

- **wasYesterday**
- string *$dateString*
- boolean *$return = false*

Returns true if given datetime string was yesterday.

- **isTomorrow**
- string *$dateString*
- boolean *$return = false*

Returns true if given datetime string is tomorrow.

- **toUnix**
- string *$dateString*
- boolean *$return = false*

Returns a UNIX timestamp from a textual datetime description. Wrapper for PHP function strtotime().

- **toAtom**
- string *$dateString*
- boolean *$return = false*

Returns a date formatted for Atom RSS feeds.

- **toRSS**
- string *$dateString*
- boolean *$return = false*

Formats date for RSS feeds

- **timeAgoInWords**
- string *$dateString*
- boolean *$return = false*

Returns either a relative date or a formatted date depending on the difference between the current time and given

datetime. $datetime should be in a strtotime-parsable format like MySQL datetime.

- **relativeTime**
- string *$dateString*
- boolean *$return = false*

Works much like timeAgoInWords(), but includes the ability to create output for timestamps in the future as well (i.e. "Yesterday, 10:33", "Today, 9:42", and also "Tomorrow, 4:34").

- **relativeTime**
- string *$timeInterval*
- string *$dateString*
- boolean *$return = false*

Returns true if specified datetime was within the interval specified, else false. The time interval should be specifed with the number as well as the units: '6 hours', '2 days', etc.

Cache

Section 2

Creating Your Own Helpers

Have the need for some help with your view code? If you find yourself needing a specific bit of view logic over and over, you can make your own view helper.

Extending the Cake Helper Class

Let's say we wanted to create a helper that could be used to output a CSS styled link you needed in your application. In order to fit your logic in to Cake's existing Helper structure, you'll need to create a new class in /app/views/helpers. Let's call our helper LinkHelper. The actual php class file would look something like this:

/app/views/helpers/link.php

```
class LinkHelper extends Helper
{
    function makeEdit($title, $url)
    {
        // Logic to create specially formatted link goes here...
    }
}
```

There are a few functions included in Cake's helper class you might want to take advantage of:

- **output**
- string *$string*
- boolean *$return = false*

Decides whether to output or return a string based on AUTO_OUTPUT (see **/app/config/core.php**) and $return's value. You should use this function to hand any data back to your view.

- **loadConfig**

Returns your application's current core configuration and tag definitions.

Let's use output() to format our link title and URL and hand it back to the view.

/app/views/helpers/link.php (logic added)

```
class LinkHelper extends Helper
{
    function makeEdit($title, $url)
    {
        // Use the helper's output function to hand formatted
        // data back to the view:

        return $this->output("<div class=\"editOuter\"><a href=\"$url\"
class=\"edit\">$title</a></div>");
    }
}
```

Including other Helpers

You may wish to use some functionality already existing in another helper. To take advantage of that, you can specify helpers you wish to use with a $helpers array, formatted just as you would in a controller.

/app/views/helpers/link.php (using other helpers)

```
class LinkHelper extends Helper
{

    var $helpers = array('Html');

    function makeEdit($title, $url)
    {
        // Use the HTML helper to output
        // formatted data:

        $link = $this->Html->link($title, $url, array('class' => 'edit'));

        return $this->output("<div class=\"editOuter\">$link</div>");
    }
}
```

Using your Custom Helper

Once you've created your helper and placed it in /app/views/helpers/, you'll be able to include it in your controllers using the special variable $helpers.

```
class ThingsController
{
    var $helpers = array('Html', 'Link');
}
```

Remember to include the HTML helper in the array if you plan to use it elsewhere. The naming conventions are similar to that of models.

1. LinkHelper = class name
2. link = key in helpers array
3. link.php = name of php file in **/app/views/helpers**.

Contributing

Please consider giving your code back to Cake - contact one of the developers using our Trac system or the mailing list, or open a new CakeForge project to share your new helper with others.

CakePHP: The Manual

Cake's Global Constants And Functions

Here are some globally available constants and functions that you might find useful as you build your application with Cake.

Section 1

Global Functions

Here are Cake's globally available functions. Many of them are convenience wrappers for long-named PHP functions, but some of them (like vendor() and uses()) can be used to include code, or perform other useful functions. Chances are if you're wanting a nice little function to do something annoying over and over, it's here.

- **config**

Loads Cake's core configuration file. Returns true on success.

- **uses**
- string *$lib1*
- string *$lib2*...

Used to load Cake's core libaries (found in **cake/libs/**). Supply the name of the lib filename without the '.php' extension.

```
uses('sanitize', 'security');
```

- **vendor**
- string *$lib1*
- string *$lib2*...

Used to load external libraries found in the /vendors directory. Supply the name of the lib filename without the '.php' extension.

```
vendor('myWebService', 'nusoap');
```

- **debug**
- mixed *$var*
- boolean *$showHtml = false*

If the application's DEBUG level is non-zero, the $var is printed out. If $showHTML is true, the data is rendered to be browser-friendly.

- **a**

Returns an array of the parameters used to call the wrapping function.

```
function someFunction()
{
    echo print_r(a('foo', 'bar'));
}

someFunction();

// output:

array(
    [0] => 'foo',
    [1] => 'bar'
)
```

- **aa**

Used to create associative arrays formed from the parameters used to call the wrapping function.

```
echo aa('a','b');
```

```
// output:

array(
    'a' => 'b'
)
```

- **e**
- string *$text*

Convenience wrapper for echo().

- **low**

Convenience wrapper for strtolower().

- **up**

Convenience wrapper for strtoupper().

- **r**
- string *$search*
- string *$replace*
- string *$subject*

Convenience wrapper for str_replace().

- **pr**
- mixed *$data*

Convenience function equivalent to:

```
echo "<pre>" . print_r($data) . "</pre>";
```

Only prints out information if DEBUG is non-zero.

- **am**
- array *$array1*
- array *$array2...*

Merges and returns the arrays supplied in the parameters.

- **env**
- string *$key*

Gets an environment variable from available sources. Used as a backup if $_SERVER or $_ENV are disabled.

This function also emulates PHP_SELF and DOCUMENT_ROOT on unsupporting servers. In fact, it's a good idea to always use env() instead of $_SERVER or getenv() (especially if you plan to distribute the code), since it's a full emulation wrapper.

- **cache**
- string *$path*
- string *$expires*
- string *$target = 'cache'*

Writes the data in $data to the path in /app/tmp specified by $path as a cache. The expiration time specified by $expires must be a valid strtotime() string. The $target of the cached data can either be 'cache' or 'public'.

- **clearCache**
- string *$search*
- string *$path = 'views'*
- string *$ext*

Used to delete files in the cache directories, or clear contents of cache directories.

If $search is a string, matching cache directory or file names will be removed from the cache. The $search parameter can also be

passed as an array of names of files/directories to be cleared. If empty, all files in /app/tmp/cache/views will be cleared.

The $path parameter can be used to specify which directory inside of /tmp/cache is to be cleared. Defaults to 'views'.

The $ext param is used to specify files with a certain file extention you wish to clear.

- **stripslashes_deep**
- array *$array*

Recursively strips slashes from all values in an array.

- **countdim**
- array *$array*

Returns the number of dimensions in the supplied array.

- **fileExistsInPath**
- string *$file*

Searches the current include path for a given filename. Returns the path to that file if found, false if not found.

- **convertSlash**
- string *$string*

Converts forward slashes to underscores and removes first and last underscores in a string.

Section 2

CakePHP Core Definition Constants

ACL_CLASSNAME: the name of the class currently performing and managing ACL for CakePHP. This constant is in place to allow for users to integrate third party classes.

ACL_FILENAME: the name of the file where the class ACL_CLASSNAME can be found inside of.

AUTO_SESSION: if set to false, session_start() is not automatically called during requests to the application.

CACHE_CHECK: if set to false, view caching is turned off for the entire application.

CAKE_SECURITY: determines the level of session security for the application in accorance with CAKE_SESSION_TIMEOUT. Can be set to 'low', 'medium', or 'high'. Depending on the setting, CAKE_SESSION_TIMEOUT is multiplied according to the following:

1. low: 300
2. medium: 100
3. high: 10

CAKE_SESSION_COOKIE: the name of session cookie for the application.

CAKE_SESSION_SAVE: set to 'php', 'file', or 'database'.

1. php: Cake uses PHP's default session handling (usually defined in php.ini)
2. file: Session data is stored and managed in /tmp
3. database: Cake's database session handling is used (see Chapter "The Cake Session Component" for more details).

CAKE_SESSION_STRING: a random string used in session mangement.

CAKE_SESSION_TABLE: the name of the table for storing session data (if CAKE_SESSION_SAVE == 'database'). Do not include a prefix here if one has already been specified for the default database connection.

CAKE_SESSION_TIMEOUT: number of seconds until session timeout. This figure is multiplied by CAKE_SECURITY.

COMPRESS_CSS: if set to true, CSS style sheets are compressed on output. This requires a /var/cache directory writable by the webserver. To use, reference your style sheets using /ccss (rather than /css) or use Controller::cssTag().

DEBUG: defines the level of error reporting and debug output the CakePHP application will render. Can be set to an integer from 0 to 3.

1. 0: Production mode. No error output, no debug messages shown.
2. 1: Development mode. Warnings and errors shown, along with debug messages.
3. 2: Same as in 1, but with SQL output.
4. 3: Same as in 2, but with full dump of current object (usually the Controller).

LOG_ERROR: Error constant. Used for differentiating error logging and debugging. Currently PHP supports LOG_DEBUG.

MAX_MD5SIZE: The maximum size (in bytes) to perform an md5() hash upon.

WEBSERVICES: If set to true, Cake's bulit in webservices functionality is turned on.

Section 3

CakePHP Path Constants

APP: the path to the application's directory.

APP_DIR: the name of the current application's app directory.

APP_PATH: absolute path to the application's app directory.

CACHE: path to the cache files directory.

CAKE: path to the application's cake directory.

COMPONENTS: path to the application's components directory.

CONFIGS: path to the configuration files directory.

CONTROLLER_TESTS: path to the controller tests directory.

CONTROLLERS: path to the application's controllers.

CSS: path to the CSS files directory.

ELEMENTS: path to the elements directory.

HELPER_TESTS: path to the helper tests directory.

HELPERS: path to the helpers directory.

INFLECTIONS: path to the inflections directory (usually inside the configuration directory).

JS: path to the JavaScript files directory.

LAYOUTS: path to the layouts directory.

LIB_TESTS: path to the Cake Library tests directory.

LIBS: path to the Cake libs directory.

LOGS: path to the logs directory.

MODEL_TESTS: path to the model tests directory.

MODELS: path to the models directory.

SCRIPTS: path to the Cake scripts directory.

TESTS: path to the tests directory (parent for the models, controllers, etc. test directories)

TMP: path to the tmp directory.

VENDORS: path to the vendors directory.

VIEWS: path to the views directory.

Section 4

CakePHP Webroot Configuration Paths

CORE_PATH: path to the Cake core libraries.

WWW_ROOT: path to the application's webroot directory (usually in /cake/

CAKE_CORE_INCLUDE_PATH: path to the Cake core libraries.

ROOT: the name of the directory parent to the base index.php of CakePHP.

WEBROOT_DIR: the name of the application's webroot directory.

CakePHP: The Manual

Data Validation

Section 1

Data Validation

Creating custom validation rules can help to make sure the data in a Model conforms to the business rules of the application, such as passwords can only be eight characters long, user names can only have letters, etc.

The first step to data validation is creating the validation rules in the Model. To do that, use the Model::validate array in the Model definition, for example:

/app/models/user.php

```
<?php
class User extends AppModel
{
   var $name = 'User';

   var $validate = array(
      'login' => '/[a-z0-9\_\-]{3,}$/i',
      'password' => VALID_NOT_EMPTY,
      'email' => VALID_EMAIL,
      'born' => VALID_NUMBER
   );
}
?>
```

Validations are defined using Perl-compatibile regular expressions, some of which are pre-defined in /libs/validators.php. These are:

- VALID_NOT_EMPTY
- VALID_NUMBER
- VALID_EMAIL
- VALID_YEAR

If there are any validations present in the model definition (i.e. in the $validate array), they will be parsed and checked during saves (i.e. in the Model::save() method). To validate the data directly use the Model::validates() (returns false if data is incorrect) and Model::invalidFields() (which returns an array of error messages).

But usually the data is implicit in the controller code. The following example demonstrates how to create a form-handling action:

Form-handling Action in /app/controllers/blog_controller.php

```
<?php
class BlogController extends AppController {

   var $uses = array('Post');

   function add ()
   {
      if (empty($this->data))
      {
         $this->render();
      }
      else
```

```
      {
         if($this->Post->save($this->data))
         {
             //ok cool, the stuff is valid
         }
         else
         {
            //Danger, Will Robinson. Validation errors.
            $this->set('errorMessage', 'Please correct errors below.');
            $this->render();
         }
      }
   }
}
?>
```

The view used by this action can look like this:

The add form view in /app/views/blog/add.thtml

```
<h2>Add post to blog</h2>
<form action="<?php echo $html->url('/blog/add')?>" method="post">
    <div class="blog_add">
        <p>Title:
            <?php echo $html->input('Post/title', array('size'=>'40'))?>
            <?php echo $html->tagErrorMsg('Post/title', 'Title is required.')?>
        </p>
        <p>Body
            <?php echo $html->textarea('Post/body') ?>
            <?php echo $html->tagErrorMsg('Post/body', 'Body is required.')?>
        </p>
        <p><?=$html->submit('Save')?></p>
    </div>
</form>
```

The Controller::validates($model[, $model...]) is used to check any custom validation added in the model. The Controller::validationErrors() method returns any error messages thrown in the model so they can be displayed by tagErrorMsg() in the view.

If you'd like to perform some custom validation apart from the regex based Cake validation, you can use the invalidate() function of your model to flag a field as erroneous. Imagine that you wanted to show an error on a form when a user tries to create a username that already exists in the system. Because you can't just ask Cake to find that out using regex, you'll need to do your own validation, and flag the field as invalid to invoke Cake's normal form invalidation process.

The controller might look something like this:

```
<?php

class UsersController extends AppController
{
    function create()
    {
        // Check to see if form data has been submitted
        if (!empty($this->data['User']))
        {
            //See if a user with that username exists
            $user = $this->User->findByUsername($this->data['User']['username']);

            // Invalidate the field to trigger the HTML Helper's error messages
            if (!empty($user['User']['username']))
            {
                $this->User->invalidate('username');//populates tagErrorMsg('User/username')
            }

            //Try to save as normal, shouldn't work if the field was invalidated.
```

```
            if($this->User->save($this->data))
            {
                $this->redirect('/users/index/saved');
            }
            else
            {
                 $this->render();
            }
        }
    }
}

?>
```

CakePHP: The Manual

Plugins

CakePHP allows you to set up a combination of controllers, models, and views and release them as a packaged application plugin that others can use in their CakePHP applications. Have a sweet user management module, simple blog, or web services module in one of your applications? Package it as a CakePHP plugin so you can pop it into other applications.

The main tie between a plugin and the application it has been installed into is the application's configuration (database connection, etc.). Otherwise, it operates in its own little space, behaving much like it would if it were an application on it's own.

Section 1

Creating a Plugin

As a working example, let's create a new plugin that orders pizza for you. What could be more useful in any CakePHP application? To start out, we'll need to place our plugin files inside the **/app/plugins** folder. The name of the parent folder for all the plugin files is important, and will be used in many places, so pick wisely. For this plugin, let's use the name 'pizza'. This is how the setup will eventually look:

Pizza Ordering Filesystem Layout

```
/app
    /plugins
        /pizza
            /controllers                <- plugin controllers go here
            /models                     <- plugin models go here
            /views                      <- plugin views go here
            /pizza_app_controller.php   <- plugin's AppController, named after the plugin
            /pizza_app_model.php        <- plugin's AppModel, named after the plugin
```

While defining an AppController and AppModel for any normal application is not required, **defining them for plugins is**. You'll need to create them before your plugin works. These two special classes are named after the plugin, and extend the parent application's AppController and AppModel. Here's what they should look like:

Pizza Plugin AppController: /app/plugins/pizza_app_controller.php

```
<?php

class PizzaAppController extends AppController
{
    //...
}

?>
```

Pizza Plugin AppModel: /app/plugins/pizza_app_model.php

```
<?php

class PizzaAppModel extends AppModel
{
    //...
}

?>
```

If you forget to define these special classes, CakePHP will hand you "Missing Controller" errors until the problem is rectified.

Section 2

Plugin Controllers

Controllers for our pizza plugin will be stored in **/app/plugins/pizza/controllers**. Since the main thing we'll be tracking is pizza orders, we'll need an OrdersController for this plugin.

While it isn't required, it is recommended that you name your plugin controllers something relatively unique in order to avoid namespace conflicts with parent applications. Its not a stretch to think that a parent application might have a UsersController, OrderController, or ProductController: so you might want to be creative with controller names, or prepend the name of the plugin to the classname (PizzaOrdersController, in this case).

So, we place our new PizzaOrdersController in **/app/plugins/pizza/controllers** and it looks like so:

/app/plugins/pizza/controllers/pizza_orders_controller.php

```
<?php

class PizzaOrdersController extends PizzaAppController
{
    var $name = 'PizzaOrders';

    function index()
    {
        //...
    }

    function placeOrder()
    {
        //...
    }
}

?>
```

Note how this controller extends the plugin's AppController (called PizzaAppController) rather than just the parent application's AppController.

Section 3

Plugin Models

Models for the plugin are stored in **/app/plugins/pizza/models**. We've already defined a PizzaOrdersController for this plugin, so let's create the model for that controller, called PizzaOrders (the classname PizzaOrders is consistent with our naming scheme, and is unique enough, so we'll leave it as is).

/app/plugins/pizza/models/pizza_order.php

```
<?php

class PizzaOrder extends PizzaAppModel
{
    var $name = 'PizzaOrder';
}

?>
```

Again, note that this class extends PizzaAppModel rather than AppModel.

Section 4

Plugin Views

Views behave exactly as they do in normal applications. Just place them in the right folder inside of the /app/plugins/[plugin]/views folder. For our pizza ordering plugin, we'll need at least one view for our PizzaOrdersController::index() action, so let's include that as well:

/app/plugins/pizza/views/pizza_orders/index.thtml

```
<h1>Order A Pizza</h1>
<p>Nothing goes better with Cake than a good pizza!</p>
<!-- An order form of some sort might go here....-->
```

Section 5

Working With Plugins

So, now that you've built evertything, it should be ready to distribute (though we'd suggest you also distribute a few extras like a readme, sql file, etc.).

Once a plugin as been installed in /app/plugins, you can access it at the URL /pluginname/controllername/action. In our pizza ordering plugin example, we'd access our PizzaOrdersController at /pizza/pizzaOrders.

Some final tips on working with plugins in your CakePHP applications:

1. When you don't have a [Plugin]AppController and [Plugin]AppModel, you'll get missing Controller errors when trying to access a plugin controller.
2. You can have a default controller with the name of your plugin. If you do that, you can access it via /[plugin]/action. For example, a plugin named 'users' with a controller named UsersController can be accessed at /users/add if there is no plugin called AddController in your [plugin]/controllers folder.
3. Plugins will use the layouts from the **/app/views/layouts** folder by default.
4. You can do inter-plugin communication by using

   ```
   $this->requestAction('/plugin/controller/action');
   ```
   in your controllers.
5. If you use requestAction, make sure controller and model names are as unique as possible. Otherwise you might get PHP "redefined class ..." errors.

Many thanks to Felix Geisendorfer (the_undefined) for the initial material for this chapter.

CakePHP: The Manual

Access Control Lists

Section 1

Understanding How ACL Works

Most important, powerful things require some sort of access control. Access control lists are a way to manage application permissions in a fine-grained, yet easily maintainable and manageable way. Access control lists, or ACL, handle two main things: things that want stuff, and things that are wanted. In ACL lingo, things (most often users) that want to use stuff are called access request objects, or AROs. Things in the system that are wanted (most often actions or data) are called access control objects, or ACOs. The entities are called 'objects' because sometimes the requesting object isn't a person - sometimes you might want to limit the access certain Cake controllers have to initiate logic in other parts of your application. ACOs could be anything you want to control, from a controller action, to a web service, to a line on your grandma's online diary.

To use all the acronyms at once: ACL is what is used to decide when an ARO can have access to an ACO.

Now, in order to help you understand this, let's use a practial example. Imagine, for a moment, a computer system used by a group of adventurers. The leader of the group wants to forge ahead on their quest while maintaining a healthy amount of privacy and security for the other members of the party. The AROs involved are as following:

```
Gandalf
Aragorn
Bilbo
Frodo
Gollum
Legolas
Gimli
Pippin
Merry
```

These are the entities in the system that will be requesting things (the ACOs) from the system. It should be noted that ACL is *not* a system that is meant to authenticate users. You should already have a way to store user information and be able to verify that user's identity when they enter the system. Once you know who a user is, that's where ACL really shines. Okay - back to our adventure.

The next thing Gandalf needs to do is make an initial list of things, or ACOs, the system will handle. His list might look something like:

```
Weapons
The One Ring
Salted Pork
Diplomacy
Ale
```

Traditionally, systems were managed using a sort of matrix, that showed a basic set of users and permissions relating to objects. If this information were stored in a table, it might look like the following, with X's indicating denied access, and O's indicating allowed access.

```
         Weapons    The One Ring    Salted Pork      Diplomacy       Ale
Gandalf     X             X             O               O            O
Aragorn     O             X             O               O            O
Bilbo       X             X             X               X            O
Frodo       X             O             X               X            O
Gollum      X             X             O               X            X
```

```
Legolas     O           X           O           O           O
Gimli       O           X           O           X           X
Pippin      X           X           X           O           O
Merry       X           X           X           X           O
```

At first glance, it seems that this sort of system could work rather well. Assignments can be made to protect security (only Frodo can access the ring) and protect against accidents (keeping the hobbits out of the salted pork). It seems fine grained enough, and easy enough to read, right?

For a small system like this, maybe a matrix setup would work. But for a growing system, or a system with a large amount of resources (ACOs) and users (AROs), a table can become unwieldy rather quickly. Imagine trying to control access to the hundreds of war encampments and trying to manage them by unit, for example. Another drawback to matrices is that you can't really logically group sections of users, or make cascading permissions changes to groups of users based on those logical groupings. For example, it would sure be nice to automatically allow the hobbits access to the ale and pork once the battle is over: Doing it on an indivudual user basis would be tedious and error prone, while making a cascading permissions change to all 'hobbits' would be easy.

ACL is most usually implemented in a tree structure. There is usually a tree of AROs and a tree of ACOs. By organizing your objects in trees, permissions can still be dealt out in a granular fashion, while still maintaining a good grip on the big picture. Being the wise leader he is, Gandalf elects to use ACL in his new system, and organizes his objects along the following lines:

```
Fellowship of the Ring:
Warriors
    Aragorn
    Legolas
    Gimli
Wizards
    Gandalf
Hobbits
    Frodo
    Bilbo
    Merry
    Pippin
Vistors
    Gollum
```

By structuring our party this way, we can define access controls to the tree, and apply those permissions to any children. The default permission is to deny access to everything. As you work your way down the tree, you pick up permissions and apply them. The last permission applied (that applies to the ACO you're wondering about) is the one you keep. So, using our ARO tree, Gandalf can hang on a few permissions:

```
Fellowship of the Ring: [Deny: ALL]
    Warriors                [Allow: Weapons, Ale, Elven Rations, Salted Pork]
        Aragorn
        Legolas
        Gimli
    Wizards                 [Allow: Salted Pork, Diplomacy, Ale]
        Gandalf
    Hobbits                 [Allow: Ale]
        Frodo
        Bilbo
        Merry
        Pippin
    Vistors                 [Allow: Salted Pork]
        Gollum
```

If we wanted to use ACL to see if the Pippin was allowed to access the ale, we'd first get his path in the tree, which is Fellowship->Hobbits->Pippin. Then we see the different permissions that reside at each of those points, and use the most specific permission relating to Pippin and the Ale.

1. Fellowship = DENY Ale, so deny (because it is set to deny all ACOs)

2. Hobbits = ALLOW Ale, so allow
3. Pippin = ?; There really isn't any ale-specific information so we stick with ALLOW.
4. Final Result: allow the ale.

The tree also allows us to make finer adjustments for more granular control - while still keeping the ability to make sweeping changes to groups of AROs:

```
Fellowship of the Ring: [Deny: ALL]
    Warriors                [Allow: Weapons, Ale, Elven Rations, Salted Pork]
        Aragorn             [Allow: Diplomacy]
        Legolas
        Gimli
    Wizards                 [Allow: Salted Pork, Diplomacy, Ale]
        Gandalf
    Hobbits                 [Allow: Ale]
        Frodo               [Allow: Ring]
        Bilbo
        Merry               [Deny: Ale]
        Pippin              [Allow: Diplomacy]
    Vistors                 [Allow: Salted Pork]
        Gollum
```

You can see this because the Aragorn ARO maintains is permissions just like others in the Warriors ARO group, but you can still make fine-tuned adjustments and special cases when you see fit. Again, permissions default to DENY, and only change as the traversal down the tree forces an ALLOW. To see if Merry can access the Ale, we'd find his path in the tree: Fellowship->Hobbits->Merry and work our way down, keeping track of ale-related permissions:

1. Fellowship = DENY (because it is set to deny all), so deny the ale.
2. Hobbits = ALLOW: ale, so allow the ale
3. Merry = DENY ale, so deny the ale
4. Final Result: deny the ale.

Section 2

Defining Permissions: Cake's INI-based ACL

Cake's first ACL implementation was based off of INI files stored in the Cake installation. While its useful and stable, we recommend that you use the database backed ACL solution, mostly because of its ability to create new ACOs and AROs on the fly. We meant it for usage in simple applications - and especially for those folks who might not be using a database for some reason.

ARO/ACO permissions are specified in **/app/config/acl.ini.php**. Instructions on specifying access can be found at the beginning of acl.ini.php:

```
; acl.ini.php - Cake ACL Configuration
; ---------------------------------------------------------------------
; Use this file to specify user permissions.
; aco = access control object (something in your application)
; aro = access request object (something requesting access)
;
; User records are added as follows:
;
; [uid]
; groups = group1, group2, group3
; allow = aco1, aco2, aco3
```

```
; deny = aco4, aco5, aco6
;
; Group records are added in a similar manner:
;
; [gid]
; allow = aco1, aco2, aco3
; deny = aco4, aco5, aco6
;
; The allow, deny, and groups sections are all optional.
; NOTE: groups names *cannot* ever be the same as usernames!
```

Using the INI file, you can specify users (AROs), the group(s) they belong to, and their own personal permissions. You can also specify groups along with their permissions. To learn how to use Cake's ACL component to check permissions using this INI file, see section 11.4.

Section 3

Defining Permissions: Cake's Database ACL

Getting Started

The default ACL permissions implementation is database stored. Database ACL, or dbACL consists of a set of core models, and a command-line script that comes with your Cake installation. The models are used by Cake to interact with your database in order to store and retrieve nodes the ACL trees. The command-line script is used to help you get started and be able to interact with your trees.

To get started, first you'll need to make sure your **/app/config/database.php** is present and correctly configured. For a new Cake installation, the easiest way to tell that this is so is to bring up the installation directory using a web browser. Near the top of the page, you should see the messages "Your database configuration file is present." and "Cake is able to connect to the database." if you've done it correctly. See section 4.1 for more information on database configuration.

Next, use the the ACL command-line script to initialize your database to store ACL information. The script found at /cake/scripts/acl.php will help you accomplish this. Initialize the your database for ACL by executing the following command (from your **/cake/scripts/** directory):

Initializing your database using acl.php

```
$ php acl.php initdb

Initializing Database...
Creating access control objects table (acos)...
Creating access request objects table (acos)...
Creating relationships table (aros_acos)...

Done.
```

At this point, you should be able to check your project's database to see the new tables. If you're curious about how Cake stores tree information in these tables, read up on modified database tree traversal. Basically, it stores nodes, and their place in the tree. The acos and aros tables store the nodes for their respective trees, and the aros_acos table is used to link your AROs to the ACOs they can access.

Now, you should be able to start creating your ARO and ACO trees.

Creating Access Request Objects (AROs) and Access Control Objects (ACOs)

There are two ways of referring to AROs/ACOs. One is by giving them an numeric id, which is usually just the primary key of the table they belong to. The other way is by giving them a string alias. The two are not mutually

exclusive.

The way to create a new ARO is by using the methods defined the the Aro Cake model. The create() method of the Aro class takes three parameters: $link_id, $parent_id, and $alias. This method creates a new ACL object under the parent specified by a parent_id - or as a root object if the $parent_id passed is null. The $link_id allows you to link a current user object to Cake's ACL structures. The alias parameter allows you address your object using a non-integer ID.

Before we can create our ACOs and AROs, we'll need to load up those classes. The easiest way to do this is to include Cake's ACL Component in your controller using the $components array:

```
var $components = array('Acl');
```

Once we've got that done, let's see what some examples of creating these objects might look like. The following code could be placed in a controller action somewhere:

```
$aro = new Aro();

// First, set up a few AROs.
// These objects will have no parent initially.

$aro->create( 1, null, 'Bob Marley' );
$aro->create( 2, null, 'Jimi Hendrix');
$aro->create( 3, null, 'George Washington');
$aro->create( 4, null, 'Abraham Lincoln');

// Now, we can make groups to organize these users:
// Notice that the IDs for these objects are 0, because
//     they will never tie to users in our system

$aro->create(0, null, 'Presidents');
$aro->create(0, null, 'Artists');

//Now, hook AROs to their respective groups:

$aro->setParent('Presidents', 'George Washington');
$aro->setParent('Presidents', 'Abraham Lincoln');
$aro->setParent('Artists', 'Jimi Hendrix');
$aro->setParent('Artists', 'Bob Marley');

//In short, here is how to create an ARO:
$aro = new Aro();
$aro->create($user_id, $parent_id, $alias);
```

You can also create AROs using the command line script using $acl.php create aro <link_id> <parent_id> <alias>.

Creating an ACO is done in a similar manner:

```
$aco = new Aco();

//Create some access control objects:
$aco->create(1, null, 'Electric Guitar');
$aco->create(2, null, 'United States Army');
$aco->create(3, null, 'Fans');

// I suppose we could create groups for these
// objects using setParent(), but we'll skip that
// for this particular example

//So, to create an ACO:
$aco = new Aco();
$aco->create($id, $parent, $alias);
```

The corresponding command line script command would be: $acl.php create aco <link_id> <parent_id> <alias>.

Assigning Permissions

After creating our ACOs and AROs, we can finally assign permission between the two groups. This is done using Cake's core Acl component. Let's continue on with our example:

```
// First, in a controller, we'll need access
// to Cake's ACL component:

class SomethingsController extends AppController
{
    // You might want to place this in the AppController
    // instead, but here works great too.

    var $components = array('Acl');

    // Remember: ACL will always deny something
    // it doesn't have information on. If any
    // checks were made on anything, it would
    // be denied. Let's allow an ARO access to an ACO.

    function someAction()
    {
        //ALLOW

        // Here is how you grant an ARO full access to an ACO
        $this->Acl->allow('Jimi Hendrix', 'Electric Guitar');
        $this->Acl->allow('Bob Marley',   'Electric Guitar');

        // We can also assign permissions to groups, remember?
        $this->Acl->Allow('Presidents', 'United States Army');

        // The allow() method has a third parameter, $action.
        // You can specify partial access using this parameter.
        // $action can be set to create, read, update or delete.
        // If no action is specified, full access is assumed.

        // Look, don't touch, gentlemen:
        $this->Acl->allow('George Washington', 'Electric Guitar', 'read');
        $this->Acl->allow('Abraham Lincoln',   'Electric Guitar', 'read');

        //DENY

        //Denies work in the same manner:

        //When his term is up...
        $this->Acl->deny('Abraham Lincoln', 'United States Army');


    }
}
```

This particular controller isn't especially useful, but it is mostly meant to show you how the process works. Using the Acl component in connection with your user management controller would be the best usage. Once a user has been created on the system, her ARO could be created and placed at the right point of the tree, and permissions could be assigned to specific ACO or ACO groups based on her identity.

Permissions can also be assigned using the command line script packaged with Cake. The syntax is similar to the model functions, and can be viewed by executing $php acl.php help.

Section 4

Checking Permissions: The ACL Component

Checking permissions is the easiest part of using Cake's ACL: it consists of using a single method in the Acl

component: check(). A good way to implement ACL in your application might be to place an action in your AppController that performs ACL checks. Once placed there, you can access the Acl component and perform permissions checks application-wide. Here's an example implementation:

```
class AppController extends Controller
{
    // Get our component
    var $components = array('Acl');

    function checkAccess($aco)
    {
        // Check access using the component:
        $access = $this->Acl->check($this->Session->read('user_alias'), $aco, $action = "*");

        //access denied
        if ($access === false)
        {
            echo "access denied";
            exit;
        }
        //access allowed
        else
        {
            echo "access allowed";
            exit;
        }
    }
}
```

Basically, by making the Acl component available in the AppController, it will be visible for use in any controller in your application.

```
// Here's the basic format:
$this->Acl->Check($aro, $aco, $action = '*');
```

CakePHP: The Manual

Data Sanitization

Section 1

Using Sanitize in Your Application

Cake comes with Sanitize, a class you can use to rid user-submitted data of malicious attacks and other unwanted data. Sanitize is a core library, so it can be used anywhere inside of your code, but is probably best used in controllers or models.

```
// First, include the core library:

uses('sanitize');

// Next, create a new Sanitize object:

$mrClean = new Sanitize();

// From here, you can use Sanitize to clean your data
// (These methods explained in the next section)
```

Section 2

Making Data Safe for use in SQL and HTML

This section explains how to use some of the functions that Sanitize offers.

- **paranoid**
- string *$string*
- array *$allowedChars*

This function strips anything out of the target $string that is not a plain-jane alphanumeric character. You can, however, let it overlook certain characters by passing them along inside the $allowed array.

```
$badString = ";:<script><html><   // >@@#";

echo $mrClean->paranoid($badString);

// output: scripthtml

echo $mrClean->paranoid($badString, array(' ', '@'));

// output: scripthtml    @@
```

- **html**
- string *$string*
- boolean *$remove = false*

This method helps you get user submitted data ready for display inside an existing HTML layout. This is especially useful if you don't want users to be able to break your layouts or insert images or scripts inside of blog comments, forum posts, and the like. If the $remove option is set to true, any HTML is removed rather than rendered as HTML entities.

```
$badString = '<font size="99" color="#FF0000">HEY</font><script>...</script>';

echo $mrClean->html($badString);

// output: &lt;font size=&quot;99&quot;
color=&quot;#FF0000&quot;&gt;HEY&lt;/font&gt;&lt;script&gt;...&lt;/script&gt;

echo $mrClean->html($badString, true);
```

```
// output: font size=99 color=#FF0000 HEY fontscript...script
```

- **sql**
- string *$string*

Used to escape SQL statements by adding slashes, depending on the system's current magic_quotes_gpc setting.

- **cleanArray**
- array *@$dirtyArray*

This function is an industrial strength, multi-purpose cleaner, meant to be used on entire arrays (like $this->params['form'], for example). The function takes an array and cleans it: nothing is returned because the array is passed by reference. The following cleaning operations are performed on each element in the array (recursively):

1. Odd spaces (including 0xCA) are replaced with regular spaces.
2. HTML is replaced by its corresponding HTML entities (including \n to
).
3. Double-check special chars and remove carriage returns for increased SQL security.
4. Add slashes for SQL (just calls the sql function outlined above).
5. Swap user-inputted backslashes with trusted backslashes.

CakePHP: The Manual

The Cake Session Component

Section 1

Cake Session Storage Options

Cake comes preset to save session data in three ways: as temporary files inside of your Cake installation, using the default PHP mechanism, or serialized in a database. By default, Cake uses PHP's default settings. To override this in order to use temp files or a database, edit your core configuration file at **/app/config/core.php**. Change the CAKE_SESSION_SAVE constant to 'cake', 'php', or 'database', depending on your application's needs.

core.php Session Configuration

```
/**
 * CakePHP includes 3 types of session saves
 * database or file. Set this to your preferred method.
 * If you want to use your own save handler place it in
 * app/config/name.php DO NOT USE file or database as the name.
 * and use just the name portion below.
 *
 * Setting this to cake will save files to /cakedistro/tmp directory
 * Setting it to php will use the php default save path
 * Setting it to database will use the database
 *
 *
 */
define('CAKE_SESSION_SAVE', 'php');
```

In order to use the database session storage, you'll need to create a table in your database. The schema for that table can be found in **/app/config/sql/sessions.sql**.

Section 2

Using the Cake Session Component

The Cake session component is used to interact with session information. It includes basic session reading and writing, but also contains features for using sessions for error and reciept messages (i.e. "Your data has been saved"). The Session Component is available in all Cake controllers by default.

Here are some of the functions you'll use most:

- **check**
- string *$name*

Checks to see if the current key specified by $name has been set in the session.

- **del**
- string *$name*

- **delete**
- string *$name*

Deletes the session variable specified by $name.

- **error**

Returns the last error created by the CakeSession component. Mostly used for debugging.

- **flash**
- string *$key = 'flash'*

Returns the last message set in the session using setFlash(). If $key has been set, the message returned is the most recent stored under that key.

- **read**
- string *$name*

Returns the session variable specified by $name.

- **renew**

Renews the currently active session by creating a new session ID, deleting the old one, and passing the old session data to the new one.

- **setFlash**
- string *$flashMessage*
- string *$layout = 'default'*
- array *$params*
- string *$key = 'flash'*

Writes the message specified by $flashMessage into the session (to be later retrieved by flash()).

If $layout is set to 'default', the message is stored as ' '.$flashMessage.' '. If $default is set to '', the message is stored just as it has been passed. If any other value is passed, the message is stored inside the Cake view specified by $layout.

Params has been placed in this function for future usage. Check back for more info.

The $key variable allows you to store flash messages under keys. See flash() for retreiving a flash message based off of a key.

- **valid**

Returns true if the session is valid. Best used before read() operations to make sure that the session data you are trying to access is in fact valid.

- **write**
- string *$name*
- mixed *$value*

Writes the variable specified by $name and $value into the active session.

CakePHP: The Manual

The Request Handler Component

Section 1

Introduction

The Request Handler component is used in Cake to determine information about the incoming HTTP request. You can use it to better inform your controller about AJAX requests, get information about the remote client's IP address and request type, or strip unwanted data from output. To use the Request Handler component, you'll need to make sure it is specified in your controller's $components array.

```
class ThingsController extends AppController
{
    var $components = array('RequestHandler');

    // ...
}
```

Section 2

Getting Client/Request Information

Let's just dive in:

- **accepts**
- string *$type*

Returns information about the content-types that the client accepts, depending on the value of $type. If null or no value is passed, it will return an array of content-types the client accepts. If a string is passed, it returns true if the client accepts the given type, by checking $type against the content-type map (see setContent()). If $type is an array, each string is evaluated individually, and accepts() will return true if just one of them matches an accepted content-type. For example:

```
class PostsController extends AppController
{
    var $components = array('RequestHandler');

    function beforeFilter ()
    {
        if ($this->RequestHandler->accepts('html'))
        {
            // Execute code only if client accepts an HTML (text/html) response
        }
        elseif ($this->RequestHandler->accepts('rss'))
        {
            // Execute RSS-only code
        }
        elseif ($this->RequestHandler->accepts('atom'))
        {
            // Execute Atom-only code
        }
        elseif ($this->RequestHandler->accepts('xml'))
        {
            // Execute XML-only code
        }

        if ($this->RequestHandler->accepts(array('xml', 'rss', 'atom')))
        {
```

```
            // Executes if the client accepts any of the above: XML, RSS or Atom
        }
    }
}
```

- **getAjaxVersion**

If you are using the Prototype JS libraries, you can fetch a special header it sets on AJAX requests. This function returns the Prototype version used.

- **getClientIP**

Returns the remote client's IP address.

- **getReferrer**

Returns the server name from which the request originated.

- **isAjax**

Returns true if the current request was an XMLHttpRequest.

- **isAtom**

Returns true if the client accepts Atom feed content (application/atom+xml).

- **isDelete**

Returns true if the current request was via DELETE.

- **isGet**

Returns true if the current request was via GET.

- **isMobile**

Returns true if the user agent string matches a mobile web browser.

- **isPost**

Returns true if the current request was via POST.

- **isPut**

Returns true if the current request was via PUT.

- **isRss**

Returns true if the clients accepts RSS feed content (application/rss+xml).

- **isXml**

Returns true if the client accepts XML content (application/xml or text/xml).

- **setContent**
- string *$name*
- string *$type*

Adds a content-type alias mapping, for use with accepts() and prefers(), where $name is the name of the mapping (string), and $type is either a string or an array of strings, each of which is a MIME type. The built-in type mappings are as follows:

```
// Name     => Type
  'js'      => 'text/javascript',
  'css'     => 'text/css',
  'html'    => 'text/html',
  'form'    => 'application/x-www-form-urlencoded',
  'file'    => 'multipart/form-data',
  'xhtml'   => array('application/xhtml+xml', 'application/xhtml', 'text/xhtml'),
  'xml'     => array('application/xml', 'text/xml'),
  'rss'     => 'application/rss+xml',
  'atom'    => 'application/atom+xml'
```

Section 3

Stripping Data

Occasionally you will want to remove data from a request or output. Use the following Request Handler functions to perform these sorts of operations.

- **stripAll**
- string *$str*

Strips the white space, images, and scripts from $str (using stripWhitespace(), stripImages(), and stripScripts()).

- **stripImages**
- string *$str*

Strips any HTML embedded images from $str.

- **stripScripts**
- string *$str*

Strips any <script> and <style> related tags from $str.

- **stripTags**
- string *$str*
- string *$tag1*
- string *$tag2...*

Removes the tags specified by $tag1, $tag2, etc. from $str.

```
$someString = '<font color="#FF0000"><bold>Foo</bold></font> <em>Bar</em>';

echo $this->RequestHandler->stripTags($someString, 'font', 'bold');

// output: Foo <em>Bar</em>
```

- **stripWhiteSpace**
- string *$str*

Strips whitespace from $str.

Section 4

Other Useful Functions

The Request Handler component is especially useful when your application includes AJAX requests. The setAjax() function is used to automatically detect AJAX requests, and set the controller's layout to an AJAX layout for that request. The benefit here is that you can make small modular views that can also double as AJAX views.

```
// list.thtml
<ul>
<? foreach ($things as $thing):?>
<li><?php echo $thing;?></li>
<?endforeach;?>
</ul>

//-------------------------------------------------------------

//The list action of my ThingsController:
function list()
{
    $this->RequestHandler->setAjax($this);
    $this->set('things', $this->Thing->findAll());
}
```

When a normal browser request is made to /things/list, the unordered list is rendered inside of the default layout for the application. If the URL is requested as part of an AJAX operation, the list is automatically rendered in the bare AJAX layout.

CakePHP: The Manual

The Security Component

Section 1

Introduction

The Security component is used to secure your controller actions against malicious or errant requests. It allows you to set up the conditions under which an action can be requested, and optionally specify how to deal with requests that don't meet those requirements. Again, before using the Security component, you must make sure that 'Security' is listed in your controllers' $components array.

Section 2

Protecting Controller Actions

The Security component contains two primary methods for restricting access to controller actions:

- **requirePost**
- string *$action1*
- string *$action2*
- string *$action3...*

In order for the specified actions to execute, they must be requested via POST.

- **requireAuth**
- string *$action1*
- string *$action2*
- string *$action3...*

Ensures that a request is coming from within the application by checking an authentication key in the POST'ed form data against an authentication key stored in the user's session. If they match, the action is allowed to execute. Be aware, however, that for reasons of flexibility, this check is only run if form data has actually been posted. If the action is called with a regular GET request, requireAuth() will do nothing. For maximum security, you should use requirePost() and requireAuth() on actions that you want fully protected. Learn more about how the authentication key is generated, and how it ends up where it should in Section 4 below.

But first, let's take a look at a simple example:

```
class ThingsController extends AppController
{
    var $components = array('Security');

    function beforeFilter()
    {
        $this->Security->requirePost('delete');
    }

    function delete($id)
    {
        // This will only happen if the action is called via an HTTP POST request
```

```
        $this->Thing->del($id);
    }
}
```

Here, we're telling the Security component that the 'delete' action requires a POST request. The beforeFilter() method is usually where you'll want to tell Security (and most other components) what to do with themselves. It will then go do what it's told right after beforeFilter() is called, but right before the action itself is called.

And that's about all there is to it. You can test this by typing the URL for the action into your browser and seeing what happens.

Section 3

Handling Invalid Requests

So if a request doesn't meet the security requirements that we define, what happens to it? By default, the request is black-holed, which means that the client is sent a 404 header, and the application immediately exits. However, the Security component has a $blackHoleCallback property, which you can set to the name of a custom callback function defined in your controller.

Rather than simply give a 404 header and then nothing, this property allows you to perform some additional checking on the request, redirect the request to another location, or even log the IP address of the offending client. However, if you choose to use a custom callback, it is your responsibility to exit the application if the request is invalid. If your callback returns true, then the Security component will continue to check the request against other defined requirements. Otherwise, it stops checking, and your application continues uninhibited.

Section 4

Advanced Request Authentication

The requireAuth() method allows you to be very detailed in specifying how and from where an action can be accessed, but it comes with certain usage stipulations, which become clear when you understand how this method of authentication works. As stated above, requireAuth() works by comparing an authentication key in the POST data to the key stored in the user's session data. Therefore, the Security component must be included in both the controller recieveing the request, as well as the controller making the request.

For example, if I have an action in PostsController with a view containing a form that POSTs to an action in CommentsController, then the Security component must be included in both CommentsController (which is receiving the request, and actually protecting the action), as well as PostsController (from which the request will be made).

Every time the Security component is loaded, even if it is not being used to protect an action, it does the following things: First, it generates an authentication key using the core Security class. Then, it writes this key to the session, along with an expiration date and some additional information (the expiration date is determined by your session security setting in **/app/config/core.php**). Next, it sets the key in your controller, to be referenced later.

Then in your view files, any form tag you generate using $html->formTag() will also contain a hidden input field with the authentication key. That way, when the form is POSTed, the Security component can compare that value to the value in the session on the receiving end of the request. After that, the authentication key is regenerated, and the session is updated for the next request.

CakePHP: The Manual

View Caching

Section 1

What is it ?

Since 0.10.9.2378_final, Cake has support for view caching (also called Full Page Caching). No, we are not kidding. You can now cache your layouts and views. You can also mark parts of your views to be ignored by the caching mechanism. The feature, when used wisely, can increase the speed of your app by a considerable amount.

When you request a URL, Cake first looks to see if the requested URL isn't already cached. If it is, Cake bypasses the dispatcher and returns the already rendered, cached version of the page. If the page isn't in the cache, Cake behaves normally.

If you've activated Cake's caching features, Cake will store the output of its normal operation in the cache for future user. The next time the page is requested, Cake will fetch it from the cache. Neat, eh? Let's dig in to see how it works.

Section 2

How Does it Work ?

Activating the cache

By default, view caching is disabled. To activate it, you first need to change the value of **CACHE_CHECK** in **/app/config/core.php** from false to true:

/app/config/core.php (partial)

```
define ('CACHE_CHECK', true);
```

This line tells Cake that you want to enable View Caching.

In the controller for the views you want to cache you have to add the **Cache** helper to the helpers array:

```
var $helpers = array('Cache');
```

Next, you'll need to specify what you want to cache.

The $cacheAction Controller Variable

In this section, we will show you how to tell Cake what to cache. This is done by setting a controller variable called $cacheAction. The $cacheAction variable should be set to an array that contains the actions you want to be cached, and the time (in seconds) you want the cache to keep its data. The time value can also be a strtotime() friendly string (i.e. '1 day' or '60 seconds').

Let's say we had a ProductsController, with some things that we'd like to cache. The following examples show how to use $cacheAction to tell Cake to cache certain parts of the controller's actions.

$cacheAction Examples

```
//Cache a few of the most oft visited product pages for six hours:
var $cacheAction = array(
    'view/23/'  => 21600,
    'view/48/'  => 21600
);

//Cache an entire action. In this case the recalled product list, for one day:
var $cacheAction = array('recalled/' => 86400);

//If we wanted to, we could cache every action by setting it to a string:
//that is strtotime() friendly to indicate the caching time.
var $cacheAction = "1 hour";

//You can also define caching in the actions using $this->cacheAction = array()...
```

Marking Content in the View

There are instances where you might not want parts of a view cached. If you've got some sort of element highlighting new products, or something similar, you might want to tell Cake to cache the view... except for a small part.

You can tell Cake not to cache content in your views by wrapping <cake:nocache> </cake:nocache> tags around content you want the caching engine to skip.

<cake:nocache> example

```
<h1> New Products! </h1>
<cake:nocache>
<ul>
<?php foreach($newProducts as $product): ?>
<li>$product['name']</li>
<?endforeach;?>
</ul>
</cake:nocache>
```

Clearing the cache

First, you should be aware that Cake will automatically clear the cache if a database change has been made. For example, if one of your views uses information from your Post model, and there has been an INSERT, UPDATE, or DELETE made to a Post, Cake will clear the cache for that view.

But there may be cases where you'll want to clear specific cache files yourself. To do that Cake provides the clearCache function, which is globally available:

<cake:nocache> example

```
//Remove all cached pages that have the controller name.
clearCache('controller');

//Remove all cached pages that have the controller_action name.
clearCache('controller_action/');

//Remove all cached pages that have the controller_action_params name.
//Note: you can have multiple params
clearCache('controller_action_params');

//You can also use an array to clear muliple caches at once.
clearCache(array('controller_action_params','controller2_action_params));
```

Section 3

Things To Remember

Below are a few things to remember about View Caching:

1. To enable cache you set **CACHE_CHECK** to true in **/app/config/core.php** .
2. In the controller for the views you want to cache you have to add the **Cache** helper to the helpers array.
3. To cache certain URLs, use **$cacheAction** in the controller.
4. To stop certain parts of a view from been cached you wrap them with **<cake:nocache> </cake:nocache>**
5. Cake automatically clears specific cache copies when changes are made to the database
6. To manually clear parts of the cache, use **clearCache()**.

CakePHP: The Manual

The Cake Blog Tutorial

Section 1

Introduction

Welcome to Cake! You're probably checking out this tutorial because you want to learn more about how Cake works. Its our aim to increase productivity and make coding more enjoyable: we hope you'll see this as you dive into the code.

This tutorial will walk you through the creation of a simple blog application. We'll be getting and installing Cake, creating and configuring a database, and creating enough application logic to list, add, edit, and delete blog posts.

Here's what you'll need:

1. A running web server. We're going to assume you're using Apache, though the instructions for using other servers should be very similar. We might have to play a little with the server configuration, but most folks can get Cake up and running without any configuration at all.
2. A database server. We're going to be using mySQL in this tutorial. You'll need to know enough about SQL in order to create a database: Cake will be taking the reigns from there.
3. Basic PHP knowledge. The more object-oriented programming you've done, the better: but fear not if you're a procedural fan.
4. Finally, you'll need a basic knowledge of the MVC programming pattern. A quick overview can be found in Chapter "Basic Concepts", Section 2: The MVC Pattern. Don't worry: its only a half a page or so.

Let's get started!

Section 2

Getting Cake

First, let's get a copy of fresh Cake code. The most recent release as of this writing is CakePHP 1.0.1.2708 is the latest release.

To get a fresh download, visit the CakePHP project at Cakeforge: http://cakeforge.org/projects/cakephp/ and download the stable release.

You can also checkout/export a fresh copy of our trunk code at: https://svn.cakephp.org/repo/trunk/cake/1.x.x.x/

Regardless of how you downloaded it, place the code inside of your DocumentRoot. Once finished, your directory setup should look something like the following:

```
/path_to_document_root
    /app
    /cake
    /vendors
    .htaccess
    index.php
    VERSION.txt
```

Now might be a good time to learn a bit about how Cake's directory structure works: check out Chapter "Basic Concepts", Section 3: Overview of the Cake File Layout.

Section 3

Creating the Blog Database

Next, lets set up the underlying database for our blog. Right now, we'll just create a single table to store our posts. We'll also throw in a few posts right now to use for testing purposes. Execute the following SQL statements into your database:

```
/* First, create our posts table: */
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

/* Then insert some posts for testing: */
INSERT INTO posts (title,body,created)
    VALUES ('The title', 'This is the post body.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('A title once again', 'And the post body follows.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

The choices on table and column names are not arbitrary. If you follow Cake's database naming conventions, and Cake's class naming conventions (both outlined in Appendix "Cake Conventions"), you'll be able to take advantage of a lot of free functionality and avoid configuration. Cake is flexible enough to accomodate even the worst legacy database schema, but adhering to convention will save you time.

Check out Appendix "Cake Conventions"
more information, but suffice it to say that naming our table 'posts' automatically hooks it to our Post model, and having fields called 'modified' and 'created' will be automagically managed by Cake.

Section 4

Cake Database Configuration

Onward and upward: let's tell Cake where our database is and how to connect to it. This will be the first and last time you configure anything.

A copy of Cake's database configuration file is found in **/app/config/database.php.default**. Make a copy of this file in the same directory, but name it **database.php**.

The config file should be pretty straightforward: just replace the values in the $default array with those that apply to your setup. A sample completed configuration array might look something like the following:

```
var $default = array('driver'   => 'mysql',
                     'connect'  => 'mysql_pconnect',
                     'host'     => 'localhost',
                     'login'    => 'cakeBlog',
                     'password' => 'c4k3-rUl3Z',
                     'database' => 'cake_blog_tutorial' );
```

Once you've saved your new database.php file, you should be able to open your browser and see the Cake welcome page. It should also tell you that your database connection file was found, and that Cake can successfully connect to the database.

Section 5

A Note On mod_rewrite

Occasionally a new user will run in to mod_rewrite issues, so I'll mention them marginally here. If the Cake welcome page looks a little funny (no images or css styles), it probably means mod_rewrite isn't functioning on your system.

Here are some tips to help get you up and running:

1. Make sure that an .htaccess override is allowed: in your httpd.conf, you should have a section that defines a section for each Directory on your server. Make sure the **AllowOverride** is set to **All** for the correct Directory.
2. Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.conf.
3. For some reason or another, you might have obtained a copy of CakePHP without the needed .htaccess files. This sometimes happens because some operating systems treat files that start with '.' as hidden, and don't copy them. Make sure your copy of CakePHP is from the downloads section of the site or our SVN repository.
4. Make sure you are loading up mod_rewrite correctly! You should see something like **LoadModule rewrite_module libexec/httpd/mod_rewrite.so** and **AddModule mod_rewrite.c** in your httpd.conf.

If you don't want or can't get mod_rewrite (or some other compatible module) up and running on your server, you'll need to use Cake's built in pretty URLs. In **/app/config/core.php**, uncomment the line that looks like:

```
define ('BASE_URL', env('SCRIPT_NAME'));
```

This will make your URLs look like www.example.com/index.php/controllername/actionname/param rather than www.example.com/controllername/actionname/param.

Section 6

Create a Post Model

The model class is the bread and butter of CakePHP applications. By creating a Cake model that will interact with our database, we'll have the foundation in place needed to do our view, add, edit, and delete operations later.

Cake's model class files go in **/app/models**, and the file we will be creating will be saved to **/app/models/post.php**. The completed file should look like this:

/app/models/post.php

```
<?php

class Post extends AppModel
{
    var $name = 'Post';
}

?>
```

Because of the way the class and file are named, this tells Cake that you want a Post model available in your PostsController that is tied to a table in your default database called 'posts'.

The $name variable is always a good idea to add, and is used to overcome some class name oddness in PHP4.

For more on models, such as table prefixes, callbacks, and validation, check out Chapter "Models".

Section 7

Create a Posts Controller

Next we'll create a controller for our posts. The controller is where all the logic for post interaction will happen, and its also where all the actions for this model will be found. You should place this new controller in a file called **posts_controller.php** inside your **/app/controllers** directory. Here's what the basic controller should look like:

/app/controllers/posts_controller.php

```
<?php

class PostsController extends AppController
{
    var $name = 'Posts';
}

?>
```

Now, lets add an action to our controller. When users request www.example.com/posts, this is the same as requesting www.example.com/posts/index. Since we want our readers to view a list of posts when they access that URL, the index action would look something like this:

/app/controllers/posts_controller.php (index action added)

```
<?php

class PostsController extends AppController
{
    var $name = 'Posts';

    function index()
    {
        $this->set('posts', $this->Post->findAll());
    }
}

?>
```

Let me explain the action a bit. By defining function index() in our PostsController, users can now access the logic there by requesting www.example.com/posts/index. Similarly, if we were to define a function called foobar(), users would be able to access that at www.example.com/posts/foobar.

The single instruction in the action uses set() to pass data to the view (which we'll create next). The line sets the view variable called 'posts' equal to the return value of the findAll() method of the Post model. Our Post model is automatically available at $this->Post because we've followed Cake's naming conventions.

To learn more about Cake's controllers, check out Chapter "Controllers".

Section 8

Creating Post Views

Now that we have our database connected using our model, and our application logic and flow defined by our controller, let's create a view for the index action we defined above.

Cake views are just HTML and PHP flavored fragments that fit inside an application's layout. Layouts can be defined and switched between, but for now, let's just use the default.

Remember in the last section how we assigned the 'posts' variable to the view using the set() method? That would hand down data to the view that would look something like this:

```
// print_r($posts) output:

Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => The title
                    [body] => This is the post body.
                    [created] => 2006-03-08 14:42:22
                    [modified] =>
                )
```

```
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => A title once again
                    [body] => And the post body follows.
                    [created] => 2006-03-08 14:42:23
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Title strikes back
                    [body] => This is really exciting! Not.
                    [created] => 2006-03-08 14:42:24
                    [modified] =>
                )
        )
)
```

Cake's view files are stored in **/app/views**
inside a folder named after the controller they correspond to (we'll have to create a folder named 'posts' in this case). To format this post data in a nice table, our view code might look something like this:

/app/views/posts/index.thtml

```
<h1>Blog posts</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

   <!-- Here's where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $html->link($post['Post']['title'], "/posts/view/".$post['Post']['id']); ?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>

</table>
```

Hopefully this should look somewhat simple.

You might have noticed the use of an object called **$html**. This is an instance of the **HtmlHelper** class. Cake comes with a set of view 'helpers' that make things like linking, form output, JavaScript and Ajax a snap. You can learn more about how to use them in Chapter "Helpers", but what's important to note here is that the **link()** method will generate an HTML link with the given title (the first parameter) and URL (the second parameter).

When specifying URL's in Cake, you simply give a path relative to the base of the application, and Cake fills in the rest. As such, your URL's will typically take the form of **/controller/action/id**.

Now you should be able to point your browser to http://www.example.com/posts/index. You should see your view, correctly formatted with the title and table listing of the posts.

If you happened to have clicked on one of the links we created in this view (that link a post's title to a URL **/posts/view/some_id**), you were probably informed by Cake that the action hasn't yet been defined. If you were not so informed, either something has gone wrong, or you actually did define it already, in which case you are very sneaky.

Otherwise, we'll create it now:

/app/controllers/posts_controller.php (view action added)

```
<?php

class PostsController extends AppController
{

    var $name = 'Posts';

    function index()
    {
          $this->set('posts', $this->Post->findAll());
    }

    function view($id = null)
    {
        $this->Post->id = $id;
        $this->set('post', $this->Post->read());
    }
}

?>
```

The set() call should look familiar. Notice we're using read() rather than findAll() because we only really want a single post's information.

Notice that our view action takes a parameter. This parameter is handed to the action by the URL called. If a user requests /posts/view/3, then the value '3' is passed as $id.

Now let's create the view for our new 'view' action and place it in /app/views/posts/view.thtml.

/app/views/posts/view.thtml

```
<h1><?php echo $post['Post']['title']?></h1>

<p><small>Created: <?php echo $post['Post']['created']?></small></p>

<p><?php echo $post['Post']['body']?></p>
```

Verify that this is working by trying the links at /posts/index or manu ally requesting a post by accessing /posts/view/1.

Section 9

Adding Posts

reading from the database and showing us the posts is fine and dandy, but let's allow for the adding of new posts.

First, start with the add() action in the PostsController:

/app/controllers/posts_controller.php (add action added)

```
<?php

class PostsController extends AppController
{
    var $name = 'Posts';

    function index()
    {
         $this->set('posts', $this->Post->findAll());
    }

    function view($id)
    {
        $this->Post->id = $id;
```

```
        $this->set('post', $this->Post->read());

    }

    function add()
    {
        if (!empty($this->data))
        {
            if ($this->Post->save($this->data))
            {
                $this->flash('Your post has been saved.','/posts');
            }
        }
    }
}

?>
```

Let me read the add() action for you in plain English: if the form data isn't empty, try to save the post model using that data. If for some reason it doesn't save, give me the data validation errors and render the view showing those errors.

When a user uses a form to POST data to your application, that information is available in $this->params. You can pr() that out if you want to see what it looks like. $this->data is an alias for $this->params['data'].

The $this->flash() function called is a controller function that flashes a message to the user for a second (using the flash layout) then forwards the user on to another URL (/posts, in this case). If DEBUG is set to 0 $this->flash() will redirect automatically, however, if DEBUG > 0 then you will be able to see the flash layout and click on the message to handle the redirect.

Calling the save() method will check for validation errors and will not save if any occur. There are several methods available so you can check for validation errors, but we talk about the validateErrors() call in a bit, so keep that on the back burner for a moment while I show you what the view looks like when we move on to the section about data validation.

Section 10

Data Validation

Cake goes a long way in taking the monotony out of form input validation. Everyone hates coding up endless forms and their validation routines, and Cake makes it easier and faster.

To take advantage of the validation features, you'll need to use Cake's HtmlHelper in your views. The HtmlHelper is available by default to all views at $html.

Here's our add view:

/app/views/posts/add.thtml

```
<h1>Add Post</h1>
<form method="post" action="<?php echo $html->url('/posts/add')?>">
    <p>
        Title:
        <?php echo $html->input('Post/title', array('size' => '40'))?>
        <?php echo $html->tagErrorMsg('Post/title', 'Title is required.') ?>
    </p>
    <p>
        Body:
        <?php echo $html->textarea('Post/body', array('rows'=>'10')) ?>
        <?php echo $html->tagErrorMsg('Post/body', 'Body is required.') ?>
    </p>
    <p>
        <?php echo $html->submit('Save') ?>
    </p>
</form>
```

As with **$html->link()**, **$html->url()**

will generate a proper URL from the controller and action we have given it. By default, it prints out a POST form tag, but this can be modified by the second parameter. The **$html->input()** and **$html->textarea()** functions spit out form elements of the same name. The first parameter tells Cake which model/field they correspond to, and the second param is for extra HTML attributes (like the size of the input field). Again, refer to Chapter "Helpers" for more on helpers.

The **tagErrorMsg()** function calls will output the error messages in case there is a validation problem.

If you'd like, you can update your **/app/views/posts/index.thtml**
view to include a new "Add Post" link that points to www.example.com/posts/add.

That seems cool enough, but how do I tell Cake about my validation requirements? This is where we come back to the model.

/app/models/post.php (validation array added)

```
<?php

class Post extends AppModel
{
    var $name = 'Post';

    var $validate = array(

        'title'   => VALID_NOT_EMPTY,
        'body'    => VALID_NOT_EMPTY

    );
}

?>
```

The **$validate** array tells Cake how to validate your data when the **save()** method is called. The values for those keys are just constants set by Cake that translate to regex matches (see **/cake/libs/validators.php**). Right now Cake's validation is regex based, but you can also use Model::invalidate() to set your own validation dynamically.

Now that you have your validation in place, use the app to try to add a post without a title or body to see how it works.

Section 11

Deleting Posts

Next, let's make a way for users to delete posts. Start with a delete() action in the PostsController:

/app/controllers/posts_controller.php (delete action only)

```
function delete($id)
{
    $this->Post->del($id);
    $this->flash('The post with id: '.$id.' has been deleted.', '/posts');
}
```

This logic deletes the post specified by $id, and uses flash() to show the user a confirmation message before redirecting them on to /posts.

Because we're just executing some logic and redirecting, this action has no view. You might want to update your index view to allow users to delete posts, however.

/app/views/posts/index.thtml (add and delete links added)

```
<h1>Blog posts</h1>
<p><?php echo $html->link('Add Post', '/posts/add'); ?></p>
<table>
    <tr>
```

```
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

   <!-- Here's where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $html->link($post['Post']['title'], '/posts/view/'.$post['Post']['id']);?>
            <?php echo $html->link(
                'Delete',
                "/posts/delete/{$post['Post']['id']}",
                null,
                'Are you sure?'
            )?>
        </td>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>

</table>
```

This view code also uses the HtmlHelper to prompt the user with a JavaScript confirmation dialog before they attempt to delete a post.

Section 12

Editing Posts

So... post editing: here we go. You're a Cake pro by now, so you should have picked up a pattern. Make the action, then the view. Here's what the edit action of the Posts Controller would look like:

/app/controllers/posts_controller.php (edit action only)

```
function edit($id = null)
{
    if (empty($this->data))
    {
        $this->Post->id = $id;
        $this->data = $this->Post->read();
    }
    else
    {
        if ($this->Post->save($this->data['Post']))
        {
            $this->flash('Your post has been updated.','/posts');
        }
    }
}
```

This checks for submitted form data. If nothing was submitted, go find the Post and hand it to the view. If some data has been submitted, try to save the Post model (or kick back and show the user the validation errors).

The edit view might look something like this:

/app/views/posts/edit.thtml

```
<h1>Edit Post</h1>
<form method="post" action="<?php echo $html->url('/posts/edit')?>">
    <?php echo $html->hidden('Post/id'); ?>
    <p>
        Title:
        <?php echo $html->input('Post/title', array('size' => '40'))?>
        <?php echo $html->tagErrorMsg('Post/title', 'Title is required.') ?>
    </p>
```

```
    <p>
        Body:
        <?php echo $html->textarea('Post/body', array('rows'=>'10')) ?>
        <?php echo $html->tagErrorMsg('Post/body', 'Body is required.') ?>
    </p>
    <p>
        <?php echo $html->submit('Save') ?>
    </p>
</form>
```

This view ouputs the edit form (with the values populated), and the necessary error messages (if present). One thing to note here: Cake will assume that you are edititing a model if the 'id' field is present and exists in a currently stored model. If no 'id' is present (look back at our add view), Cake will assume that you are inserting a new model when save() is called.

You can now update your index view with links to edit specific posts:

/app/views/posts/index.thtml (edit links added)

```
<h1>Blog posts</h1>
<p><a href="/posts/add">Add Post</a></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

   <!-- Here's where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $html->link($post['Post']['title'], '/posts/view/'.$post['Post']['id']);?>
            <?php echo $html->link(
                'Delete',
                "/posts/delete/{$post['Post']['id']}",
                null,
                'Are you sure?'
            )?>
            <?php echo $html->link('Edit', '/posts/edit/'.$post['Post']['id']);?>
        </td>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>

</table>
```

Section 13

Routes

This part is optional, but helpful in understanding how URLs map to specific function calls in Cake. We're only going to make a quick change to routes in this tutorial. For more information, see Chapter "Configuration", Section 3: Routes Configuration.

Cake's default route will take a person visiting the root of your site (i.e. http://www.example.com) to the PagesController, and render a view called home. Rather than do that, we'll want users of our blog application to go to our soon-to-be-created PostsController.

Cake's routing is found in **/app/config/routes.php**. You'll want to comment out or remove the line that looks like this:

```
$Route->connect ('/', array('controller'=>'pages', 'action'=>'display', 'home'));
```

This line connects the URL / with the default Cake home page. We want it to connect with our own controller, so add a

line that looks like this:

```
$Route->connect ('/', array('controller'=>'posts', 'action'=>'index'));
```

This should connect users requesting '/' to the index() action of our soon-to-be-created PostsController.

Section 14

Conclusion

Creating applications this way will win you peace, honor, women, and money beyond even your wildest fantasies. Simple, isn't it? Keep in mind that this tutorial was very basic. Cake has many more features to offer, and is flexible in ways we didn't wish to cover here. Use the rest of this manual as a guide for building more feature-rich applications.

Now that you've created a basic Cake application you're ready for the real thing. Start your own project, read the rest of the Manual and API.

If you need help, come see us in #cakephp. Welcome to Cake!

CakePHP: The Manual

Example: Simple User Authentication

Section 1

The Big Picture

If you're new to CakePHP, you'll be strongly tempted to copy and paste this code for use in your mission critical, sensitive-data-handling production application. Resist ye: this chapter is a discussion on Cake internals, not application security. While I doubt we'll provide for any extremely obvious security pitfalls, **the point of this example is to show you how Cake's internals work**, and allow you to create a bulletproof brute of an application on your own.

Cake has access control via its built-in ACL engine, but what about user authentication and persistence? What about that?

Well, for now, we've found that user authentication systems vary from application to application. Some like hashed passwords, others, LDAP authentication - and almost every app will have User models that are slightly different. For now, we're leaving it up to you. Will this change? We're not sure yet. For now, we think that the extra overhead of building this into the framework isn't worth it, because creating your own user authentication setup is easy with Cake.

You need just three things:

1. A way to authenticate users (usually done by verifying a user's identity with a username/password combination)
2. A way to persistently track that user as they navigate your application (usually done with sessions)
3. A way to check if a user has been authenticated (also often done by interacting with sessions)

In this example, we'll create a simple user authentication system for a client management system. This fictional application would probably be used by an office to track contact information and related notes about clients. All of the system functionality will be placed behind our user authentication system except for few bare-bones, public-safe views that shows only the names and titles of clients stored in the system.

We'll start out by showing you how to verify users that try to access the system. Authenticated user info will be stored in a PHP session using Cake's Session Component. Once we've got user info in the session, we'll place checks in the application to make sure application users aren't entering places they shouldn't be.

One thing to note - authentication is not the same as access control. All we're after in this example is how to see if people are who they say they are, and allow them basic access to parts of the application. If you want to fine tune this access, check out the chapter on Cake's Access Control Lists. We'll make notes as to where ACLs might fit in, but for now, let's focus on simple user authentication.

I should also say that this isn't meant to serve as some sort of primer in application security. We just want to give you enough to work with so you can build bulletproof apps of your own.

Section 2

Authentication and Persistence

First, we need a way to store information about users trying to access our client management system. The client management system we're using stores user information in a database table that was created using the following SQL:

Table 'users', Fictional Client Management System Database

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL auto_increment,
  `username` varchar(255) NOT NULL,
  `password` varchar(32) NOT NULL,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

Pretty simple, right? The Cake Model for this table can be pretty bare:

```
<?php
class User extends AppModel
{
    var $name = 'User';
}
?>
```

First thing we'll need is a login view and action. This will provide a way for application users to attempt logins and a way for the system to process that information to see if they should be allowed to access the system or not. The view is just a HTML form, created with the help of Cake's Html Helper:

/app/views/users/login.thtml

```
<?if ($error): ?>
<p>The login credentials you supplied could not be recognized. Please try again.</p>
<? endif; ?>

<form action="<?php echo $html->url('/users/login'); ?>" method="post">
<div>
    <label for="username">Username:</label>
    <?php echo $html->input('User/username', array('size' => 20)); ?>
</div>
<div>
    <label for="password">Password:</label>
    <?php echo $html->password('User/password', array('size' => 20)); ?>
</div>
<div>
    <?php echo $html->submit('Login'); ?>
</div>
</form>
```

This view presents a simple login form for users trying to access the system. The action for the form is **/users/login**, which is in the UsersController and looks like this:

/app/controllers/users_controller.php (partial)

```
<?php
class UsersController extends AppController
{
    function login()
    {
        //Don't show the error message if no data has been submitted.
        $this->set('error', false);
```

```
        // If a user has submitted form data:
        if (!empty($this->data))
        {
            // First, let's see if there are any users in the database
            // with the username supplied by the user using the form:

            $someone = $this->User->findByUsername($this->data['User']['username']);

            // At this point, $someone is full of user data, or its empty.
            // Let's compare the form-submitted password with the one in
            // the database.

            if(!empty($someone['User']['password']) && $someone['User']['password'] ==
$this->data['User']['password'])
            {
                // Note: hopefully your password in the DB is hashed,
                // so your comparison might look more like:
                // md5($this->data['User']['password']) == ...

                // This means they were the same. We can now build some basic
                // session information to remember this user as 'logged-in'.

                $this->Session->write('User', $someone['User']);

                // Now that we have them stored in a session, forward them on
                // to a landing page for the application.

                $this->redirect('/clients');
            }
            // Else, they supplied incorrect data:
            else
            {
                // Remember the $error var in the view? Let's set that to true:
                $this->set('error', true);
            }
        }
    }

    function logout()
    {
        // Redirect users to this action if they click on a Logout button.
        // All we need to do here is trash the session information:

        $this->Session->delete('User');

        // And we should probably forward them somewhere, too...

        $this->redirect('/');
    }
}
?>
```

Not too bad: the contents of the login() action could be less than 20 lines if you were concise. The result of this action is either 1: the user information is entered into the session and forwarded to the landing page of the app, or 2: kicked back to the login screen and presented the login form (with an additional error message).

Section 3

Access Checking in your Application

Now that we can authenticate users, let's make it so the application will kick out users who try to enter the system from points other than the login screen and the "basic" client directory we detailed earlier.

One way to do this is to create a function in the AppController that will do the session checking and kicking

for you.

/app/app_controller.php

```
<?php
class AppController extends Controller
{
    function checkSession()
    {
        // If the session info hasn't been set...
        if (!$this->Session->check('User'))
        {
            // Force the user to login
            $this->redirect('/users/login');
            exit();
        }
    }
}
?>
```

Now you have a function you can use in any controller to make sure users aren't trying to access controller actions without logging in first. Once this is in place you can check access at any level - here are some examples:

Forcing authentication before all actions in a controller

```
<?php
class NotesController extends AppController
{
    // Don't want non-authenticated users looking at any of the actions
    // in this controller? Use a beforeFilter to have Cake run checkSession
    // before any action logic.

    function beforeFilter()
    {
        $this->checkSession();
    }
}
?>
```

Forcing authentication before a single controller action

```
<?php
class NotesController extends AppController
{
    function publicNotes($clientID)
    {
        // Public access to this action is okay...
    }

    function edit($noteId)
    {
        // But you only want authenticated users to access this action.
        $this->checkSession();
    }
}
?>
```

Now that you have the basics down, you might want to venture out on your own and implement some advanced or customized features past what has been outlined here. Integration with Cake's ACL component might be a good first step.

CakePHP: The Manual

Cake Conventions

Section 1

Conventions, eh ?

Yes, conventions. According to thefreedictionary:

1. General agreement on or acceptance of certain practices or attitudes: By convention, north is at the top of most maps.
2. A practice or procedure widely observed in a group, especially to facilitate social interaction; a custom: the convention of shaking hands.
3. A widely used and accepted device or technique, as in drama, literature, or painting: the theatrical convention of the aside.

Conventions in cake are what make the magic happen, read it **automagic**. Needless to say by favorizing convention over configuration, Cake makes your productivity increase to a scary level without any loss to flexibility. Conventions in cake are really simple and intuitive. They were extracted from the best practices good web developers have used throughout the years in the field of web developement.

Section 2

Filenames

Filenames are **underscore**. As a general rule, if you have a class **MyNiftyClass**, then in Cake, its file should be named my_nifty_class.php.

So if you find a snippet you automatically know that:

1. If it's a Controller named **KissesAndHugsController**, then its filename must be **kisses_and_hugs_controller.php**(notice _controller in the filename)
2. If it's a Model named **OptionValue**, then its filename must be **option_value.php**
3. If it's a Component named **MyHandyComponent**, then its filename must be **my_handy.php**(no need for _component in the filename)
4. If it's a Helper named **BestHelperEver**, then its filename must be **best_helper_ever.php**

Section 3

Models

1. Model class names are **singular**.
2. Model class names are Capitalized for single-word models, and UpperCamelCased for multi-word models.
    1. Examples: Person, Monkey, GlassDoor, LineItem, ReallyNiftyThing
3. many-to-many join tables should be named: alphabetically_first_table_plural_alphabetically_second_table_plural ie: tags_users
4. Model filenames use a lower-case underscored syntax.
    1. Examples: person.php, monkey.php, glass_door.php, line_item.php, really_nifty_thing.php

5. Database tables related to models also use a lower-case underscored syntax - but they are **plural**.
    1. Examples: people, monkeys, glass_doors, line_items, really_nifty_things

CakePHP naming conventions are meant to streamline code creation and make code more readable. If you find it getting in your way, you can override it.

1. Model name: Set `var $name` in your model definition.
2. Model-related database tables: Set `var $useTable` in your model definition.

Section 4

Controllers

1. Controller class names are **plural**.
2. Controller class names are Capitalized for single-word controllers, and UpperCamelCased for multi-word controllers. Controller class names also end with 'Controller'.
    1. Examples: PeopleController, MonkeysController, GlassDoorsController, LineItemsController, ReallyNiftyThingsController
3. Controller file names use a lower-case underscored syntax. Controller file names also end with '_controller'. So if you have a controller class called PostsController, the controller file name should be posts_controller.php
    1. Examples: people_controller.php, monkeys_controller.php, glass_doors_controller.php, line_items_controller.php, really_nifty_things_controller.php
4. For protected member visibility, controller action names should be prepended with '-'.
5. For private member visibility, controller action names should be prepended with '--'.

Section 5

Views

1. Views are named after actions they display.
2. Name the view file after action name, in lowercase.
    1. Examples: PeopleController::worldPeace() expects a view in **/app/views/people/world_peace.thtml**; MonkeysController::banana() expects a view in **/app/views/monkeys/banana.thtml**.

You can force an action to render a specific view by calling $this->render('name_of_view_file_without_dot_thtml'); at the end of your action.

Section 6

Helpers

1. Helper classname is CamelCased and ends in "Helper", the filename is underscored.
    1. Example: class MyHelperHelper extends Helper is in **/app/views/helpers/my_helper.php**.

Include in the controller with var $helpers = array('Html','MyHelper'); in the view you can access with $myHelper->method().

Section 7

Components

1. Component classname is CamelCased and ends in "Component", the filename is underscored.
    1. Example: class MyComponentComponent extends Object is in **/app/controllers/components/my_component.php**.

Include in the controller with var $components = array('MyComponent'); in the controller you can access with $this->MyComponent->method().

Section 8

Vendors

Vendors don't follow any convention for obvious reasons: they are thirdparty pieces of code, Cake has no control over them.