CAKEPHP 1.2

The Good Stuff

John David Anderson
January 2007

Models

BEGINNING WITH CAKEPHP

Patty-cake, patty-cake...

Preface

Welcome to web development heaven.

If you're reading the preface to a technical manual, you probably have too much time on your hands. I'm no celebrity, and the material is what you're after, so skip this superfluous section and dive right in.

Audience

In order to read this manual, you're going to need to be moderately familiar with PHP. Some familiarity with object-oriented programming will help, though I suppose the introductory sections of this manual could act as a primer of sorts. That said, this material is written to developers of all levels of ability—anyone who wants to create robust, maintainable applications quickly and enjoyably.

I should say that there will be sections that address technologies that are really out of the scope of this manual. Web server administration, AJAX and JavaScript details and the like may be touched on in places, but for the most part we're just going to stick to CakePHP.

Introduction to CakePHP

What is CakePHP? Why Use it?

CakePHP is a free, open-source, rapid development framework for PHP. It's a foundational structure for programmers to create web applications. Our primary goal is to enable you to work in a structured and rapid manner—without loss of flexibility.

CakePHP takes the monotony out of web development. We provide you with all the tools you need to get started coding what you *really* need to get done: the logic specific to your application. Instead of reinventing the wheel every time you sit down to a new project, check out a copy of CakePHP and get started with the real guts of your application.

CakePHP has an active developer team and community, bringing great value to the project. In addition to keeping you from re-inventing, using CakePHP means your application's core is well tested and is being constantly improved.

Here's a quick list of features you'll enjoy when using CakePHP:

- Active, friendly community
- Flexible licensing
- Compatible with PHP4 and PHP5
- Integrated CRUD for database interaction
- Application scaffolding
- Code generation
- Model View Controller (MVC) architecture
- Request dispatcher with clean, custom URLs and routes
- Built-in validation
- Fast and flexible templating (PHP syntax, with helpers)
- View Helpers for AJAX, Javascript, HTML Forms and more
- Email, Cookie, Security, Session, and Request Handling Components
- Flexible access control lists
- Data Sanitization
- Flexible View Caching
- Localization
- Works from any web site subdirectory, with little to no Apache configuration involved
- @@@1.2 new stuff??

The History of CakePHP

In 2005, Michal Tatarynowicz wrote a minimal version of a rapid application framework in PHP. Michal published the framework, dubbing it Cake, and opened it up to a community of developers, who now maintain Cake under the name CakePHP.

The CakePHP Development Team

Larry Masters (PhpNut), Development Lead

Garrett Woodworth (gwoo), Project Manager

Nate Abele (_nate_), Core Developer

John Anderson (_psychic_), Documentation Lead

Daniel Hofstetter (dhofset), Documentation

Armando Sosa (sosa), Designer

Felix Geisendörfer (the_undefined), Contributor

Chris Partridge (parchy), Contributor

Jeff Loiselle (phishy), Oracle DBO

Daniel Feinberg (CSDread_), Contributor

Where to Get Help

You've started in the right place. This manual (and the API) should probably be the first place you go to get answers. As with many other open source projects, we get new folks regularly. Try your best to answer your questions on your own first. Answers may come slower, but will remain longer–and you'll also be lightening our support load. Both the manual and the API have an online component.

http://manual.cakephp.org/

http://api.cakephp.org/

If you're stumped, give us a holler in the CakePHP IRC channel. Someone from the development team is usually there, especially during the daylight hours for North and South America users. We'd love to hear from you, whether you need some help, want to find users in your area, or would like to donate your brand new sports car.

#cakephp @ irc.freenode.net

The CakePHP Bakery is a clearing house for all things CakePHP. Check it out for tutorials, case studies, and code examples. Once you're acquainted with CakePHP, log on and share your knowledge with the community and gain instant fame and fortune.

http://bakery.cakephp.org

CakeForge is another developer resource you can use to host your CakePHP projects to share with others. If you're looking for (or want to share) a killer component or a praiseworthy plugin, check out CakeForge.

http://www.cakeforge.org

The Official CakePHP website is always a great place to visit. It features links to oft-used developer tools, screencasts, donation opportunities, and downloads.

http://www.cakephp.org

Understanding Model-View-Controller

Overview

Well written CakePHP applications follow the MVC (Model-View-Controller) software design pattern. Programming using MVC separates your application into three main parts. The *model* represents an application's data, the *view* renders a presentation of model data, and the *controller* handles and routes requests made by users.

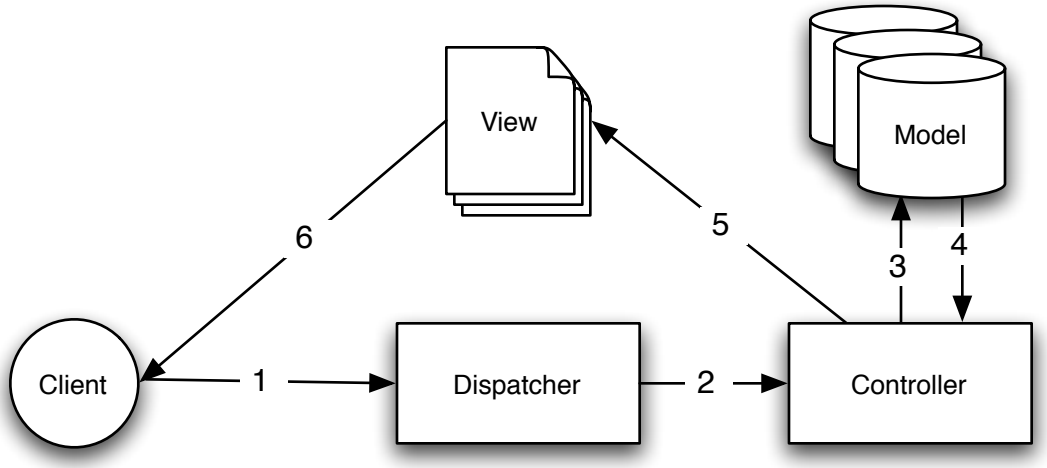

Figure 1: A Basic MVC Request.

Figure 1 shows an example of a bare-bones MVC request in CakePHP. For illustrative purposes, let's say a user named Ricardo just clicked on the "Buy A Custom Cake Now!" link on your application's landing page.

1. Ricardo clicks the link pointing to http://www.example.com/cakes/buy, and his browser makes a request to your web server.
2. The dispatcher checks the request URL (/cakes/buy), and hands the request to the correct controller.
3. The controller performs application specific logic. For example, it may check to see if Ricardo has logged in.
4. The controller also uses models to gain access to the application's data. Most often, models represent database tables, but they could also represent LDAP entries, RSS feeds, or files on the system. In this example, the controller uses a model to fetch Ricardo's last purchases from the database.
5. Once the controller has worked its magic on the data, it hands it to a view. The view takes this data and gets it ready for presentation to the user. Views in CakePHP most often come in HTML format, but a view could as easily be a PDF, XML document, or JSON object depending on your needs.
6. Once the view has used the data from the controller to build a fully rendered view, the content of that view is returned to Ricardo's browser.

Almost every request to your application will follow this basic pattern. We'll fill in some Cake-specific details later on, so keep this in the back of your mind as we move on.

Benefits

Why use MVC? Because it is a tried and true software design pattern that turns an application into a maintainable, modular, rapidly developed package. Crafting application tasks into separate models, views, and controllers makes your application very light on its feet. New features are easily added, and new faces on old features are a snap. The modular and separate design also allows developers and designers to work simultaneously, including the ability to rapidly prototype. Separation also allows developers to make changes in one part of the application without affecting others.

If you've never built an application this way, it takes some getting used to, but we're confident that once you've built your first CakePHP application, you won't want to go back.

BASIC PRINCIPLES OF CAKEPHP

The Start of Becoming a Smart Cookie

CakePHP Structure

CakePHP features Controller, Model, and View classes, but it also features some additional classes and objects that make development in MVC a little quicker and more enjoyable. These meta-MVC classes usually aid the main MVC classes directly. Right now we'll stay at a higher level, so look for the details on how to use these tools later on.

Controller Aids

A *Component* is a class that aids in controller logic. If you have some logic you want to share between controllers (or applications), a component is usually a good fit. As an example, the core EmailComponent class makes creating and sending emails a snap. Rather than writing a controller method in a single controller that performs this logic, you can package the logic so it can be shared.

Controllers are also fitted with callbacks. These callbacks are available for your use, just in case you need to insert some logic between CakePHP's core operations. Callbacks available include:

- beforeFilter(), executed before any controller action logic
- beforeRender(), executed after controller logic, but before the view is rendered
- afterRender(), executed after the view has been rendered
- afterFilter(), executed after all controller logic, including the view render. There may be no difference between afterRender() and afterFilter() unless you've manually made a call to render() in your controller action and have included some logic after that call.

View Aids

A *Helper* is a class that aids in view logic. Much like a component used among controllers, helpers allow presentational logic to be accessed and shared between

views. One of the core helpers, AjaxHelper, makes Ajax requests within views much easier.

Most applications have pieces of view code that repeatedly used. CakePHP facilitates view code reuse with *layouts* and *elements*. By default, every view rendered by a controller is placed inside a layout. Elements are used when small snippets of content need to be reused in multiple views.

Model Aids

Similarly, *Behaviors* work as ways to add common functionality between models. For example, if you store user data in a tree structure, you can specify your User model as behaving like a tree, and gain free functionality for removing, adding, and shifting nodes in your underlying tree structure.

Models also are supported by another class called a *DataSource*. DataSources are an abstraction that enable models to manipulate different types of data consistently. While the main source of data in a CakePHP application is often a database, you might write additional DataSources that allow your models to represent RSS feeds, CSV files, LDAP entries, or iCal events. DataSources allow you to associate records from different sources: rather than being limited to SQL joins, DataSources allow you to tell your LDAP model that it is associated to many iCal events.

Just like controllers, models are featured with callbacks as well:

- beforeFind()
- afterFind()
- beforeValidate()
- beforeSave()
- afterSave()
- beforeDelete()
- afterDelete()

The names of these methods should be descriptive enough to let you know what they do. Be sure to get the details in the models chapter.

Application Aids

Both controllers and models have a parent class you can use to define application-wide changes. AppController (located at `/app/app_controller.php`), and AppModel (located at `/app/app_model.php`) are great places to put methods you want to share between all controllers or models.

Although they aren't a class or file, *routes* play a role in requests made to CakePHP. Route definitions tell CakePHP how to map URLs to controller actions. The default behavior assumes that the URL "/controller/action/var1/var2" maps to `Controller:action($var1, $var2)`, but you can use routes to customize URLs and how they are interpreted by your application.

Some features in an application merit packaging as a whole. A *plugin* is a package of models, controllers and views that accomplish a specific purpose that can span multiple applications. A user management system or a simplified blog might be good fits for CakePHP plugins.

CakePHP File Structure

Let's take a look at what CakePHP looks like right out of the box. You know what CakePHP looks like from a basic MVC request standpoint, but you'll need to know how its files are organized as well.

- app
- cake
- docs
- index.php
- vendors

When you download CakePHP, you will see four main folders. The *app* folder will be where you work your magic: it's where your application's files will be placed. The *cake* folder is where we've worked our magic. Make a personal commitment not to edit files in this folder. We can't help you if you've modified the core. The *docs* folder is for the quintessential readme, changelog, and licensing information. Finally, the *vendors* folder is where you'll place third-party PHP libraries you need to use with your CakePHP applications.

The App Folder

CakePHP's *app* folder is where you will do most of your application development. Let's look a little closer and the folders inside of *app*.

- config
  - Holds the (few) configuration files CakePHP uses. Database connection details, bootstrapping, core configuration files and more should be stored here.
- controllers
  - Contains your application's controllers and their components.
- locale
  - @@@
- models
  - Contains your application's models, behaviors, and datasources.
- plugins
  - Contains plugin packages.
- tmp
  - This is where CakePHP stores temporary data. The actual data it stores depends on how you have CakePHP configured, but this folder is usually used to store model descriptions, logs, and sometimes session information.
- vendors
  - Any third-party classes or libraries should be placed here. Doing so makes them easy to access using the vendors() function.
- views
  - Presentational files are placed here: elements, error pages, helpers, layouts, and view files.
- webroot

- In a production setup, this folder should serve as the document root for your application. Folders here also serve as holding places for CSS stylesheets, images, and JavaScript files.

A Typical CakePHP Request

We've covered the basic ingredients in CakePHP, so let's look at how each object works together to complete a basic request. Continuing with our original request example, let's imagine that our friend Ricardo just clicked on the "Buy A Custom Cake Now!" link on a CakePHP application's landing page.

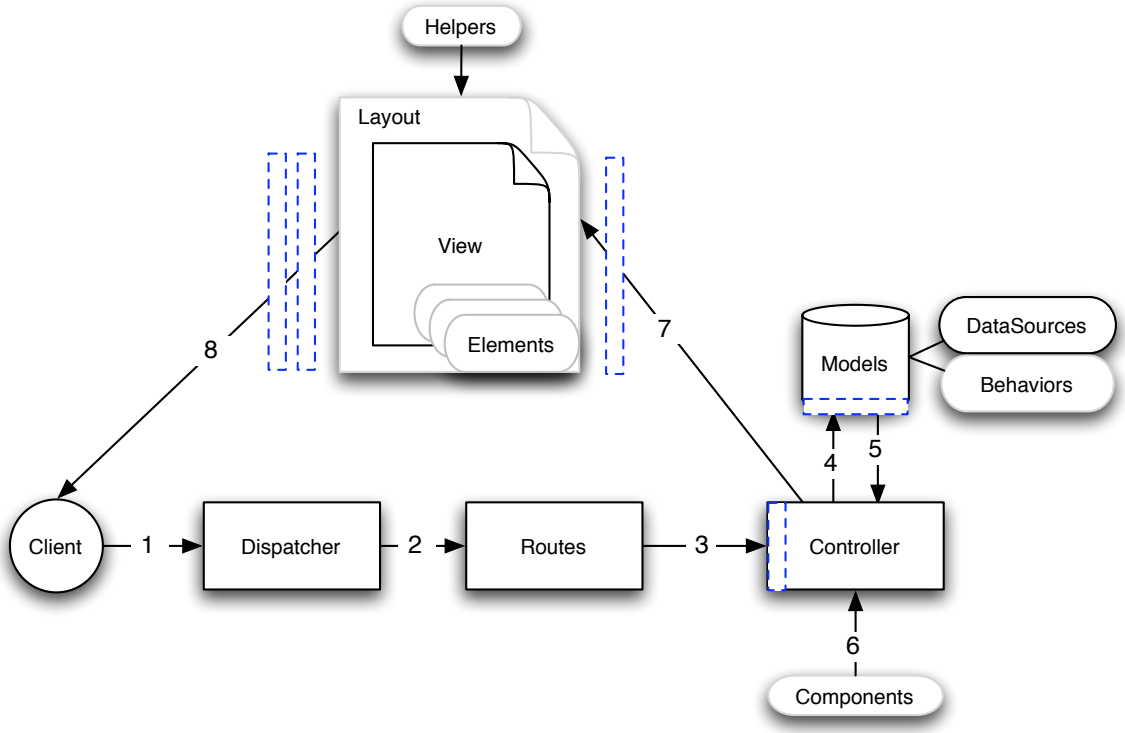

Figure 2. Typical Cake Request.

Black = required element, Gray = optional element, Blue = callback

1. Ricardo clicks the link pointing to http://www.example.com/cakes/buy, and his browser makes a request to your web server.

2. The Dispatcher is a CakePHP core class that interprets the URL, uses Routes to create the correct controller instance, calls the appropriate callbacks, and calls the correct controller methods to render a requested action.

3. Using routes, a request URL is mapped to a controller action (a method in a specific controller class). In this case, it's the buy() method of the CakesController. The controller's beforeFilter() callback is called before any controller action logic is executed.

4. The controller may use models to gain access to the application's data. In this example, the controller uses a model to fetch Ricardo's last purchases from the database. Any applicable model callbacks, behaviors, and DataSources may apply during this operation. While model usage is not required, all CakePHP controllers initially require at least one model.
5. After the model has retrieved the data, it is returned to the controller. Model callbacks may apply.
6. The controller may use components to further refine the data or perform other operations (session manipulation or sending emails, for example).
7. Once the controller has used models and components to prepare the data sufficiently, that data is handed to the view using the controller's set() method. Controller callbacks may be applied before the data is sent. The view logic is performed, which may include the use of elements and/or helpers. By default, the view is rendered inside of a layout.
8. Additional controller callbacks (afterRender and afterFilter) may be applied. The complete, rendered view code is sent to Ricardo's browser.

CakePHP Conventions

We're a big fan of convention over configuration. While it takes a bit of time to learn CakePHP's conventions, you save time in the long run: by following convention, you get free functionality, and you free yourself from the maintenance nightmare of tracking config files. Convention also makes for a very uniform system development, allowing other developers to jump in an help more easily.

CakePHP's conventions have been distilled out of years of web development experience and best practices. While we suggest you use these conventions while developing with CakePHP, we should mention that each of these tenets is easily overridden—something that is especially handy when working with legacy systems.

File and Classname Conventions

In general, filenames are underscored, while classnames are CamelCased. The class `KissesAndHugsController` can be found in the file `kisses_and_hugs_controller.php`, for example.

The name of the class a file holds may not necessarily be found in the filename, however. The class `EmailComponent` is found in a file named `email.php`, and the class `HtmlHelper` is found in a file named `html.php`.

Model Conventions

Model classnames are singular and CamelCased. `Person`, `BigPerson`, and `ReallyBigPerson` are all examples of conventional model names.

Table names corresponding to CakePHP models are plural and underscored. The underlying tables for the above mentioned models would be `people`, `big_people`, and `really_big_people`, respectively.

Join tables, used in hasAndBelongsToMany relationships between models should be named after the model tables they will join, in lexical order. If your application features this type of relationship between your `Tag` and `Post` models, the table name would be `posts_tags`.

Controller Conventions

Controller classnames are plural, CamelCased, and end in 'Controller.' PeopleController, BigPeopleController, and ReallyBigPeopleController are all examples of conventional controller names.

The first function you write for a controller might be the `index()` function. When a request specifies a controller but not an action, the default CakePHP behavior is to render the `index()` function of that controller. For example, a request to http://www.example.com/apples/ maps to a call on the `index()` function of the `ApplesController`, where as http://www.example.com/apples/view maps to a call on the `view()` function of the `ApplesController`.

You can also simulate private and protected controller functions in CakePHP by prepending controller function names with underscores. For protected member visibility, controller function names should be prepended with _ (a single underscore). For private member visibility, controller function names should be prepended with __ (two underscores).

View Conventions

View template files are named after the controller functions they display, in an underscored form. The `getReady()` function of the `PeopleController` class will look for a view template in `/app/views/people/get_ready.ctp`.

The basic pattern is /app/views/controller/underscored_function_name.ctp.

By naming the pieces of your application using CakePHP conventions, you gain functionality without the hassle and maintenance tethers of configuration. Here's a final example thatties the conventions

- Database table: 'people'
- Model class: 'Person', found at /app/models/person.php
- Controller class: 'PeopleController', found at /app/controllers/people_controller.php
- View template, found at /app/views/people/index.ctp

Using these conventions, CakePHP knows that a request to http://example.com/people/ maps to a call on the index() function of the PeopleController, where the Person model is automatically available (and automatically tied to the 'people' table in the database), and renders to a file. None of these relationships have been configured by any means other than by creating classes and files that you'd need to create anyway.

Now that you've been introduced to CakePHP's fundamentals, you might try a run through the CakePHP Blog Tutorial, an appendix at the end of this manual.

DEVELOPING WITH CAKEPHP

Now you're cooking.

Requirements

- HTTP Server. Apache with mod_rewrite is preferred, but by no means required.
- PHP 4.3.2 or greater. Yes, CakePHP works great on PHP 4 and 5.

Technically a database engine isn't required, but we imagine that most applications will utilize one. CakePHP supports a variety of database storage engines:

- MySQL (4 or greater)
- PostgreSQL
- Microsoft SQL Server
- Oracle
- SQLite
- ODBC
- ADOdb
- PEAR::DBO

Installation Preparation

Getting CakePHP

There are two main ways to get a fresh copy of CakePHP. First, you can download an archive (zip/tar.gz/tar.bz2), or you can check out a code from our repository (SVN).

To get a fresh archive, visit our web site at http://www.cakephp.org. Follow the huge "Download Now!" link to paradise. CakePHP downloads are hosted at CakeForge, and you can also visit the project web site at http://cakeforge.org/projects/cakephp.

If you want to live on the edge, check out our nightly downloads at http://cakephp.org/downloads/index/nightly. CakePHP nightlies are stable, and include fixes between releases.

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

Permissions

CakePHP uses the `/app/tmp` directory for a number of different operations. Model descriptions, cached views, and session information are just a few examples.

As such, make sure the `/app/tmp` directory in your cake installation is writable by the web server user.

Installation

Installing CakePHP can be as simple as slapping it in your web server's document root, or as complex and flexible as you wish. This section will cover the three main installation types for CakePHP: development, production, and advanced.

- **Development:** easy to get going, URLs for the application include the CakePHP installation directory name, and less secure.
- **Production:** Requires the ability to configure the web server's document root, clean URLs, very secure.
- **Advanced:** With some configuration, allows you to place key CakePHP directories in different parts of the filesystem, possibly sharing a single CakePHP core library folder amongst many CakePHP applications.

Development

Just place your cake install inside your web server's document root. For example, assuming your web server's document root is `/var/www/html`, a development setup would look like this on the filesystem:

- `/var/www/html`
  - `/cake`
    - `/app`

    - `/cake`
    - `/docs`
    - `/index.php`
    - `/vendors`

To see your CakePHP application, point your web browser to http://www.example.com/cake/.

Production

In order to utilize a production setup, you will need to have the rights to change the document root for your web server. Choosing the production setup means the whole domain acts as a single CakePHP application.

The production setup uses the following layout:

- `/path_to_cake_install/`
  - `/app`
    - `/webroot` (this directory is set as the document root for the web server)
  - `/cake`
  - `/docs`
  - `/index.php`
  - `/vendors`

If this application was going to be hosted on Apache, the DocumentRoot directive for the domain would look something like:

```
DocumentRoot /path_to_cake_install/app/webroot
```

To see your CakePHP application, point your web browser to http://www.example.com.

Advanced Installation

There may be some situations where you may wish to place CakePHP's directories on different places on the filesystem. 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. This section describes how to spread your CakePHP directories across a filesystem.

First, realize that there are three main parts to a Cake application:

1. The core CakePHP libraries, in `/cake`.
2. Your application code, in `/app`.
3. The application's webroot, usually in `/app/webroot`.

Each of these directories can be located anywhere on your file system, with the exception of the `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`. There are three constants that you'll need to edit: `ROOT`, `APP_DIR`, and `CAKE_CORE_INCLUDE_PATH`.

- `ROOT` should be set to the path of the directory that *contains* your app folder.
- `APP_DIR` should be set to the path of your app folder.
- `CAKE_CORE_INCLUDE_PATH` should be set to the path of your CakePHP libraries folder.

Let's run through an example so you can see what an advanced installation might look like in practice. Imagine that I wanted to set up CakePHP to work as follows:

- The CakePHP core libraries will be placed in `/usr/lib/cake`.
- My application's `webroot` directory will be `/var/www/mysite/`.
- My application's `app` directory will be stored in `/home/me/mysite`.

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

```
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');
}
```

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

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

Apache and mod_rewrite

While CakePHP is built to work with `mod_rewrite` out of the box–and usually does–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:

- Make sure that an `.htaccess` override is allowed. In your `httpd.conf`, you should have a section that defines your Directory on the server. Make sure the `AllowOverride` is set to `All` for the correct Directory.
- Make sure you are editing the system `httpd.conf` rather than a user- or site-specific `httpd.conf`.
- Is CakePHP missing its needed .htaccess files? This sometimes happens during copying or moving because some operating systems treat files that start with '.' as hidden. Make sure your copy of CakePHP is from the downloads section of the site or our SVN repository, and has been unpacked correctly.
- 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 are installing CakePHP into a user directory (http://example.com/~username), you'll need to modify the `.htaccess` file in the base directory of your CakePHP installation. Just add the line "`RewriteBase /~myusername/`".

Fire It Up

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

Congratulations! You are ready to create your first CakePHP application.

Configuration

Database Configuration

CakePHP expects database configuration details to be in a file at `app/config/database.php`. An example database configuration file can be found at `app/config/database.php.default`. A finished configuration should look something like this.

```
var $default = array('driver'   => 'mysql',
                     'connect'  => 'mysql_connect',
                     'host'     => 'localhost',
                     'login'    => 'cakephpuser',
                     'password' => 'c4k3roxx!',
                     'database' => 'my_cakephp_project',
                     'prefix'   => '');
```

Example Database Configuration

The `$default` connection array is used unless another connection is specified by the `$useDbConfig` property in a model. For example, if my application an additional legacy database in addition to the default one, I could use it in my models by creating a new `$legacy` database connection array similar to the `$default` array, and by setting `var $useDbConfig = 'legacy';` in the appropriate models.

Fill out the key/value pairs in the configuration array to best suit your needs.

Key	Values
driver	The name of the database driver this configuration array is for. Examples: `mysql`, `postgres`, `sqlite`, `pear-drivername`, `adodb-drivername`, `mssql`, `oracle`, or `odbc`.
connect	The name of the php function you want CakePHP to use when connecting. Examples: `mysql_connect`, `mysql_pconnect`, `sqlite_open`, `ocilogon`, `ociplogon`.
host	The database server's hostname (or IP address).
login	The username for the account.
password	The password for the account.
database	The name of the database for this connection to use.
prefix	The string that prefixes every table name in the database. If you're tables don't have prefixes, set this to an empty string.
port (optional)	The TCP port used to connect to the server.

Note that the `prefix` setting is for tables, **not** models. For example, if you create a join table for your Apple and Flavor models, you name it `prefix_apples_flavors` (**not** `prefix_apples_prefix_flavors`), and set your `prefix` setting to '`prefix_`'.

At this point, you might want to take a look at the CakePHP Conventions, listed in an appendix to this manual. The correct naming for your tables (and the addition of some columns) can score you some free functionality and help you avoid configuration.

Core Configuration

Application configuration in CakePHP is found in /app/config/core.php. This file is a collection of constant definitions that determine how your application behaves.

core.php constant	Description
BASE_URL	Un-comment this definition if you **don't** plan to use Apache's mod_rewrite with CakePHP. Don't forget to remove your .htaccess files too.
DEBUG	Changes CakePHP debugging output. 0 = Production mode. No output. 1 = Show errors and warnings. 2 = Show errors, warnings, and SQL. 3 = Show errors, warnings, SQL, and complete controller dump.
CACHE_CHECK	When set to false, caching is disabled site-wide.
LOG_ERROR	Defines the default error type when using the log() function.
CAKE_SESSION_SAVE	Tells CakePHP which session storage mechanism to use. php = Use the default PHP session storage. cake = Store session data in /app/tmp database = store session data in a database table. Make sure to set up the table using the SQL file located at /app/config/sql/sessions.sql.
CAKE_SESSION_TABLE	The name of the table (not including any prefix) that stores session information.
CAKE_SESSION_STRING	@@@
CAKE_SESSION_COOKIE	The name of the cookie used to track sessions.
CAKE_SECURITY	Sets the level of session security. @@@ timeouts, ID regeneration

core.php constant	Description
WEBSERVICES	Enables the use of webservices routing when set to true.
COMPRESS_CSS	Compresses output CSS (removes comments, whitespace, repeating tags) when set to true. This requires a /var/cache directory to be writable by the web server. @@@ To use, prefix the CSS link URL with '/ccss/' instead of '/css/' or use Controller::cssTag().
AUTO_SESSION	Automatically starts sessions when set to true.
MAX_MD5SIZE	@@@
ACL_CLASSNAME, ACL_FILENAME, ACL_DATABASE	Constants used for CakePHP's Access Control List functionality. See the Access Control Lists chapter for more information.

The Configure class can also be used to read and write core configuration settings on the fly. This can be especially handy if you want to turn the DEBUG setting on for a limited section of logic in your application, for instance.

@@@Configure class details, which vars, static calls from everywhere?

Routes Configuration

Routing is a feature that maps URLs to controller actions. It was added to CakePHP to make pretty URLs more configurable and flexible. Using Apache's mod_rewrite is not required for using routes, but it will make your address bar look much more tidy.

Routes in CakePHP 1.2 have been expanded and can be very powerful.

Before you learn about configuring your own routes, you should know that CakePHP comes configured with a default set of routes. CakePHP's default routing will get you pretty far in any application. You can access an action directly via the

URL by putting its name in the request. You can also pass parameters to your controller actions using the URL.

```
http://example.com/controller/action/param1/param2/param3
```

URL pattern default routes

The URL `/posts/view` maps to the `view()` action of the PostsController, and `/products/viewClearance` maps to the `view_clearance()` action of the ProductsController. If no action is specified in the URL, the `index()` method is assumed.

The default routing setup also allows you to pass parameters to your actions using the URL. A request for `/posts/view/25` would be equivalent to calling `view(25)` on the PostsController, for example.

New in CakePHP 1.2 is the ability to use named parameters. You can name parameters and send their values using the URL. A request for `/posts/view/title:first+post/category:general` would result in a call to the view() action of the PostsController. In that action, you'd find the values of the title and category parameters inside `$this->passedArgs['title']` and `$this->passedArgs['category']` respectively.

Some summarizing examples for default routes might prove helpful.

```
URL: /monkeys/jump
Mapping: MonkeysController->jump();

URL: /products
Mapping: ProductsController->index();

URL: /tasks/view/45
Mapping: TasksController->view(45);

URL: /donations/view/recent/2001
Mapping: DonationsController->view('recent', '2001');

URL: /contents/view/chapter:models/section:associations
Mapping: ContentsController->view();
$this->passedArgs['chapter'] = 'models';
$this->passedArgs['section'] = 'associations';
```

URL to controller action mapping using default routes

Defining your own routes allows you to define how your application will respond to a given URL. Define your own routes in the `/app/config/routes.php` file using the `Router::connect()` method.

The `connect()` method takes up to three parameters: the URL you wish to match, the the default values for custom route elements, and regular expression rules to help the router match elements in the URL.

The basic format for a route definition is:

```
Router::connect(
    'URL',
    array('paramName' => 'defaultValue'),
    array('paramName' => 'matchingRegex')
)
```

Route definition pattern.

The first parameter is used to tell the router what sort of URL you're trying to control. The URL is a normal slash delimited string, but can also contain a wildcard (*) or custom route elements (URL elements prefixed with a colon). Using a wildcard tells the router what sorts of URLs you want to match, and specifying route elements allows you to gather parameters for your controller actions.

Once you've specified a URL, you use the last two parameters of connect() to tell CakePHP what to do with a request once it has been matched. The second parameter is an associative array. The keys of the array should be named after the route elements in the URL, or the default elements: :controller, :action, and :plugin. The values in the array are the default values for those keys. Let's look at some basic examples before we start using the third parameter of `connect()`.

```
Router::connect(
    '/pages/*',
    array('controller' => 'pages', 'action' => 'display')
);
```

This route is found in the `routes.php` file distributed with CakePHP (line 40). This route matches any URL starting with `/pages/` and hands it to the display() method of the PagesController(); The request `/pages/products` would be mapped to `PagesController->display('products')`, for example.

```
Router::connect(
    '/government',
    array('controller' => 'products', 'action' => 'display', 5)
);
```

This second example shows how you can use the second parameter of `connect()` to define default parameters. If you built a site that features products for different categories of customers, you might consider creating a route. This allows you link to `/government` rather than `/products/display/5`.

For additional flexibility, you can specify custom route elements. Doing so gives you the power to define places in the URL where parameters for controller actions should lie. When a request is made, the values for these custom route elements are found in `$this->params` of the controller. This is different than named parameters are handled, so note the difference: named parameters (/controller/action/name:value) are found in `$this->passedArgs`, whereas custom route element data is found in `$this->params`. When you define a custom route element, you also need to specify a regular expression - this tells CakePHP how to know if the URL is correctly formed or not.

```
Router::connect(
    '/:controller/:id',
    array('action' => 'view'),
    array('id' => '[0-9]+')
);
```

This simple example illustrates how to create a quick way to view models from any controller by crafting a URL that looks like `/controllername/id`. The URL provided to `connect()` specifies two route elements: `:controller` and `:id`. The `:controller` element is a CakePHP default route element, so the router knows how to match and identify controller names in URLs. The `:id` element is a custom route element, and must be further clarified by specifying a matching regular expression in the third parameter of `connect()`. This tells CakePHP how to recognize the ID in the URL as opposed to something else, such as an action name.

Once this route has been defined, requesting `/apples/5` is the same as requesting `/apples/view/5`. Both would call the `view()` method of the `ApplesController`. Inside the `view()` method, you would need to access the passed ID at `$this->params['id']`.

One more example, and you'll be a routing pro.

```
Router::connect(
      '/:controller/:year/:month/:day',
      array('action' => 'index', 'day' => null),
      array(
            'year' => '[12][0-9]{3}',
            'month' => '(0[1-9]|1[012])',
            'day' => '(0[1-9]|[12][0-9]|3[01])'
      )
);
```

This is rather involved, but shows how powerful routes can really become. The URL supplied has four route elements. The first is familiar to us: it's a default route element that tells CakePHP to expect a controller name.

Next, we specify some default values. Regardless of the controller, we want the `index()` action to be called. We set the `day` parameter (the fourth element in the URL) to `null` to flag it as being *optional*.

Finally, we specify some regular expressions that will match years, months and days in numerical form.

Once defined, this route will match `/articles/2007/02/01`, `/posts/2004/11/16`, and `/products/2001/05` (remember that the day parameter is optional?), handing the requests to the `index()` actions of their respective controllers, with the custom date parameters in `$this->params`.

Custom Inflections

Cake's naming conventions can be really nice—you can name your database table big_boxes, your model BigBox, your controller BigBoxesController, and everything just works together automatically. The way CakePHP knows how to tie things together is by *inflecting* the words between their singular and plural forms.

There are occasions (especially for our non-english speaking friends) where you may run into situations where CakePHP's inflector (the class that pluralizes, singularizes, camelCases, and under_scores) might not work as you'd like. If CakePHP won't recognize your Foci or Fish, editing the custom inflections configuration file is where you can tell CakePHP about your special cases. This file is found in `/app/config/inflections.php`.

In this file, you will find six variables. Each allows you to fine-tune CakePHP inflection behavior.

`inflections.php` variable	Description
$pluralRules	This array contains regular expression rules for pluralizing special cases. The keys of the array are patterns, and the values are replacements.
$uninflectedPlural	An array containing words that do not need to be modified in order to be plural (mass nouns, etc.).
$irregularPlural	An array containing words and their plurals. The keys of the array contain the singular form, the values, plural forms. This array should be used to store words that don't follow rules defined in $pluralRules.
$singularRules	Same as with $pluralRules, only this array holds rules that singularize words.
$uninflectedSingular	Same as with $uninflectedPlural, only this array holds words that have no singular form. This is set equal to $uninflectedPlural by default.
$irregularSingular	Same as with $irregularPlural, only with words in singular form.

Bootstrapping CakePHP

If you have any additional configuration needs, use CakePHP's bootstrap file, found in `/app/config/bootstrap.php`. This file is executed just after CakePHP's core bootstrapping.

This file is ideal for a number of common bootstrapping tasks:

- Defining convenience functions
- Registering global constants
- Defining additional model, view, and controller paths

Be careful to maintain the MVC software design pattern when you add things to the bootstrap file: it might be tempting to place formatting functions there in order to use them in your controllers.

Resist the urge. You'll be glad you did later on down the line.

You might also consider placing things in the AppController class. This class is a parent class to all of the controllers in your application. AppController is handy place to use controller callbacks and define methods to be used by all of your controllers.

Controllers

Introduction

A controller is used to manage the logic for a part of your application. Most commonly, controllers are used to manage the logic for a single model. For example, if you were building a site for an online bakery, you might have a RecipesController and a IngredientsController managing your reciepes and their ingredients. In CakePHP, controllers are named after the model they handle, in plural form.

The Recipe model is handled by the RecipesController, the Product model is handled by the ProductsController, and so on.

Your application's controllers are classes that extend the CakePHP AppController class, which in turn extends a core Controller class. The AppController class can be defined in `/app/app_controller.php` and it should contain methods that are shared between all of your application's controllers. It extends the Controller class which is a standard CakePHP library.

Controllers can include any number of methods which are usually referred to as *actions*. Actions are controller methods used to display views. An action is a single method of a controller. CakePHP's dispatcher calls actions when an incoming request matches a URL to a controller's action. Returning to our online bakery example, our RecipesController might contain the view(), share(), and search() actions. The controller would be found in /app/controllers/recipes_controller.php and contain:

```
<?php

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

    function share($customer_id, $recipe_id)
    {
        //action logic goes here..
    }

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

?>
```

/app/controllers/recipes_controller.php

In order for you to use a controller effectively in your own application, we'll cover some of the core attributes and methods provided by CakePHP's controllers.

Controller Attributes

For a complete list of controller attributes and their descriptions visit the CakePHP API. Check out http://api.cakephp.org/1.2/classController.html.

$name

PHP4 users should start out their controller definitions using the `$name` attribute. The `$name` attribute should be set to the name of the controller. Usually this is just the plural form of the primary model the This takes care of some PHP4 classname oddities and helps CakePHP resolve naming.

```
<?php

class RecipesController extends AppController
{
    var $name = 'Recipes';
}

?>
```

$name controller attribute usage example

$components, $helpers and $uses

The next most oft used controller attributes tell CakePHP what helpers, components, and models you'll be using in conjunction with the current controller. Using these attributes make these MVC classes available to the controller as a class variable (`$this->ModelName`, for example).

Please note that each controller has a some of these classes available by default, so you may not need to configure your controller to use additional classes.

Controllers have their primary model available by default. Our RecipesController will have the Recipe model class available at `$this->Recipe`, and our ProductsController also features the Product model at `$this->Product`.

The Html- and SessionHelpers are always available by default, as is the SessionComponent. To learn more about these classes, be sure to check out their respective sections later in this manual.

Let's look at how to tell a CakePHP controller that you plan to use additional MVC classes.

```
<?php

class RecipesController extends AppController
{
    var $name = 'Recipes';

    var $uses        = array('Recipe', 'User');
    var $helpers     = array('Html', 'Ajax');
    var $components  = array('Session', 'Email');
}

?>
```

When defining these attributes, make sure to include the default classes (like including the HtmlHelper in the $helpers array, for example) if you intend to use them.

Page-related Attributes: $layout and $pageTitle

A few attributes exist in CakePHP controllers that give you control over how your views set inside of a layout.

The `$layout` attribute can be set to the name of a layout saved in `/app/views/layouts`. You specify a layout by setting `$layout` equal to the name of the layout file minus the .ctp extension. If this attribute has not been defined, CakePHP renders the default layout. If you haven't defined one at `/app/views/default.ctp`, CakePHP's core default layout will be rendered.

```
<?php

class RecipesController extends AppController
{
    function quickSave()
    {
        $this->layout = 'ajax';
    }
}

?>
```

Using $layout to define an alternate layout.

The `$pageTitle` controller attribute allows you to set the title of the rendered page. In order for this to work properly, your layout needs to include the `$title_for_layout` variable, at least between <title> tags in the head of the HTML document.

Just set `$pageTitle` to a string you'd like to see in the <title> of your document.

The Parameters Attribute ($params)

Controller parameters are available at `$this->params` in your CakePHP controller. This variable is used to provide access to information about the current request. The most common usage of `$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.

```
// The HtmlHelper is used to create a form element:

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

// When rendered, it looks something like:

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

// When the form is submitted to the controller via POST,
// the data shows up in $this->data.

//The submitted first name can be found here:
$this->data['User']['first_name'];
```

$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 empty, `0` if not.

$this->params['isAjax']

Stores `1` if the current layout is set to '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 requested, `$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 requested, `$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 requested, `$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 contain:

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

Other Attributes

While you can check out the details for all controller attributes in the API, there are other controller attributes that merit their own sections in the manual.

The `$cacheAction` attribute aids in caching views, and the `$paginate` attribute is used to set pagination defaults for the controller. For more information on how to use these attributes, check out their respective sections in later on in this manual.

Controller Methods

For a complete list of controller mthods and their descriptions visit the CakePHP API. Check out http://api.cakephp.org/1.2/classController.html#_details.

Interacting with Views

set(string $var, mixed $value)

The set() method is the main way to get data from your controller to your view. Once you've used set(), the variable can be accessed in your view.

```
//First you pass data from the controller:

$this->set('color', "pink");

//Then, in the view, you can utilize the data:

<p>You have selected <?php echo $color; ?> icing for the cake.</p>
```

set() usage example

render(string $action, string $layout, string $file)

The render() method is automatically called at the end of each requested controller action. This method performs all the view logic (using the data you've given in using the set() method), places the view inside its layout and serves it back to the end user.

The default view file used by render is determined by convention. If the search() action of the RecipesController is requested, the view file in /app/views/recipes/search.ctp will be rendered.

Although CakePHP will automatically call it after every action's logic, you can use it to specify an alternate view file by specifying an action name in the controller using $action. You can also specify an alternate view file using the third parameter, $file. When using $file, don't forget to utilize a few of CakePHP's global constants (such as VIEWS).

The $layout parameter allows you to specify the layout the view is rendered in .

Flow Control

redirect(string $url, integer $status, boolean $exit)

The flow control method you'll use most often is redirect(). This method takes its first parameter in the form of a CakePHP-relative URL. When a user has successfully placed an order, you might wish to redirect them to a receipt screen.

```
function placeOrder()
{
    //Logic for finalizing order goes here

    if($success)
    {
        $this->redirect('/orders/thanks');
    }
    else
    {
        $this->redirect('/orders/confirm');
    }
}
```

redirect() usage example

The second parameter of redirect() allows you to define an HTTP status code to accompany the redirect. You may want to use 301 (moved permanently) or 303 (see other), depending on the nature of the redirect.

This method does not exit() after a redirect, so set the third parameter to true if you would rather have application execution halt after a redirect.

flash(string $message, string $url, integer $pause)

Similarly, the flash() method is used to direct a user to a new page after an operation. The flash() method is different in that it shows a message before passing the user on to another URL.

The first parameter should hold the message to be displayed, and the second parameter is a CakePHP-relative URL. CakePHP will display the $message for $pause seconds before forwarding the user on.

For in-page flash messages, be sure to check out SessionComponent's setFlash() method.

Callbacks

CakePHP controllers come fitted with callbacks you can use to insert logic just before or after controller actions are rendered.

beforeFilter()

This function is executed before every action in the controller. Its a handy place to check for an active session or inspect user permissions.

beforeRender()

Called after controller action logic, but before the view is rendered. This callback is not used often, but may be needed if you are calling render() manually before the end of a given action.

afterFilter()

Called after every controller action.

afterRender()

Called after an action has been rendered.

Other Useful Methods

constructClasses()

This method loads the models required by the controller. This loading process is done by CakePHP normally, but this method is handy to have when accessing controllers from a different perspective. If you need CakePHP in a command-line script or in your Flash AMF Remoting requests, constructClasses() may come in handy.

referrer()

Returns the referring URL for the current request.

disableCache()

Used to tell the user's **browser** not to cache the results of the current request. This is different than view caching, covered in a later chapter.

postConditions(array $data, mixed $op, string $bool, boolean $exclusive)

Use this method to turn a set of POSTed model data (from HtmlHelper-compatible inputs) into set of find conditions for a model. This function offers a quick shortcut on building search logic. For example, an administrative user may want to be able to search orders in order to know which items need to be shipped. You can use

CakePHP's Form- and HtmlHelpers to create a quick form based on the Order model. Then a controller action can use the data posted from that form to craft find conditions:

```
function index()
{
    $o = $this->Orders->findAll($this->postConditions($this->data));
    $this->set('orders', $o);
}
```

If `$this->data['Order']['referrer']` equals "Old Towne Bakery", postConditions converts that condition to an array compatible for use in a `Model->findAll()` method. In this case, array("Order.referrer" => "Old Towne Bakery").

If you want use a different SQL operator between terms, supply them using the second parameter.

```
/*
Contents of $this->data
array(
    'Order' => array(
        'num_items' => '4',
        'referrer' => 'Ye Olde'
    )
)
*/

//Let's get orders that have at least 4 items and contain 'Ye Olde'
$o = $this->Order->findAll($this->postConditions(
    $this->data,
    array('>=', 'LIKE')
));
```

The key in specifying the operators is the order of the columns in the $this->data array. Since num_items is first, the >= operator applies to it.

The third parameter allows you to tell CakePHP what SQL boolean operator to use between the find conditions. String like 'AND', 'OR' and 'XOR' are all valid values.

Finally, if the last parameter is set to true, and the $op parameter is an array, fields not included in $op will not be included in the returned conditions.

cleanUpFields(string $modelClass = null)

This convenience method concatenates date parts in $this->data back together before storage. If you have Form- or HtmlHelper date inputs, this method concatentates the year, month, day and time into a more storage-compatible string.

This method uses the controller's default model (i.e., the Cookie model for the CookiesController) as the target for the concatenation, but an alternate class can be supplied using the first parameter.

paginate()

This method is used for paginating results fetched by your models. You can specify page sizes, model find conditions and more. Details on this method follow later. Check out the pagination chapter later on in this manual.

requestAction(string $url, array $options)

This function calls a controller's action from any location and returns data from the action. The $url passed is a CakePHP-relative URL (/controllername/actionname/params). If the $options array includes a 'return' key, AutoRender is automatically set to true for the controller action, having requestAction hand you back a fully rendered view.

Note: while you can use requestAction() to retrieve a fully rendered view, the performance hit you take on running through the whole view layer another time isn't often worth it. The requestAction() method is best used in conjunction with elements–as a way to fetch business logic for an element before rendering.

First, let's look at how to get data from a controller action. First, we need to set up a controller action that returns some data we might need in various places throughout the applicaiton:

```
// Here is our simple controller:

class UsersController extends AppController
{
    function getUserList()
    {
        return $this->User->findAll("User.active = 1");
    }
}
```

Imagine that we needed to create a simple table showing the active users in the system. Instead of duplicating list-generating 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 element in your application that is not static, you might want to use requestAction() to grab controller-like logic for the element as you inject it into your views. If you have created a controller action that supplies the logic needed, you can grab that data and pass it to the second parameter of the view's renderElement() method using requestAction().

```
<?php
echo $this->renderElement(
    ‘users’,
    $this->requestAction(‘/users/getUserList’)
);
?>
```

If the `$options` array contains a ‘return’ key, the controller action is rendered inside an empty layout and returned. In this way, 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.

Components

Introduction

Components are packages of logic that are shared between controllers. If you find yourself wanting to copy and paste things between controllers, you might consider wrapping some functionality in a component.

CakePHP also comes with a fantastic set of core components you can use to aid in:

- Security
- Sessions
- Access control lists
- Emails
- Cookies
- Authentication
- Request handling

Each of these core components are detailed in their own chapters. For now, we'll show you how to create your own components. Creating components keeps controller code clean and allows you to reuse code between projects.

Creating Your Own

Suppose our online application needs to interface with a legacy system that tracks inventory in our brick-and-mortar stores. Let's create an example component that connects to the inventory management system and checks the inventory for a given product (specified by a product ID).

The first step is to create a new component file and class. Create the file in `/app/controllers/components/inventory.php`. The basic structure for the component would look something like this.

```
<?php

class InventoryComponent extends Object
{
    function getInStock($pid)
    {
        //Do inventory logic here...

        return $in_stock;
    }
}

?>
```

inventory.php, Bare-bones component example

Once our component is finished, we can use it in the application's controllers by placing the components name in the controller's `$components` arrays.

```
//Make the new component available at $this->Inventory

var $components = ('Inventory', 'Session');
```

MVC Class Access Within Components

To get access to the controller instance from within your newly created component, you'll need to implement the startup() method. This special method takes a reference to the controller as its first parameter, and is automatically called after beforeFilter() in the controller. This allows you to set component properties in the beforeFilter method, which the component can act on in it's startup() method.

To get access to models inside a component, just create new instances manually. For this component, we might need access to a Product model to do lookups.

```
<?php

class InventoryComponent extends Object
{

    function startup(&$controller)
    {
        //Set up a model instance for this component
        $this->Product &= new Product();
    }

    function getInStock($pid)
    {
        //Do inventory logic here...
    }
}

?>
```

You might also want to utilize other components inside a custom component. To do so, just create a $components class variable (just like you would in a controller) as an array that holds the names of components you wish to utilize.

Models

Introduction

Models are used in CakePHP applications for data access. Models most commonly represent a database table, though they can be used to access files, LDAP records, iCal events, or rows in a CSV file.

Models can also be associated in order to make data access easier. If you find that every time you fetch data with your Recipe model, you'd also like to fetch its related Author and Ingredient entries, model associations can help.

New in CakePHP is DataSources and Behaviors. DataSources form a unified foundation for CakePHP models. This allows for greater model interaction, including associations. Model behaviors provide a way to mix in functionality. Specifying behaviors for models gives you access to commonly needed data manipulation logic: things like tree pruning, upload handling, and more.

Fully functional model classes can look pretty bare:

```
<?php

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

?>
```

With just a few lines of code, you have full CRUD (create, retrieve, update, delete) functionality. Most of this comes from the model's grandparent class Model. Once you have a model defined, you can utilize it in your controller. Here's a basic example of a model being used (`$this->Ingredient`) to fetch a list of Ingredient records to hand to the view:

```
<?php

class IngredientsController extends AppController
{

    function index()
    {
        //Hand a list of all ingredients to the view:
        $this->set(
            'ingredients',
            $this->Ingredient->findAll();
        )
    }
}

?>
```

ingredients_controller.php: basic model usage example

Using models to fetch and save data can save you serious headaches. CakePHP models provide a standard and centralized way to approach your data store, while at the same time offering a level of security for your application. @@@

Model classes can be expanded to include a number of attributes and methods to enhance their functionality. These attributes and methods will be explained next.

Model Attributes

For a complete list of model attributes and their descriptions visit the CakePHP API. Check out http://api.cakephp.org/1.2/classModel.html.

$useDbConfig

The name of the database configuration this model uses. Configurations are specified in `/app/config/database.php` as class variables. Defaults to 'default.'

$useTable

The name of the database table this model uses. The default table used is the lower-cased, plural version of the model's classname. Set this attribute to the name of an alternate table, or set it to *false* if you wish the model to have no database relation at all.

$primaryKey

If the model is tied to a database table, this attribute holds the name of the column of the tables primary key. Defaults to 'id'.

$displayField

This attribute is used to define the field the model will use for display in scaffolding elements. When associated data is presented (i.e., which Ingredient you'd like to associate to a Recipe) scaffolding will use the row's ID, 'name', or 'title' fields unless $displayField has been specified.

$recursive

An integer specifying the number of levels you wish CakePHP to fetch associated model data in find() and findAll() operations.

Imagine your application features Groups which have many Users which in turn have many Articles. You can set $recursive to different values based on the amount of data you want back from a $this->Group->find() call:

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

Set it no higher than you need–having CakePHP fetch data you aren't going to use unnecessarily slows your app.

$order

The default ordering of data for any find operation. Possible values include:

$order = "field"

$order = "Model.field";

$order = "Model.field asc";

$order = "Model.field ASC";

$order = "Model.field DESC";

$order = array("Model.field" => "asc", "Model.field2" => "DESC");

$id

Value of the primary key ID of the record that this model is currently pointing to. If you are saving model data inside of a loop, set `$id` to null (`$this->ModelName->id = null`) in order to ready the model class for the next set of data.

$data

The container for the model's fetched data. While data returned from a model class is normally used as returned from a `find()` call, you may need to access information stored in `$data` inside of model callbacks.

$_tableInfo

Contains metadata describing the model's database table fields. Each field is described by:

- name
- type (integer, string, datetime, etc.)
- null
- default value
- length

$validate

This attribute holds rules that allow the model to make data validation decisions before saving. Keys named after fields hold regex values allowing the model to try to make matches.

For more information on validation, see the Data Validation chapter later on in this manual.

$name

The name of this model. CakePHP model names are CamelCased and singular. CakePHP users on PHP4 should always declare $name in order to avoid the quirks involved with PHP4 class-naming.

$cacheQueries

If set to true, data fetched by the model during a single request is cached. This caching is in-memory only, and only lasts for the duration of the request. Any duplicate requests for the same data is handled by the cache.

$belongsTo, $hasOne, $hasMany, $hasAndBelongsToMany

These model attributes will be covered later in this chapter. They hold arrays that define how this model is related to others.

$actsAs

This attribute is also covered later on. The $actsAs attribute stores the names of Behaviors this model uses.

Model Methods

For a complete list of model methods and their descriptions visit the CakePHP API. Check out http://api.cakephp.org/1.2/classModel.html.

Retrieving Your Data

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 $page (default is page 1). The `$conditions` should be formed just as

they would in an SQL statement: `$conditions = "Pastry.type LIKE '%cake%' AND Pastry.created_on > '2007-01-01'"`, for example. Prefixing conditions with the model's name ('Pastry.type' rather that just 'type') is always a good practice, especially when associated data is being fetched in a query.

Setting the $recursive parameter to an integer forces `findAll()` to fetch data according to the behavior described in the Model Attributes `$recursive` section outlined earlier.

Data from findAll() is returned in an array, following this basic format:

```
Array
(
    [0] => Array
        (
            [ModelName] => Array
                (
                    [id] => 83
                    [field1] => value1
                    [field2] => value2
                    [field3] => value3
                )

            [AssociatedModelName] => Array
                (
                    [id] => 1
                    [field1] => value1
                    [field2] => value2
                    [field3] => value3
                )
        )
    [1] => Array
        (
            [ModelName] => Array
                (
                    [id] => 85
                    [field1] => value1
                    [field2] => value2
                    [field3] => value3
                )

            [AssociatedModelName] => Array
                (
                    [id] => 2
                    [field1] => value1
                    [field2] => value2
                    [field3] => value3
                )
        )
)
```

find(string $conditions, array $fields, string $order, int $recursive)

Same as in findAll(), except that find only returns the first record matching the supplied `$conditions`.

findAllBy<fieldName>(string $value)

findBy<fieldName>(string $value);

These magic functions can be used as a shortcut to search your tables by a certain field. Just add the name of the field, in CamelCase format, to the end of these functions, and supply the criteria for that field as the first parameter.

`findAllBy<x> Example`	Corresponding SQL Fragment
$this->Product->findAllByOrderStatus('3');	Product.order_status = 3
$this->Recipe->findAllByType('Cookie');	Recipe.type = 'Cookie'
$this->User->findAllByLastName('Anderson');	User.last_name = 'Anderson'
$this->Cake->findById(7);	Cake.id = 7
$this->User->findByUserName('psychic');	User.user_name = 'psychic'

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

findNeighbors(string $conditions, array $field, string $value)

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

This is helpful shortcut in creating '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 be able to show the image...
        $this->set('image', $this->Image->findById($id);

        // But we also want links to the previous and next images...
        $this->set(
            'neighbors',
            $this->Image->findNeighbours(null, 'id', $id);
        )
    }
}
```

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

field(string $name, string $conditions, string $order)

Returns the value of a single field, specified as `$name`, from the first record matched by `$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 an 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` specify 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->Role->generateList(
    null,
    'role_name ASC',
    null,
    '{n}.Role.id',
    '{n}.Role.role_name'
);

//This would return something like:
array(
    '1' => 'Head Honcho',
    '2' => 'Marketing',
    '3' => 'Department Head',
    '4' => 'Grunt'
);
```

Many people are a little bewildered by the '{n}' syntax used by generateList(). Fret not, for it serves as a place holder for switching between model DataSources, covered later on in this chapter.

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 supply no return value).

If you're ever using custom SQL queries in your application, be sure to check out CakePHP's Sanitize library (covered later in this manual), which aids in cleaning up user-provided data from injection and cross-site scripting attacks.

Complex Find Conditions

Most of the model's find 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. If you find yourself needing 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 CakePHP 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:

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

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

The structure here is fairly self-explanatory: it will find any post where the title equals "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")
```

Notice the '<>' that prefixes the expression. CakePHP 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 and the 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, CakePHP 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 nest-able. Let's say you had a hasMany/belongsTo relationship between Posts and Authors, which would result in a LEFT JOIN. Let's say you wanted to find all the posts that contained a certain keyword ("magic") 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

CakePHP makes saving model data a snap. Data ready to be saved should be passed to the model's save() method using the following basic format:

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

Most of the time you won't even need to worry about this format: CakePHP's HtmlHelper, FormHelper, and find methods all package data in this format. If you're using either of the helpers, the data is also conveniently available in `$this->data` for quick usage.

Here's a quick example of a controller action that uses a CakePHP model to save data to a database table:

```
function edit($id)
{
    //Has any form data been POSTed?
    if(!empty($this->data))
    {
        //If the form data can be validated and saved...
        if($this->Recipe->save($this->data['Recipe']))
        {
            //Set a session flash message and redirect.
            $this->Session->setFlash("Recipe Saved!");
            $this->redirect('/recipes');
            exit(0);
        }
    }

    //If no form data, find the recipe to be edited
    //and hand it to the view.
    $this->set('recipe', $this->Recipe->findById($id);
}
```

One additional note: when save is called, the data passed to it in the first parameter is validated using CakePHP validation mechanism (see the Data Validation chapter for more information). If for some reason your data isn't saving, be sure to check to see if some validation rules aren't being broken.

There are a few other save-related methods in the model that you'll find useful:

save(array $data = null, boolean $validate = true, array $fieldList = array())

Featured above, this method saves array-formatted data. The second parameter allows you to sidestep validation, and the third allows you to supply a list of model fields to be saved. For added security, you can limit the saved fields to those listed in `$fieldList`.

create(array $data = array())

This method initializes the model class for saving new information. If you're ever calling `save()` inside of a loop (in order to create many records at once), make sure to call `create()` just before `save()`.

If you supply the `$data` parameter (using the array format outlined above), the newly created model will be ready to save with that data (accessible at `$this->data`).

saveField(string $fieldName, string $fieldValue, $validate = false)

Used to save a single field value. Set the ID of the model (`$this->ModelName->id = $id`) just before calling `saveField()`.

updateAll(array $conditions, array $fields)

Updates many records in a single call. Records to be updated are identified by the $conditions array, and fields to be updated, along with their values, are identified by the $fields array.

For example, if I wanted to approve all bakers who have been members for over a year, the update call might look something like:

```
$this_year = date('Y-m-d h:i:s', strtotime('-1 year'));

$this->Baker->updateAll(
    array('Baker.created' => "<= $this_year"),
    array('Baker.approved' => true)
);
```

remove(int $id = null, boolean $cascade = true);

del(int $id = null, boolean $cascade = true);

Deletes the record identified by `$id`. By default, also deletes records dependent on the record specified to be deleted.

For example, when deleting a User record that is tied to many Recipe records:

- if $cascade is set to true, the related Recipe records are also deleted.
- if $cascade is set to false, the Recipe records will remain after the User has been deleted.

deleteAll(mixed $conditions, $cascade = true)

Same as with `del()` and `remove()`, except that `deleteAll()` deletes all records that match the supplied conditions. The `$conditions` array should be supplied as an SQL fragment or array.

Model Callbacks

If you want to sneak in some logic just before or after a CakePHP model operation, use model callbacks. These functions can be defined in model classes (including your

AppModel) class. Be sure to note the expected return values for each of these special functions.

beforeFind(mixed $queryData)

Called before any find-related operation. The `$queryData` passed to this callback contains information about the current query: conditions, fields, etc.

If you do not wish the find operation to begin (possibly based on a decision relating to the `$queryData` options), return *false*.

You might use this callback to restrict find operations based on a user's role, or make caching decisions based on the current load.

afterFind(array $results)

Use this callback to modify results that have been returned from a find operation, or to perform any other post-find logic. The `$results` parameter passed to this callback contains the returned results from the model's find operation

The return value for this callback should be the (possibly modified) results for the find operation that triggered this callback.

This callback might be used to pretty up date or currency formats.

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 the current `save()` execution will abort.

beforeSave()

Place any pre-save logic in this function. This function executes immediately after model data has been successfully validated, but just before the data is saved. This function should also return *true* if you want the save operation to continue.

This callback is especially handy for any data-massaging logic that needs to happen before your data is stored. If your storage engine needs dates in a specific format, access it at `$this->data` and modify it.

afterSave()

If you have logic you need to be executed just after every save operation, place it 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.

User-Defined Functions

While CakePHP's model functions should get you where you need to go, don't forget that model classes are just that: classes that allow you to write your own methods.

Any operation that handles the saving and fetching of data is best housed in your model classes.

Associations

Introduction

One of the most powerful features of CakePHP is the relational mapping provided by the model. In CakePHP, the links between models are handled through associations.

Defining relations between different objects in your application should be a natural process. For example: in a recipe database, a recipe may have many reviews, reviews have a single author, and authors may have many recipes. Defining the way these relations work allows you to access your data in an intuitive and powerful way.

The purpose of this section is to show you how to plan for, define, and utilize associations between models in CakePHP.

Because the most common form of storage in web applications is a relational database, most of what we'll cover in this manual will be in a database related context.

The four association types in CakePHP are: hasOne, hasMany, belongsTo, and hasAndBelongsToMany (HABTM).

Association Type	Example
hasOne	A user has one profile.
hasMany	Users in a system can have multiple recipes.
belongsTo	A recipe belongs to a user.
hasAndBelongsToMany	Recipes have, and belong to many tags.

Associations are defined by creating a class variable named after the association you are defining. The class variable can sometimes be as simple as a string, but can be as complete as a multidimensional array used to define association specifics.

```
<?php

class User extends AppModel
{
    var $name = 'User';
    var $hasOne = 'Profile';
    var $hasMany = array(
        'Recipe' => array(
            'className'  => 'Recipe',
            'conditions' => 'Recipe.approved = 1',
            'order'      => 'Recipe.created DESC'
        )
    );
}

?>
```

hasOne

Let's set up a User model with a hasOne relationship to a Profile model.

First, your database tables need to be keyed correctly. For a hasOne relationship to work, one table has to contain a foreign key that points to a record in the other. In this case the `profiles` table will contain a field called `user_id`. The basic pattern is:

```
hasOne: the *other* model contains the foreign key.

Apple hasOne Banana      => bananas.apple_id
User hasOne Profile      => profiles.user_id
Doctor hasOne Mentor     => mentors.doctor_id
```

The User model file will be saved in `/app/models/user.php`. To define the 'User hasOne Profile' association, add the `$hasOne` property to the model class. Remember to have a Profile model in /app/models/profile.php, or the association won't work.

```
<?php

class User extends AppModel
{
    var $name = 'User';
    var $hasOne = 'Profile';
}

?>
```

There are two ways to describe this relationship in your model files. The simplest method is to set the `$hasOne` attribute to a string containing the classname of the associated model, as we've done above.

If you need more control, you can define your associations using array syntax. For example, you might want to sort related rows in descending order according to a date, or you might want to limit the association to include only certain records.

```
<?php

class User extends AppModel
{
    var $name = 'User';
    var $hasOne = array(
        'Profile' => array(
            'className'    => 'Profile',
            'conditions'   => 'Profile.published = 1',
            'dependent'    => true
        )
    );
}

?>
```

Possible keys for hasOne association arrays include:

- **className**: the classname of the model being associated to the current model. If you're defining a 'User hasOne Profile' relationship, the className key should equal 'Profile.'

- **foreignKey**: the name of the foreign key found in the other model. This is especially handy if you need to define multiple hasOne relationships. The default value for this key is the underscored, singular name of the other model, suffixed with '_id'.
- **conditions**: An SQL fragment used filter related model records. It's good practice to use model names in SQL fragments: "Profile.approved = 1" is always better than just "approved = 1."
- **fields**: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.
- **dependent**: When the dependent key is set to true, and the model's `delete()` method is called with the cascade parameter set to true, associated model records are also deleted. In this case we set it true so that deleting a User will also delete her associated Profile.

Once this association has been defined, find operations on the User model will also fetch a related Profile record if it exists:

```
//Sample results from a $this->User->find() call.

Array
(
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Profile] => Array
        (
            [id] => 12
            [user_id] => 121
            [skill] => Baking Cakes
            [created] => 2007-05-01 10:31:01
        )
)
```

belongsTo

Now that we have Profile data access from the User model, let's define a belongsTo association in the Profile model in order to get access to related User data. The belongsTo association is a natural complement to the hasOne and hasMany associations: it allows us to see the data from the other direction.

When keying your database tables for a belongsTo relationship, follow this convention:

```
belongsTo: the *current* model contains the foreign key.

Banana belongsTo Apple      => bananas.apple_id
Profile belongsTo User      => profiles.user_id
Mentor belongsTo Doctor     => mentors.doctor_id

If a table contains a foreign key, it belongsTo.
```

We can define the belongsTo association in our Profile model at /app/models/profile.php using the string syntax as follows:

```
<?php

class Profile extends AppModel
{
    var $name = 'Profile';
    var $hasOne = 'User';
}

?>
```

We can also define a more specific relationship using array syntax:

```
<?php

class Profile extends AppModel
{
    var $name = 'Profile';
    var $belongsTo = array(
        'User' => array(
            'className'    => 'User',
            'foreignKey'    => 'user_id'
        )
    );
}

?>
```

Possible keys for belongsTo association arrays include:

- **className**: the classname of the model being associated to the current model. If you're defining a 'Profile belongsTo User' relationship, the className key should equal 'User.'
- **foreignKey**: the name of the foreign key found in the other model. This is especially handy if you need to define multiple belongsTo relationships. The default value for this key is the underscored, singular name of the other model, suffixed with '_id'.
- **conditions**: An SQL fragment used filter related model records. It's good practice to use model names in SQL fragments: "User.active = 1" is always better than just "active = 1."
- **fields**: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.

Once this association has been defined, find operations on the Profile model will also fetch a related User record if it exists:

```
//Sample results from a $this->Profile->find() call.

Array
(
   [Profile] => Array
        (
            [id] => 12
            [user_id] => 121
            [skill] => Baking Cakes
            [created] => 2007-05-01 10:31:01
        )
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
)
```

hasMany

Next step: defining a "User hasMany Comment" association. A hasMany associatoin will allow us to fetch a user's comments when we fetch a User record.

When keying your database tables for a hasMany relationship, follow this convention:

```
hasMany: the *other* model contains the foreign key.

User hasMany Comment     => Comment.user_id
Cake hasMany Virtue      => Virtue.cake_id
Product hasMany Option   => Option.product_id
```

We can define the hasMany association in our User model at /app/models/user.php using the string syntax as follows:

```
<?php

class User extends AppModel
{
    var $name = 'User';
    var $hasOne = 'Comment';
}

?>
```

We can also define a more specific relationship using array syntax:

```
<?php

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

?>
```

Possible keys for hasMany association arrays include:

- **className**: the classname of the model being associated to the current model. If you're defining a 'User hasMany Comment' relationship, the className key should equal 'Comment.'
- **foreignKey**: the name of the foreign key found in the other model. This is especially handy if you need to define multiple hasMany relationships. The default value for this key is the underscored, singular name of the other model, suffixed with '_id'.
- **conditions**: An SQL fragment used filter related model records. It's good practice to use model names in SQL fragments: "Comment.status = 1" is always better than just "status = 1."
- **fields**: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.
- **order**: An SQL fragment that defines the sorting order for the returned associated rows.
- **limit**: The maximum number of associated rows you want returned.
- **offset**: The number of associated rows to skip over (given the current *conditions* and *order*) before fetching and associating.
- **dependent**: When dependent is set to *true,* recursive model deletion is possible. In this example, Comment records will be deleted when their associated User record has been deleted.
- *Note:* the second parameter of the `Model->delete()` method must be set to true in order for recursive deletion to occur.
- **finderQuery**: A complete SQL query CakePHP can use to fetch associated model records. This should be used in situations that require very custom results.

Once this association has been defined, find operations on the User model will also fetch related Comment records if they exist:

```
//Sample results from a $this->User->find() call.

Array
(
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Comment] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [user_id] => 121
                    [title] => On Gwoo the Kungwoo
                    [body] => The Kungwooness is not so Gwooish
                    [created] => 2006-05-01 10:31:01
                )
            [1] => Array
                (
                    [id] => 123
                    [user_id] => 121
                    [title] => More on Gwoo
                    [body] => But what of the ‘Nut?
                    [created] => 2006-05-01 10:41:01
                )
        )
)
```

One thing to remember is that you'll need a complimentary Comment belongsTo User association in order to get the data from both directions. What we've outlined in this section empowers you to get Comment data from the User. Adding the Comment belongsTo User association in the Comment model empowers you to get User data from the Comment model - completing the connection and allowing the flow of information from either model's perspective.

hasAndBelongsToMany (HABTM)

Alright. At this point, you can already call yourself a CakePHP model associations professional. You're already well versed in the three associations that take up the bulk of object relations.

Let's tackle the final relationship type: hasAndBelongsToMany, or HABTM. This association is used when you have two models that need to be joined up, repeatedly, many times, in many different ways.

The main difference between hasMany and HABTM is that a link between models in HABTM is not exclusive. For example, we're about to join up our Recipe model with a Tag model using HABTM. Attaching the "Italian" tag to my grandma's Gnocci recipe doesn't "use up" the tag. I can also tag my Honey Glazed BBQ Spaghettio's with "Italian" if I want to.

Links between hasMany associated objects *are* exclusive. If my User hasMany Comments, a comment is *only* linked to a specific user. It's no longer up for grabs.

Moving on. We'll need to set up an extra table in the database to handle HABTM associations. This new join table's name needs to include the names of both models involved, in alphabetical order. The contents of the table should be at least two fields, each foreign keys (which should be integers) pointing to both of the involved models.

```
HABTM: Requires a separate join table that includes both model names.

Note: table name must be in alphabetical order.

Recipe HABTM Tag  => recipes_tags.recipe_id, recipes_tags.tag_id
Cake HABTM Fan    => cakes_fans.cake_id, cakes_fans.fan_id
Foo HABTM Bar     => bars_foos.foo_id, bars_foos.bar_id
```

Once this new table has been created, we can define the HABTM association in the model files. We're gonna skip straight to the array syntax this time:

```
<?php

class Recipe extends AppModel
{
    var $name = 'Recipe';
    var $hasAndBelongsToMany = array(
        'Tag' =>
            array('className'           => 'Tag',
                'joinTable'             => 'posts_tags',
                'foreignKey'            => 'tag_id',
                'associationForeignKey' => 'recipe_id',
                'conditions'            => '',
                'order'                 => '',
                'limit'                 => '',
                'uniq'                  => true,
                'finderSql'             => '',
                'deleteQuery'           => '',
                'counterSql'            => ''
            )
        );
}

?>
```

Possible keys for hasMany association arrays include:

- **className**: the classname of the model being associated to the current model. If you're defining a 'User hasMany Comment' relationship, the className key should equal 'Comment.'
- **joinTable**: The name of the join table used in this association (if the current table doesn't adhere to the naming convention for HABTM join tables).
- **foreignKey**: the name of the foreign key found in the *other* model. This is especially handy if you need to define multiple HABTM relationships. The default value for this key is the underscored, singular name of the other model, suffixed with '_id'.
- **associationForeignKey**: the name of the foreign key found in the *current* model. This is especially handy if you need to define multiple HABTM relationships. The default value for this key is the underscored, singular name of the current model, suffixed with '_id'.
- **conditions**: An SQL fragment used filter related model records. It's good practice to use model names in SQL fragments: "Comment.status = 1" is always better than just "status = 1."

- **fields**: A list of fields to be retrieved when the associated model data is fetched. Returns all fields by default.
- **order**: An SQL fragment that defines the sorting order for the returned associated rows.
- **limit**: The maximum number of associated rows you want returned.
- **offset**: The number of associated rows to skip over (given the current *conditions* and *order*) before fetching and associating.
- **finderQuery**: A complete SQL query CakePHP can use to fetch associated model records. This should be used in situations that require very custom results.

Once this association has been defined, find operations on the Recipe model will also fetch related Tag records if they exist:

```
//Sample results from a $this->Recipe->find() call.

Array
(
    [Recipe] => Array
        (
            [id] => 2745
            [name] => Chocolate Frosted Sugar Bombs
            [created] => 2007-05-01 10:31:01
            [user_id] => 2346
        )
    [Tag] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [name] => Breakfast
                )
           [1] => Array
                (
                    [id] => 123
                    [name] => Dessert
                )
           [2] => Array
                (
                    [id] => 123
                    [name] => Heart Disease
                )
        )
)
```

Remember to define a HABTM association in the Tag model if you'd like to fetch Recipe data when using the Tag model.

Saving Related Model Data (hasOne, hasMany, belongsTo)

When working with associated models, it is important to realize that saving model data should always be done by the corresponding CakePHP 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 model records exists in the system yet (for example, you want to save a new User and their related Profile records 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 UsersController that handles the saving of a new User and a related Profile. The example action shown below will assume that you've POSTed enough data (using the FormHelper) to create a single User and a single Profile.

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

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

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

        $user_id = $this->User->getLastInsertId();

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

        $this->data['Profile']['user_id'] = $user_id;

        //Because our User hasOne Profile, we can access
        //the Profile model through the User model:

        $this->User->Profile->save($this->data);

    }
}
```

As a rule, when working with hasOne, hasMany, and belongsTo associations, its all about keying. The basic idea is to get the key from one model and place it in the foreign key field on the other. Sometimes this might involve using getLastInsertId() after a save(), but other times it might just involve gathering the ID from a hidden input on a form that's just been POSTed to a controller action.

Saving Related Model Data (HABTM)

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 HABTM, it's 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 Tag to Recipe. Let's now create a form that creates recipes, 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.

Let's just start out with the part of the form that creates a recipe:

```
//@@@
```

Creating and Destroying Associations on the Fly

asdf

DataSources

asdf

Behaviors

asdf