CakePHP 1.3 Manual

Prepared by Andre C Santiago

www.caboone.com / www.bravanews.com

admin@caboone.com

Table of Contents

1. Beginning With CakePHP

Welcome to the Cookbook, the manual for the CakePHP web application framework that makes developing a piece of cake!

This manual assumes that you have a general understanding of PHP and a basic understanding of object-oriented programming (OOP). Different functionality within the framework makes use of different technologies – such as SQL, JavaScript, and XML – and this manual does not attempt to explain those technologies, only how they are used in context.

1.1 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 wheel-reinventing, 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 versions 4 and 5 of PHP
- Integrated CRUD for database interaction
- Application scaffolding
- Code generation
- 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 ACL
- Data Sanitization
- Flexible Caching
- Localization
- Works from any web site directory, with little to no Apache configuration involved

1.2 Where to Get Help

 # The Official CakePHP website

http://www.cakephp.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.

 # The Cookbook

http://book.cakephp.org

This manual 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.

 # The Bakery

http://bakery.cakephp.org

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.

 # The API

http://api.cakephp.org/

Straight to the point and straight from the core developers, the CakePHP API (Application Programming Interface) is the most comprehensive documentation around for all the nitty gritty details of the internal workings of the framework. Its a straight forward code reference, so bring your propeller hat.

 # CakeForge

http://www.cakeforge.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.

 # The Test Cases

http://api.cakephp.org/tests

If you ever feel the information provided in the API is not sufficient, check out the code of the test cases provided with CakePHP 1.3. They can serve as practical examples for function and data member usage for a class. To get the core test cases you need to download or checkout 1.3 branch from a git repository. The test cases will be located under

```
1.      cake/tests/cases
```

 # The IRC channel

IRC Channels on irc.freenode.net:

- #cakephp -- General Discussion
- #cakephp-docs -- Documentation
- #cakephp-bakery -- Bakery

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.

 # The Google Group

http://groups.google.com/group/cake-php

CakePHP also has a very active Google Group. It can be a great resource for finding archived answers, frequently asked questions, and getting answers to immediate problems.

1.3 Understanding Model-View-Controller

CakePHP follows the MVC software design pattern. Programming using MVC separates your application into three main parts:

1. The Model represents the application data
2. The View renders a presentation of model data
3. The Controller handles and routes requests made by the client

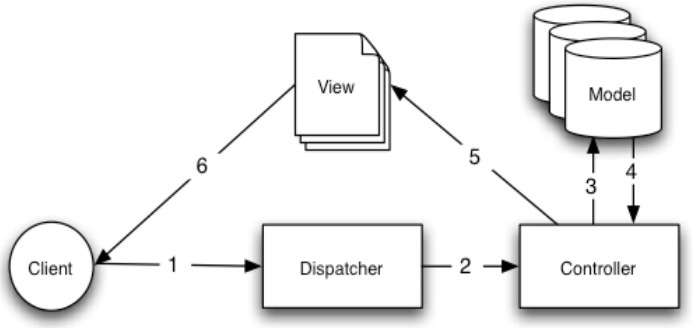

Figure: 1: A Basic MVC Request

Figure: 1 shows an example of a bare-bones MVC request in CakePHP. To illustrate, assume a client named "Ricardo" just clicked on the "Buy A Custom Cake Now!" link on your application's home page.

- Ricardo clicks the link pointing to http://www.example.com/cakes/buy, and his browser makes a request to your web server.
- The dispatcher checks the request URL (/cakes/buy), and hands the request to the correct controller.
- The controller performs application specific logic. For example, it may check to see if Ricardo has logged in.
- The controller also uses models to gain access to the application's data. Models usually 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.
- 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 client. Views in CakePHP are usually in HTML format, but a view could just as easily be a PDF, XML document, or JSON object depending on your needs.
- 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 add some details later on which are specific to CakePHP, so keep this in mind as we proceed.

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 time getting used to, but we're confident that once you've built your first application using CakePHP, you won't want to do it any other way.

2 Basic Principles of CakePHP

The CakePHP framework provides a robust base for your application. It can handle every aspect, from the user's initial request all the way to the final rendering of a web page. And since the framework follows the principles of MVC, it allows you to easily customize and extend most aspects of your application.

The framework also provides a basic organizational structure, from filenames to database table names, keeping your entire application consistent and logical. This concept is simple but powerful. Follow the conventions and you'll always know exactly where things are and how they're organized.

2.1 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. Components, Behaviors, and Helpers are classes that provide extensibility and reusability to quickly add functionality to the base MVC classes in your applications. Right now we'll stay at a higher level, so look for the details on how to use these tools later on.

2.1.1 Controller Extensions ("Components")

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
- `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.

2.1.2 View Extensions ("Helpers")

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 are used repeatedly. 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.

2.1.3 Model Extensions ("Behaviors")

Similar to Controllers and Helpers, "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, such as allowing 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. You can find the details in the models chapter.

2.1.4 Application Extensions

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

Although they aren't classes or files, 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 accomplishes a specific purpose that can span multiple applications. A user management system or a simplified blog might be a good fit for CakePHP plugins.

2.2 A Typical CakePHP Request

We've covered the basic ingredients in CakePHP, so let's look at how objects work 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.

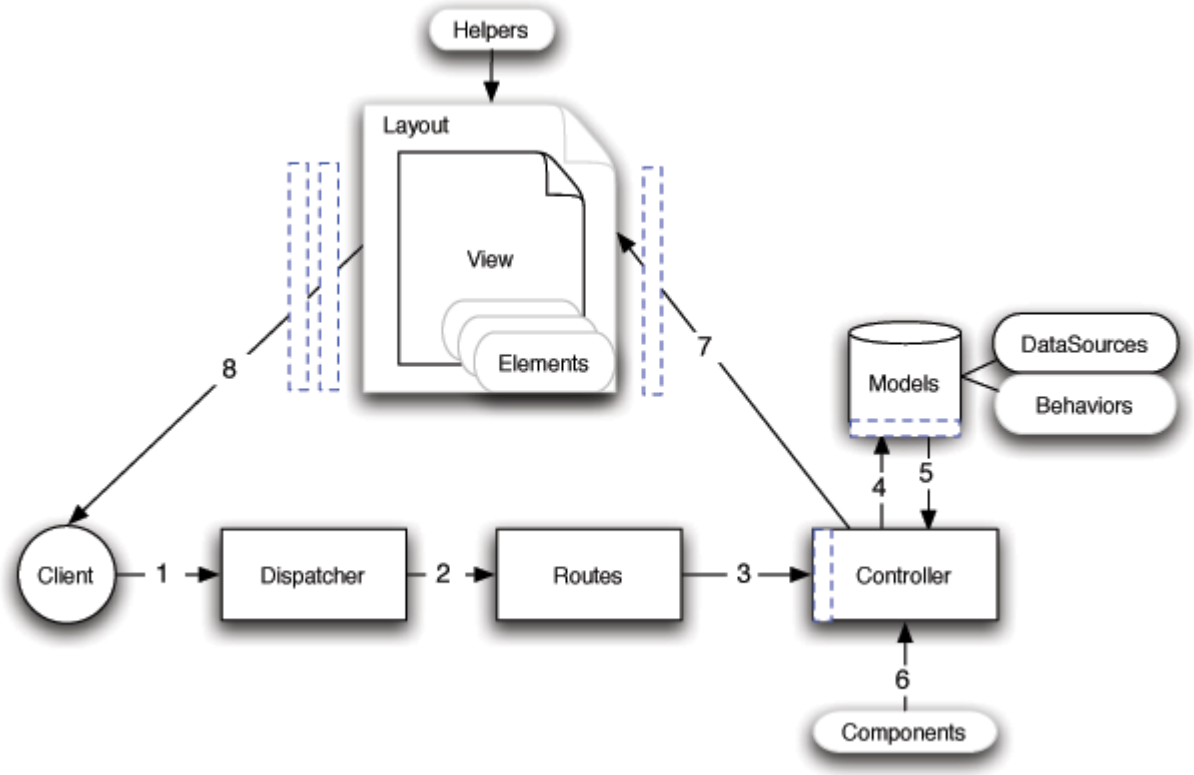

Figure: 2. Typical Cake Request.

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 Router parses the URL in order to extract the parameters for this request: the controller, action, and any other arguments that will affect the business logic during this request.
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, authentication, 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 (like afterFilter) may be applied. The complete, rendered view code is sent to Ricardo's browser.

2.3 CakePHP Folder Structure

After you've downloaded and extracted CakePHP, these are the files and folders you should see:

- app
- cake
- vendors
- plugins
- .htaccess
- index.php
- README

You'll notice three 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.
- 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 at 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.
libs	Contains 1st party libraries that do not come from 3rd parties or external vendors. This allows you to separate your organization's internal libraries from vendor libraries.
locale	Stores string files for internationalization.
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. Make sure that this folder exists and that it is writable, otherwise the performance of your application will be severely impacted. In debug mode, CakePHP will warn you if it is not the case.
vendors	Any third-party classes or libraries should be placed here. Doing so makes them easy to access using the App::import('vendor', 'name') function. Keen observers will note that this seems redundant, as there is also a vendors folder at the top level of our directory structure. We'll get into the differences between the two when we discuss managing multiple applications and more complex system setups.
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.

2.4 CakePHP Conventions

We are big fans 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 and 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 many of these tenets are easily overridden – something that is especially handy when working with legacy systems.

2.4.1 File and Classname Conventions

In general, filenames are underscored while classnames are CamelCased. So if you have a class **MyNiftyClass**, then in Cake, the file should be named **`my_nifty_class.php`**. Below are examples of how to name the file for each of the different types of classes you would typically use in a CakePHP application:

- The Controller class **KissesAndHugsController** would be found in a file named **`kisses_and_hugs_controller.php`** (notice `_controller` in the filename)
- The Component class **MyHandyComponent** would be found in a file named **`my_handy.php`**
- The Model class **OptionValue** would be found in a file named **`option_value.php`**
- The Behavior class **EspeciallyFunkableBehavior** would be found in a file named **`especially_funkable.php`**
- The View class **SuperSimpleView** would be found in a file named **`super_simple.php`**
- The Helper class **BestEverHelper** would be found in a file named **`best_ever.php`**

Each file would be located in or under (can be in a subfolder) the appropriate folder in your app folder.

2.4.2 Model and Database 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.

You can use the utility library "Inflector" to check the singular/plural of words. See the Inflector documentation for more information.

Field names with two or more words are underscored like, first_name.

Foreign keys in hasMany, belongsTo or hasOne relationships are recognized by default as the (singular) name of the related table followed by _id. So if a Baker hasMany Cake, the cakes table will refer to the bakers table via a baker_id foreign key. For a multiple worded table like category_types, the foreign key would be category_type_id.

Join tables, used in hasAndBelongsToMany (HABTM) relationships between models should be named after the model tables they will join in alphabetical order (apples_zebras rather than zebras_apples).

All tables with which CakePHP models interact (with the exception of join tables), require a singular primary key to uniquely identify each row. If you wish to model a table which does not have a single-field primary key, CakePHP's convention is that a single-field primary key is added to the table. You have to add a single-field primary key if you want to use that table's model.

CakePHP does not support composite primary keys. If you want to directly manipulate your join table data, use direct query calls or add a primary key to act on it as a normal model. E.g.:

```
CREATE TABLE posts_tags (
id INT(10) NOT NULL AUTO_INCREMENT,
post_id INT(10) NOT NULL,
tag_id INT(10) NOT NULL,
PRIMARY KEY(id));
```

Rather than using an auto-increment key as the primary key, you may also use char(36). Cake will then use a unique 36 character uuid (String::uuid) whenever you save a new record using the Model::save method.

2.4.3 Controller Conventions

Controller classnames are plural, CamelCased, and end in `Controller`. `PeopleController` and `LatestArticlesController` are both examples of conventional controller names.

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

You can also change the visibility of controller methods in CakePHP by prefixing controller method names with underscores. If a controller method has been prefixed with an underscore, the method will not be accessible directly from the web but is available for internal use. For example:

```
1.  <?php
2.  class NewsController extends AppController {
3.      function latest() {
4.          $this->_findNewArticles();
5.      }
6.
7.      function _findNewArticles() {
8.          //Logic to find latest news articles
9.      }
10. }
11. ?>
```

While the page http://www.example.com/news/latest/ would be accessible to the user as usual, someone trying to get to the page http://www.example.com/news/_findNewArticles/ would get an error, because the method is preceded with an underscore.

2.4.3.1 URL Considerations for Controller Names

As you've just seen, single word controllers map easily to a simple lower case URL path. For example, `ApplesController` (which would be defined in the file name 'apples_controller.php') is accessed from `http://example.com/apples`.

Multiple word controllers *can* be any 'inflected' form which equals the controller name so:

- `/redApples`
- `/RedApples`
- `/Red_apples`
- `/red_apples`

will all resolve to the index of the RedApples controller. However, the convention is that your urls are lowercase and underscored, therefore `/red_apples/go_pick` is the correct form to access the `RedApplesController::go_pick` action.

For more information on CakePHP URLs and parameter handling, see Routes Configuration.

2.4.4 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 that ties 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 to see how things fit together.

3 Developing with CakePHP

Now you're cooking.

3.1 Requirements

- HTTP Server. For example: Apache. 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

3.2 Installation Preparation

CakePHP is fast and easy to install. The minimum requirements are a webserver and a copy of Cake, that's it! While this manual focuses primarily on setting up with Apache (because it's the most common), you can configure Cake to run on a variety of web servers such as LightHTTPD or Microsoft IIS.

Installation preparation consists of the following steps:

- Downloading a copy of CakePHP
- Configuring your web server to handle php if necessary
- Checking file permissions

3.2.1 Getting CakePHP

There are two main ways to get a fresh copy of CakePHP. You can either download an archive copy (zip/tar.gz/tar.bz2) from the main website, or check out the code from the git repository.

To download the latest major release of CakePHP. Visit the main website http://www.cakephp.org and follow the "Download Now" link.

All current releases of CakePHP are hosted on Github. Github houses both CakePHP itself as well as many other plugins for CakePHP. The CakePHP releases are available at Github downloads.

Alternatively you can get fresh off the press code, with all the bug-fixes and up to the minute(well, to the day) enhancements. These can be accessed from github by cloning the repository. Github.

3.2.2 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.

3.3 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.

3.3.1 Development

A development installation is the fastest method to setup Cake. This example will help you install a CakePHP application and make it available at `http://www.example.com/cake_1_3/`. We assume for the purposes of this example that your document root is set to `/var/www/html`.

Unpack the contents of the Cake archive into `/var/www/html`. You now have a folder in your document root named after the release you've downloaded (e.g. `cake_1.3.0`). Rename this folder to `cake_1_3`. Your development setup will look like this on the file system:

- `/var/www/html`
    - `/cake_1_3`
        - `/app`
        - `/cake`
        - `/vendors`
        - `/.htaccess`
        - `/index.php`
        - `/README`

If your web server is configured correctly, you should now find your Cake application accessible at `http://www.example.com/cake_1_3/`.

3.3.2 Production

A production installation is a more flexible way to setup Cake. Using this method allows an entire domain to act as a single CakePHP application. This example will help you install Cake anywhere on your filesystem and make it available at `http://www.example.com`. Note that this installation may require the rights to change the `DocumentRoot` on Apache webservers.

Unpack the contents of the Cake archive into a directory of your choosing. For the purposes of this example, we assume you choose to install Cake into `/cake_install`. Your production setup will look like this on the filesystem:

- `/cake_install/`
    - `/app`
        - `/webroot` (this directory is set as the `DocumentRoot` directive)
    - `/cake`
    - `/vendors`
    - `/.htaccess`
    - `/index.php`

- /README

Developers using Apache should set the DocumentRoot directive for the domain to:

```
DocumentRoot /cake_install/app/webroot
```

If your web server is configured correctly, you should now find your Cake application accessible at http://www.example.com.

3.3.3 Advanced Installation

There may be some situations where you 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 following files.

- /app/webroot/index.php
- /app/webroot/test.php (if you use the Testing feature.)

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 (base)name 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 `/home/me/myapp`.

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:

```
1.      // /app/webroot/index.php (partial, comments removed)
2.      if (!defined('ROOT')) {
3.          define('ROOT', DS.'home'.DS.'me');
4.      }
5.      if (!defined('APP_DIR')) {
6.          define ('APP_DIR', 'myapp');
7.      }
8.      if (!defined('CAKE_CORE_INCLUDE_PATH')) {
9.          define('CAKE_CORE_INCLUDE_PATH', DS.'usr'.DS.'lib');
10.     }
```

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.

3.3.3.1 Additional Class Paths

It's occasionally useful to be able to share MVC classes between applications on the same system. If you want the same controller in both applications, you can use CakePHP's `bootstrap.php` to bring these additional classes into view.

In `bootstrap.php`, define some specially-named variables to make CakePHP aware of other places to look for MVC classes:

```
App::build(array(
    'plugins' => array('/full/path/to/plugins/', '/next/full/path/to/plugins/'),
    'models' =>  array('/full/path/to/models/', '/next/full/path/to/models/'),
    'views' => array('/full/path/to/views/', '/next/full/path/to/views/'),
    'controllers' => array('/full/path/to/controllers/', '/next/full/path/to/controllers/'),
    'datasources' => array('/full/path/to/datasources/', '/next/full/path/to/datasources/'),
    'behaviors' => array('/full/path/to/behaviors/', '/next/full/path/to/behaviors/'),
    'components' => array('/full/path/to/components/', '/next/full/path/to/components/'),
    'helpers' => array('/full/path/to/helpers/', '/next/full/path/to/helpers/'),
    'vendors' => array('/full/path/to/vendors/', '/next/full/path/to/vendors/'),
    'shells' => array('/full/path/to/shells/', '/next/full/path/to/shells/'),
    'locales' => array('/full/path/to/locale/', '/next/full/path/to/locale/'),
    'libs' => array('/full/path/to/libs/', '/next/full/path/to/libs/')
));
```

Also changed is the order in which boostrapping occurs. In the past `app/config/core.php` was loaded **after** `app/config/bootstrap.php`. This caused any `App::import()` in an application bootstrap to be un-cached and considerably slower than a cached include. In 1.3 core.php is loaded and the core cache configs are created **before** bootstrap.php is loaded.

3.3.4 Apache and mod_rewrite (and .htaccess)

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. First look at your httpd.conf (Make sure you are editing the system httpd.conf rather than a user- or site-specific httpd.conf).

1. Make sure that an .htaccess override is allowed and that AllowOverride is set to All for the correct DocumentRoot. You should see something similar to:

```
#
# Each directory to which Apache has access can be configured with respect
# to which services and features are allowed and/or disabled in that
# directory (and its subdirectories).
#
# First, we configure the "default" to be a very restrictive set of
# features.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

16. Make sure you are loading up mod_rewrite correctly. You should see something like:

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

In many systems these will be commented out (by being prepended with a #) by default, so you may just need to remove those leading # symbols.

After you make changes, restart Apache to make sure the settings are active.

Verify that you your .htaccess files are actually in the right directories.

This can happen during copying because some operating systems treat files that start with '.' as hidden and therefore won't see them to copy.

17. Make sure your copy of CakePHP is from the downloads section of the site or our GIT repository, and has been unpacked correctly by checking for .htaccess files.

Cake root directory (needs to be copied to your document, this redirects everything to your Cake app):

```
1.      <IfModule mod_rewrite.c>
2.          RewriteEngine on
3.          RewriteRule    ^$ app/webroot/    [L]
4.          RewriteRule    (.*) app/webroot/$1 [L]
5.      </IfModule>
```

Cake app directory (will be copied to the top directory of your application by bake):

```
6.      <IfModule mod_rewrite.c>
7.          RewriteEngine on
8.          RewriteRule    ^$    webroot/    [L]
9.          RewriteRule    (.*) webroot/$1    [L]
10.      </IfModule>
```

Cake webroot directory (will be copied to your application's web root by bake):

```
11.     <IfModule mod_rewrite.c>
12.         RewriteEngine On
13.         RewriteCond %{REQUEST_FILENAME} !-d
14.         RewriteCond %{REQUEST_FILENAME} !-f
15.         RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]
16.     </IfModule>
```

For many hosting services (GoDaddy, 1and1), your web server is actually being served from a user directory that already uses mod_rewrite. If you are installing CakePHP into a user directory (http://example.com/~username/cakephp/), or any other URL structure that already utilizes mod_rewrite, you'll need to add RewriteBase statements to the .htaccess files CakePHP uses (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

This can be added to the same section with the RewriteEngine directive, so for example your webroot .htaccess file would look like:

```
        <IfModule mod_rewrite.c>
            RewriteEngine On
            RewriteBase /path/to/cake/app
            RewriteCond %{REQUEST_FILENAME} !-d
            RewriteCond %{REQUEST_FILENAME} !-f
            RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]
        </IfModule>
```

The details of those changes will depend on your setup, and can include additional things that are not Cake related. Please refer to Apache's online documentation for more information.

3.3.5 Pretty URLs and Lighttpd

While lighttpd features a rewrite module, it is not an equivalent of Apache's mod_rewrite. To get 'pretty URLs' while using Lighty, you have two options. Option one is using mod_rewrite, the second one is by using a LUA script and mod_magnet.

Using mod_rewrite

The easiest way to get pretty URLs is by adding this script to your lighty config. Just edit the URL, and you should be okay. Please note that this doesn't work on Cake installations in subdirectories.

```
$HTTP["host"] =~ "^(www\.)?example.com$" {
        url.rewrite-once = (
                # if the request is for css|files etc, do not pass on to Cake
                "/(css|files|img|js)/(.*)" => "/$1/$2",
                "^([^\?]*)(\?(.+))?$" => "/index.php?url=$1&$3",
        )
        evhost.path-pattern = "/home/%2-%1/www/www/%4/app/webroot/"
}
```

Using mod_magnet

To use pretty URLs with CakePHP and Lighttpd, place this lua script in `/etc/lighttpd/cake`.

```
-- little helper function
function file_exists(path)
  local attr = lighty.stat(path)
  if (attr) then
      return true
  else
      return false
  end
end
function removePrefix(str, prefix)
  return str:sub(1,#prefix+1) == prefix.."/" and str:sub(#prefix+2)
end
-- prefix without the trailing slash
local prefix = ''
-- the magic ;)
if (not file_exists(lighty.env["physical.path"])) then
    -- file still missing. pass it to the fastcgi backend
    request_uri = removePrefix(lighty.env["uri.path"], prefix)
    if request_uri then
      lighty.env["uri.path"]          = prefix .. "/index.php"
      local uriquery = lighty.env["uri.query"] or ""
      lighty.env["uri.query"] = uriquery .. (uriquery ~= "" and "&" or "") .. "url=" .. request_uri
      lighty.env["physical.rel-path"] = lighty.env["uri.path"]
      lighty.env["request.orig-uri"]  = lighty.env["request.uri"]
      lighty.env["physical.path"]     = lighty.env["physical.doc-root"] .. lighty.env["physical.rel-path"]
    end
end
-- fallthrough will put it back into the lighty request loop
-- that means we get the 304 handling for free. ;)
```

If you run your CakePHP installation from a subdirectory, you must set prefix = 'subdirectory_name' in the above script.

Then tell Lighttpd about your vhost:

```
$HTTP["host"] =~ "example.com" {
        server.error-handler-404  = "/index.php"

        magnet.attract-physical-path-to = ( "/etc/lighttpd/cake.lua" )

        server.document-root = "/var/www/cake-1.2/app/webroot/"

        # Think about getting vim tmp files out of the way too
        url.access-deny = (
                "~", ".inc", ".sh", "sql", ".sql", ".tpl.php",
                ".xtmpl", "Entries", "Repository", "Root",
                ".ctp", "empty"
        )
}
```

3.3.6 Pretty URLs on nginx

nginx is a popular server that, like Lighttpd, uses less system resources. It's drawback is that it does not make use of .htaccess files like Apache and Lighttpd, so it is necessary to create those rewritten URLs in the site-available configuration. Depending upon your setup, you will have to modify this, but at the very least, you will need PHP running as a FastCGI instance.

```
1.      server {
2.          listen   80;
3.          server_name www.example.com;
4.          rewrite ^(.*) http://example.com$1 permanent;
5.      }
6.      server {
7.          listen   80;
8.          server_name example.com;
```

```
        access_log /var/www/example.com/log/access.log;
        error_log /var/www/example.com/log/error.log;
        location / {
            root   /var/www/example.com/public/app/webroot/;
            index  index.php index.html index.htm;
            if (-f $request_filename) {
                break;
            }
            rewrite ^(.+)$ /index.php?url=$1 last;
        }
        location ~ .*\.php[345]?$ {
            include /etc/nginx/fcgi.conf;
            fastcgi_pass    127.0.0.1:10005;
            fastcgi_index   index.php;
            fastcgi_param SCRIPT_FILENAME /var/www/example.com/public/app/webroot$fastcgi_script_name;
        }
    }
```

3.3.7 URL Rewrites on IIS7 (Windows hosts)

IIS7 does not natively support .htaccess files. While there are add-ons that can add this support, you can also import htaccess rules into IIS to use CakePHP's native rewrites. To do this, follow these steps:

1. Use Microsoft's Web Platform Installer to install the URL Rewrite Module 2.0.
2. Create a new file in your CakePHP folder, called `web.config`
3. Using Notepad or another XML-safe editor, copy the following code into your new `web.config` file...

```
    <?xml version="1.0" encoding="UTF-8"?>
    <configuration>
        <system.webServer>
```

```
4.                <rewrite>
5.                    <rules>
6.                    <rule name="Imported Rule 1" stopProcessing="true">
7.                    <match url="^(.*)$" ignoreCase="false" />
8.                    <conditions logicalGrouping="MatchAll">
9.                                <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
10.                               <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
11.                   </conditions>
12.                   <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true" />
13.                   </rule>
14.                   <rule name="Imported Rule 2" stopProcessing="true">
15.                     <match url="^$" ignoreCase="false" />
16.                     <action type="Rewrite" url="/" />
17.                   </rule>
18.                   <rule name="Imported Rule 3" stopProcessing="true">
19.                     <match url="(.*)" ignoreCase="false" />
20.                     <action type="Rewrite" url="/{R:1}" />
21.                   </rule>
22.                   <rule name="Imported Rule 4" stopProcessing="true">
23.                     <match url="^(.*)$" ignoreCase="false" />
24.                     <conditions logicalGrouping="MatchAll">
25.                               <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
26.                               <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
27.                     </conditions>
28.                     <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true" />
29.                   </rule>
30.                   </rules>
31.               </rewrite>
32.           </system.webServer>
33.       </configuration>
```

It is also possible to use the Import functionality in IIS's URL Rewrite module to import rules directly from CakePHP's `.htaccess` files in root, `/app/`, and `/app/webroot/` - although some editing within IIS may be necessary to get these to work. When Importing the rules this way, IIS will automatically create your `web.config` file for you.

Once the `web.config` file is created with the correct IIS-friendly rewrite rules, CakePHP's links, css, js, and rerouting should work correctly.

3.3.8 Fire It Up

Alright, let's 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.

Not working? If you're getting timezone related error from PHP uncomment one line in `app/config/core.php`.

```
1.      /**
2.       * If you are on PHP 5.3 uncomment this line and correct your server timezone
3.       * to fix the date & time related errors.
4.       */
5.          date_default_timezone_set('UTC');
```

3.4 Configuration

Configuring a CakePHP application is a piece of cake. After you have installed CakePHP, creating a basic web application requires only that you setup a database configuration.

There are, however, other optional configuration steps you can take in order to take advantage of CakePHP flexible architecture. You can easily add to the functionality inherited from the CakePHP core, configure additional/different URL mappings (routes), and define additional/different inflections.

3.4.1 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.

```
1.      var $default = array('driver'      => 'mysql',
2.                           'persistent'  => false,
3.                           'host'        => 'localhost',
4.                           'login'       => 'cakephpuser',
5.                           'password'    => 'c4k3roxx!',
6.                           'database'    => 'my_cakephp_project',
7.                           'prefix'      => '');
```

The $default connection array is used unless another connection is specified by the $useDbConfig property in a model. For example, if my application has 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	**Value**
driver	The name of the database driver this configuration array is for. Examples: mysql, postgres, sqlite, pear-drivername, adodb-drivername, mssql, oracle, or odbc. Note that for non-database sources (e.g. LDAP, Twitter), leave this blank and use "datasource".
persistent	Whether or not to use a persistent connection to the database.
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 (*optional*)	The string that prefixes every table name in the database. If your tables don't have prefixes, set this to an empty string.
port (*optional*)	The TCP port or Unix socket used to connect to the server.
encoding	Indicates the character set to use when sending SQL statements to the server. This defaults to the database's default encoding for all databases other than DB2. If you wish to use UTF-8 encoding with mysql/mysqli connections you must use 'utf8' without the hyphen.
schema	Used in PostgreSQL database setups to specify which schema to use.
datasource	non-DBO datasource to use, e.g. 'ldap', 'twitter'

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. The correct naming for your tables (and the addition of some columns) can score you some free functionality and help you avoid configuration. For example, if you name your database table big_boxes, your model BigBox, your controller BigBoxesController, everything just works together automatically. By convention, use underscores, lower case, and plural forms for your database table names - for example: bakers, pastry_stores, and savory_cakes.

3.4.2 Core Configuration

Application configuration in CakePHP is found in /app/config/core.php. This file is a collection of Configure class variable definitions and constant definitions that determine how your application behaves. Before we dive into those particular variables, you'll need to be familiar with Configure, CakePHP's configuration registry class.

3.4.3 The Configuration Class

Despite few things needing to be configured in CakePHP, it's sometimes useful to have your own configuration rules for your application. In the past you may have defined custom configuration values by defining variable or constants in some files. Doing so forces you to include that configuration file every time you needed to use those values.

CakePHP's new Configure class can be used to store and retrieve application or runtime specific values. Be careful, this class allows you to store anything in it, then use it in any other part of your code: a sure temptation to break the MVC pattern CakePHP was designed for. The main goal of Configure class is to keep centralized variables that can be shared between many objects. Remember to try to live by "convention over configuration" and you won't end up breaking the MVC structure we've set in place.

This class acts as a singleton and its methods can be called from anywhere within your application, in a static context.

```
1.      <?php Configure::read('debug'); ?>
```

3.4.3.1 Configure Methods

3.4.3.1.1 write

`write(string $key, mixed $value)`

Use `write()` to store data in the application's configuration.

```
1.      Configure::write('Company.name','Pizza, Inc.');
2.      Configure::write('Company.slogan','Pizza for your body and soul');
```

The dot notation used in the `$key` parameter can be used to organize your configuration settings into logical groups.

The above example could also be written in a single call:

```
1.      Configure::write(
2.          'Company',array('name'=>'Pizza, Inc.','slogan'=>'Pizza for your body and soul')
3.      );
```

You can use `Configure::write('debug', $int)` to switch between debug and production modes on the fly. This is especially handy for AMF or SOAP interactions where debugging information can cause parsing problems.

3.4.3.1.2 read

read(string $key = 'debug')

Used to read configuration data from the application. Defaults to CakePHP's important debug value. If a key is supplied, the data is returned. Using our examples from write() above, we can read that data back:

```
1.      Configure::read('Company.name');    //yields: 'Pizza, Inc.'
2.      Configure::read('Company.slogan');  //yields: 'Pizza for your body and soul'
3.
4.      Configure::read('Company');
5.
6.      //yields:
7.      array('name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul');
```

3.4.3.1.3 delete

delete(string $key)

Used to delete information from the application's configuration.

```
1.      Configure::delete('Company.name');
```

3.4.3.1.4 load

load(string $path)

Use this method to load configuration information from a specific file.

```
1.      // /app/config/messages.php:
```

```
2.      <?php
3.      $config['Company']['name'] = 'Pizza, Inc.';
4.      $config['Company']['slogan'] = 'Pizza for your body and soul';
5.      $config['Company']['phone'] = '555-55-55';
6.      ?>
7.
8.      <?php
9.      Configure::load('messages');
10.     Configure::read('Company.name');
11.     ?>
```

Every configure key-value pair is represented in the file with the `$config` array. Any other variables in the file will be ignored by the `load()` function.

3.4.3.1.5 version

`version()`

Returns the CakePHP version for the current application.

3.4.3.2 CakePHP Core Configuration Variables

The Configure class is used to manage a set of core CakePHP configuration variables. These variables can be found in app/config/core.php. Below is a description of each variable and how it affects your CakePHP application.

Configure Variable	Description
debug	Changes CakePHP debugging output. 0 = Production mode. No output. 1 = Show errors and warnings. 2 = Show errors, warnings, and SQL. [SQL log is only shown when you add $this->element('sql_dump') to your view or layout.]

App.baseUrl	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.
Routing.prefixes	Un-comment this definition if you'd like to take advantage of CakePHP prefixed routes like admin. Set this variable with an array of prefix names of the routes you'd like to use. More on this later.
Cache.disable	When set to true, caching is disabled site-wide.
Cache.check	If set to true, enables view caching. Enabling is still needed in the controllers, but this variable enables the detection of those settings.
Session.save	Tells CakePHP which session storage mechanism to use. php = Use the default PHP session storage. cache = Use the caching engine configured by Cache::config(). Very useful in conjunction with Memcache (in setups with multiple application servers) to store both cached data and sessions. 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.
Session.model	The model name to be used for the session model. The model name set here should *not* be used elsewhere in your application.
Session.table	This value has been deprecated as of CakePHP 1.3
Session.database	The name of the database that stores session information.
Session.cookie	The name of the cookie used to track sessions.
Session.timeout	Base session timeout in seconds. Actual value depends on Security.level.
Session.start	Automatically starts sessions when set to true.
Session.checkAgent	When set to false, CakePHP sessions will not check to ensure the user agent does not change between requests.
Security.level	The level of CakePHP security. The session timeout time defined in 'Session.timeout' is multiplied according to the settings here.

	Valid values: 'high' = x 10 'medium' = x 100 'low' = x 300 'high' and 'medium' also enable session.referer_check CakePHP session IDs are also regenerated between requests if 'Security.level' is set to 'high'.
Security.salt	A random string used in security hashing.
Security.cipherSeed	A random numeric string (digits only) used to encrypt/decrypt strings.
Asset.timestamp	Appends a timestamp which is last modified time of the particular file at the end of asset files urls (CSS, JavaScript, Image) when using proper helpers. Valid values: (bool) false - Doesn't do anything (default) (bool) true - Appends the timestamp when debug > 0 (string) 'force' - Appends the timestamp when debug >= 0
Acl.classname, Acl.database	Constants used for CakePHP's Access Control List functionality. See the Access Control Lists chapter for more information.

Cache configuration is also found in core.php — We'll be covering that later on, so stay tuned.

The Configure class can 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.

3.4.3.3 Configuration Constants

While most configuration options are handled by Configure, there are a few constants that CakePHP uses during runtime.

Constant	Description

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

3.4.4 The App Class

Loading additional classes has become more streamlined in CakePHP. In previous versions there were different functions for loading a needed class based on the type of class you wanted to load. These functions have been deprecated, all class and library loading should be done through App::import() now. App::import() ensures that a class is only loaded once, that the appropriate parent class has been loaded, and resolves paths automatically in most cases.

Make sure you follow the file and Classname conventions.

3.4.4.1 Using App::import()

```
App::import($type, $name, $parent, $search, $file, $return);
```

At first glance `App::import` seems complex, however in most use cases only 2 arguments are required.

3.4.4.2 Importing Core Libs

Core libraries such as Sanitize, and Xml can be loaded by:

```
1.      App::import('Core', 'Sanitize');
```

The above would make the Sanitize class available for use.

3.4.4.3 Importing Controllers, Models, Components, Behaviors, and Helpers

All application related classes should also be loaded with App::import(). The following examples illustrate how to do so.

3.4.4.3.1 Loading Controllers

```
App::import('Controller', 'MyController');
```

Calling `App::import` is equivalent to `require`'ing the file. It is important to realize that the class subsequently needs to be initialized.

```
<?php
// The same as require('controllers/users_controller.php');
App::import('Controller', 'Users');
// We need to load the class
$Users = new UsersController;
// If we want the model associations, components, etc to be loaded
$Users->constructClasses();
?>
```

3.4.4.3.2 Loading Models

```
App::import('Model', 'MyModel');
```

3.4.4.3.3 Loading Components

```
App::import('Component', 'Auth');
```

```
<?php
App::import('Component', 'Mailer');
// We need to load the class
$Mailer = new MailerComponent();
?>
```

3.4.4.3.4 Loading Behaviors

```
App::import('Behavior', 'Tree');
```

3.4.4.3.5 Loading Helpers

```
App::import('Helper', 'Html');
```

3.4.4.4 Loading from Plugins

Loading classes in plugins works much the same as loading app and core classes except you must specify the plugin you are loading from.

```
1.      App::import('Model', 'PluginName.Comment');
```

To load APP/plugins/plugin_name/vendors/flickr/flickr.php

```
1.      App::import('Vendor', 'PluginName.flickr/flickr');
```

3.4.4.5 Loading Vendor Files

The vendor() function has been deprecated. Vendor files should now be loaded through App::import() as well. The syntax and additional arguments are slightly different, as vendor file structures can differ greatly, and not all vendor files contain classes.

The following examples illustrate how to load vendor files from a number of path structures. These vendor files could be located in any of the vendor folders.

3.4.4.5.1 Vendor examples

To load **vendors/geshi.php**

```
1.      App::import('Vendor', 'geshi');
```

The geishi file must be a lower-case file name as Cake will not find it otherwise.

To load **vendors/flickr/flickr.php**

```
1.      App::import('Vendor', 'flickr/flickr');
```

To load **vendors/some.name.php**

```
1.      App::import('Vendor', 'SomeName', array('file' => 'some.name.php'));
```

To load **vendors/services/well.named.php**

```
1.      App::import('Vendor', 'WellNamed', array('file' => 'services'.DS.'well.named.php'));
```

It wouldn't make a difference if your vendor files are inside your /app/vendors directory. Cake will automatically find it.

To load **app/vendors/vendorName/libFile.php**

```
1.      App::import('Vendor', 'aUniqueIdentifier', array('file' =>'vendorName'.DS.'libFile.php'));
```

3.4.5 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.

3.4.5.1 Default Routing

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.

```
    URL pattern default routes:
    http://example.com/controller/action/param1/param2/param3
```

The URL /posts/view maps to the view() action of the PostsController, and /products/view_clearance 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.

3.4.5.2 Passed arguments

Passed arguments are additional arguments or path segments that are used when making a request. They are often used to pass parameters to your controller methods.

```
http://localhost/calendars/view/recent/mark
```

In the above example, both `recent` and `mark` are passed arguments to `CalendarsController::view()`. Passed arguments are given to your controllers in three ways. First as arguments to the action method called, and secondly they are available in `$this->params['pass']` as a numerically indexed array. Lastly there is `$this->passedArgs` available in the same way as the second one. When using custom routes you can force particular parameters to go into the passed arguments as well. See passing parameters to an action for more information.

Arguments to the action method called

```
1.      CalendarsController extends AppController{
2.          function view($arg1, $arg2){
3.              debug($arg1);
4.              debug($arg2);
5.              debug(func_get_args());
6.          }
7.      }
```

For this, you will have...

```
recent
mark
Array
(
    [0] => recent
    [1] => mark
)
```

$this->params['pass'] as a numerically indexed array

```
1.      debug($this->params['pass'])
```

For this, you will have...

```
Array
(
    [0] => recent
    [1] => mark
)
```

$this->passedArgs as a numerically indexed array

```
1.      debug($this->passedArgs)
```

```
Array
(
    [0] => recent
    [1] => mark
)
```

$this->passedArgs may also contain Named parameters as a named array mixed with Passed arguments.

3.4.5.3 Named parameters

You can name parameters and send their values using the URL. A request for /posts/view/title:first/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 $this->passedArgs['category'] respectively. You can also access named parameters from `$this->params['named']`. `$this->params['named']` contains an array of named parameters indexed by their name.

Some summarizing examples for default routes might prove helpful.

```
URL to controller action mapping using default routes:

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';
$this->params['named']['chapter'] = 'models';
$this->params['named']['section'] = 'associations';
```

When making custom routes, a common pitfall is that using named parameters will break your custom routes. In order to solve this you should inform the Router about which parameters are intended to be named parameters. Without this knowledge the Router is unable to determine whether named parameters are intended to actually be named parameters or routed parameters, and defaults to assuming you intended them to be routed parameters. To connect named parameters in the router use `Router::connectNamed()`.

```
1.      Router::connectNamed(array('chapter', 'section'));
```

Will ensure that your chapter and section parameters reverse route correctly.

3.4.5.4 Defining 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 default values for your route elements, and regular expression rules to help the router match elements in the URL.

The basic format for a route definition is:

```
1.      Router::connect(
```

```
2.            'URL',
3.            array('paramName' => 'defaultValue'),
4.            array('paramName' => 'matchingRegex')
5.        )
```

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 route elements (variable names 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().

```
1.        Router::connect(
2.            '/pages/*',
3.            array('controller' => 'pages', 'action' => 'display')
4.        );
```

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.

```
1.        Router::connect(
2.            '/government',
3.            array('controller' => 'products', 'action' => 'display', 5)
4.        );
```

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`.

Another common use for the Router is to define an "alias" for a controller. Let's say that instead of accessing our regular URL at `/users/someAction/5`, we'd like to be able to access it by `/cooks/someAction/5`. The following route easily takes care of that:

```
1.      Router::connect(
2.          '/cooks/:action/*', array('controller' => 'users', 'action' => 'index')
3.      );
```

This is telling the Router that any url beginning with `/cooks/` should be sent to the users controller.

When generating urls, routes are used too. Using `array('controller' => 'users', 'action' => 'someAction', 5)` as a url will output `/cooks/someAction/5` if the above route is the first match found

If you are planning to use custom named arguments with your route, you have to make the router aware of it using the `Router::connectNamed` function. So if you want the above route to match urls like `/cooks/someAction/type:chef` we do:

```
1.      Router::connectNamed(array('type'));
2.      Router::connect(
3.          '/cooks/:action/*', array('controller' => 'users', 'action' => 'index')
4.      );
```

You can specify your own 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 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.

```
1.      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']`.

If you have a single controller in your application and you want that controller name does not appear in url, e.g have urls like /demo instead of /home/demo:

```
        Router::connect('/:action', array('controller' => 'home'));
```

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. Note that parenthesis (grouping) are not supported in the regular expressions. You can still specify alternates, as above, but not grouped with parenthesis.

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

3.4.5.5 Passing parameters to action

Assuming your action was defined like this and you want to access the arguments using `$articleID` instead of `$this->params['id']`, just add an extra array in the 3rd parameter of `Router::connect()`.

```
1.      // some_controller.php
2.      function view($articleID = null, $slug = null) {
3.          // some code here...
4.      }
5.      // routes.php
6.      Router::connect(
7.          // E.g. /blog/3-CakePHP_Rocks
8.          '/blog/:id-:slug',
9.          array('controller' => 'blog', 'action' => 'view'),
10.         array(
11.             // order matters since this will simply map ":id" to $articleID in your action
12.             'pass' => array('id', 'slug'),
13.             'id' => '[0-9]+'
14.         )
```

```
15.         );
```

And now, thanks to the reverse routing capabilities, you can pass in the url array like below and Cake will know how to form the URL as defined in the routes.

```
1.          // view.ctp
2.          // this will return a link to /blog/3-CakePHP_Rocks
3.          <?php echo $html->link('CakePHP Rocks', array(
4.              'controller' => 'blog',
5.              'action' => 'view',
6.              'id' => 3,
7.              'slug' => Inflector::slug('CakePHP Rocks')
8.          )); ?>
```

3.4.5.6 Prefix Routing

Many applications require an administration section where privileged users can make changes. This is often done through a special URL such as `/admin/users/edit/5`. In CakePHP, prefix routing can be enabled from within the core configuration file by setting the prefixes with Routing.prefixes. Note that prefixes, although related to the router, are to be configured in /app/config/core.php

```
1.          Configure::write('Routing.prefixes', array('admin'));
```

In your controller, any action with an `admin_` prefix will be called. Using our users example, accessing the url `/admin/users/edit/5` would call the method `admin_edit` of our `UsersController` passing 5 as the first parameter. The view file used would be `app/views/users/admin_edit.ctp`

You can map the url `/admin` to your `admin_index` action of pages controller using following route

```
1.          Router::connect('/admin', array('controller' => 'pages', 'action' => 'index', 'admin' => true));
```

You can configure the Router to use multiple prefixes too. By adding additional values to `Routing.prefixes`. If you set

```
1.      Configure::write('Routing.prefixes', array('admin', 'manager'));
```

Cake will automatically generate routes for both the admin and manager prefixes. Each configured prefix will have the following routes generated for it.

```
1.      $this->connect("/{$prefix}/:plugin/:controller", array('action' => 'index', 'prefix' => $prefix, $prefix =>
true));
2.      $this->connect("/{$prefix}/:plugin/:controller/:action/*", array('prefix' => $prefix, $prefix => true));
3.      Router::connect("/{$prefix}/:controller", array('action' => 'index', 'prefix' => $prefix, $prefix => true));
4.      Router::connect("/{$prefix}/:controller/:action/*", array('prefix' => $prefix, $prefix => true));
```

Much like admin routing all prefix actions should be prefixed with the prefix name. So `/manager/posts/add` would map to `PostsController::manager_add()`.

When using prefix routes its important to remember, using the HTML helper to build your links will help maintain the prefix calls. Here's how to build this link using the HTML helper:

```
1.      // Go into a prefixed route.
2.      echo $html->link('Manage posts', array('manager' => true, 'controller' => 'posts', 'action' => 'add'));
3.      // leave a prefix
4.      echo $html->link('View Post', array('manager' => false, 'controller' => 'posts', 'action' => 'view', 5));
```

3.4.5.7 Plugin routing

Plugin routing uses the **plugin** key. You can create links that point to a plugin, but adding the plugin key to your url array.

```
1.      echo $html->link('New todo', array('plugin' => 'todo', 'controller' => 'todo_items', 'action' => 'create'));
```

Conversely if the active request is a plugin request and you want to create a link that has no plugin you can do the following.

```
1.      echo $html->link('New todo', array('plugin' => null, 'controller' => 'users', 'action' => 'profile'));
```

By setting `plugin => null` you tell the Router that you want to create a link that is not part of a plugin.

3.4.5.8 File extensions

To handle different file extensions with your routes, you need one extra line in your routes config file:

```
1.        Router::parseExtensions('html', 'rss');
```

This will tell the router to remove any matching file extensions, and then parse what remains.

If you want to create a URL such as /page/title-of-page.html you would create your route as illustrated below:

```
1.            Router::connect(
2.                '/page/:title',
3.                array('controller' => 'pages', 'action' => 'view'),
4.                array(
5.                    'pass' => array('title')
6.                )
7.            );
```

Then to create links which map back to the routes simply use:

```
1.        $html->link('Link    title',    array('controller'    =>    'pages',    'action'    =>    'view',    'title'    =>
Inflector::slug('text to slug', '-'), 'ext' => 'html'))
```

3.4.5.9 Custom Route classes

Custom route classes allow you to extend and change how individual routes parse requests and handle reverse routing. A route class should extend `CakeRoute` and implement one or both of `match()` and `parse()`. Parse is used to parse requests and match is used to handle reverse routing.

You can use a custom route class when making a route by using the `routeClass` option, and loading the file containing your route before trying to use it.

```
1.      Router::connect(
2.           '/:slug',
3.           array('controller' => 'posts', 'action' => 'view'),
4.           array('routeClass' => 'SlugRoute')
5.      );
```

This route would create an instance of `SlugRoute` and allow you to implement custom parameter handling

3.4.6 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, you can tell CakePHP about your special cases.

Loading custom inflections

You can use `Inflector::rules()` in the file `app/config/bootstrap.php` to load custom inflections.

```
1.      Inflector::rules('singular', array(
2.          'rules' => array('/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' => '\1ta'),
3.          'uninflected' => array('singulars'),
4.          'irregular' => array('spins' => 'spinor')
5.      ));
```

or

```
1.      Inflector::rules('plural', array('irregular' => array('phylum' => 'phyla')));
```

Will merge the supplied rules into the inflection sets defined in `cake/libs/inflector.php`, with the added rules taking precedence over the core rules.

3.4.7 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 a handy place to use controller callbacks and define methods to be used by all of your controllers.

3.5 Controllers

 # Introduction

-
- View just this section
- Comments (0)
- History

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 recipes 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, which are part of the CakePHP library. 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.

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 (refer to "Routes Configuration" for an explanation on how controller actions and parameters are mapped from the URL).

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:

```
1.      <?php
2.
3.          # /app/controllers/recipes_controller.php
4.          class RecipesController extends AppController {
5.              function view($id)     {
6.                  //action logic goes here..
7.              }
8.              function share($customer_id, $recipe_id) {
9.                  //action logic goes here..
10.             }
11.             function search($query) {
12.                 //action logic goes here..
```

```
13.             }
14.         }
15.         ?>
```

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.

3.5.1 The App Controller

As stated in the introduction, the AppController class is the parent class to all of your application's controllers. AppController itself extends the Controller class included in the CakePHP core library.

As such, AppController is defined in `/cake/libs/controller/app_controller.php` or `/app/app_controller.php`. If `/app/app_controller.php` does not exist then copy from `/cake` location before customizing for application.

Do not customize cake frameworks controller: `/cake/libs/controller/app_controller.php`. These changes will be overwritten during upgrades.

It contains a skeleton definition:

```
1.      <?php
2.      class AppController extends Controller {
3.      }
4.      ?>
```

Controller attributes and methods created in your AppController will be available to all of your application's controllers. It is the ideal place to create code that is common to all of your controllers. Components (which you'll learn about later) are best used for code that is used in many (but not necessarily all) controllers.

While normal object-oriented inheritance rules apply, CakePHP also does a bit of extra work when it comes to special controller attributes, like the list of components or helpers used by a controller. In these cases, AppController value arrays are merged with child controller class arrays.

CakePHP merges the following variables from the AppController to your application's controllers:

- $components
- $helpers
- $uses

Remember to add the default Html and Form helpers, if you define var $helpers in your AppController

Please also remember to call AppController's callbacks within child controller callbacks for best results:

```
1.      function beforeFilter(){
2.          parent::beforeFilter();
3.      }
```

3.5.2 The Pages Controller

CakePHP core ships with a default controller called the Pages Controller (`cake/libs/controller/pages_controller.php`). The home page you see after installation is generated using this controller. It is generally used to serve static pages. Eg. If you make a view file `app/views/pages/about_us.ctp` you can access it using url `http://example.com/pages/about_us`

When you "`bake`" an app using CakePHP's console utility the pages controller is copied to your `app/controllers/` folder and you can modify it to your needs if required. Or you could just copy the `pages_controller.php` from core to your app.

Do not directly modify ANY file under the `cake` folder to avoid issues when updating the core in future

3.5.3 Controller Attributes

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

3.5.3.1 $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 controller uses. This takes care of some PHP4 classname oddities and helps CakePHP resolve naming.

```
1.      <?php
2.      #   $name controller attribute usage example
3.      class RecipesController extends AppController {
4.         var $name = 'Recipes';
5.      }
6.      ?>
```

3.5.3.2 $components, $helpers and $uses

The next most often used controller attributes tell CakePHP what helpers, components, and models you'll be using in conjunction with the current controller. Using these attributes make MVC classes given by `$components` and `$uses` available to the controller as class variables (`$this->ModelName`, for example) and those given by `$helpers` to the view as an object reference variable (`$helpername`).

Each controller has some of these classes available by default, so you may not need to configure your controller at all.

Controllers have access to 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`. However, when allowing a controller to access additional models through the `$uses` variable, the name of the current controller's model must also be included. This is illustrated in the example below.

The Html, Form, and Session Helpers are always available by default, as is the SessionComponent. But if you choose to define your own $helpers array in AppController, make sure to include `Html` and `Form` if you want them still available by default in your own Controllers. 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.

```
1.      <?php
2.      class RecipesController extends AppController {
```

```
3.          var $name = 'Recipes';
4.          var $uses = array('Recipe', 'User');
5.          var $helpers = array('Ajax');
6.          var $components = array('Email');
7.      }
8.      ?>
```

Each of these variables are merged with their inherited values, therefore it is not necessary (for example) to redeclare the Form helper, or anything that is declared in your App controller.

If you do not wish to use a Model in your controller, set `var $uses = array()`. This will allow you to use a controller without a need for a corresponding Model file.

It's bad practice to just add all the models your controller uses to the `$uses` array. Check here and here to see how to properly access associated and unassociated models respectively.

3.5.3.3 Page-related Attribute: $layout

A few attributes exist in CakePHP controllers that give you control over how your view is 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, `default.ctp`. If you haven't defined one at `/app/views/layouts/default.ctp`, CakePHP's core default layout will be rendered.

```
1.      <?php
2.      //   Using $layout to define an alternate layout
3.      class RecipesController extends AppController {
4.          function quickSave() {
5.              $this->layout = 'ajax';
6.          }
```

```
7.          }
8.          ?>
```

3.5.3.4 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 handled to the controller via POST or GET operations.

3.5.3.4.1 form

```
$this->params['form']
```

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

3.5.3.4.2 admin

```
$this->params['admin']
```

Is set to 1 if the current action was invoked via admin routing.

3.5.3.4.3 bare

```
$this->params['bare']
```

Stores 1 if the current layout is empty, 0 if not.

3.5.3.4.4 isAjax

```
$this->params['isAjax']
```

Stores 1 if the current request is an ajax call, 0 if not. This variable is only set if the RequestHandler Component is being used in the controller.

3.5.3.4.5 controller

$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".

3.5.3.4.6 action

$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".

3.5.3.4.7 pass

$this->params['pass']

Returns an array (numerically indexed) of URL parameters after the Action.

```
// URL: /posts/view/12/print/narrow

Array
(
    [0] => 12
    [1] => print
    [2] => narrow
)
```

3.5.3.4.8 url

$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
)
```

3.5.3.4.9 data

$this->data

Used to handle POST data sent from the FormHelper forms to the controller.

```
1.        // The FormHelper is used to create a form element:
2.        $form->text('User.first_name');
```

Which when rendered, 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`

```
1.
2.        //The submitted first name can be found here:
3.        $this->data['User']['first_name'];
```

3.5.3.4.10 prefix

$this->params['prefix']

Set to the routing prefix. For example, this attribute would contain the string "admin" during a request to /admin/posts/someaction.

3.5.3.4.11 named

$this->params['named']

Stores any named parameters in the url query string in the form /key:value/. For example, if the URL `/posts/view/var1:3/var2:4` was requested, `$this->params['named']` would be an array containing:

```
[named] => Array
(
    [var1] => 3
    [var2] => 4
)
```

3.5.3.5 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 later on in this manual.

3.5.3.6 persistModel

Stub. Update Me!

Used to create cached instances of models a controller uses. When set to true, all models related to the controller will be cached. This can increase performance in many cases.

3.5.4 Controller Methods

For a complete list of controller methods and their descriptions visit the CakePHP API. Check out http://api13.cakephp.org/class/controller.

3.5.4.1 Interacting with Views

Controllers interact with the view in a number of ways. First they are able to pass data to the views, using `set()`. You can also decide which view class to use, and which view file should be rendered from the controller.

set

```
set(string $var, mixed $value)
```

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

```
<?php

//First you pass data from the controller:
$this->set('color', 'pink');
//Then, in the view, you can utilize the data:
?>

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

The `set()` method also takes an associative array as its first parameter. This can often be a quick way to assign a set of information to the view.

Array keys will be no longer be inflected before they are assigned to the view ('underscored_key' does not become 'underscoredKey' anymore, etc.):

```
<?php

$data = array(
    'color' => 'pink',
    'type' => 'sugar',
    'base_price' => 23.95
);
//make $color, $type, and $base_price
//available to the view:
$this->set($data);
?>
```

The attribute `$pageTitle` no longer exists, use `set()` to set the title

```
<?php
$this->set('title_for_layout', 'This is the page title');
?>
```

 # render

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.

```
class RecipesController extends AppController {
...
    function search() {
        // Render the view in /views/recipes/search.ctp
        $this->render();
    }
...
}
```

Although CakePHP will automatically call it (unless you've set `$this->autoRender` to false) after every action's logic, you can use it to specify an alternate view file by specifying an action name in the controller using `$action`.

If `$action` starts with '/' it is assumed to be a view or element file relative to the `/app/views` folder. This allows direct rendering of elements, very useful in ajax calls.

```
// Render the element in /views/elements/ajaxreturn.ctp
$this->render('/elements/ajaxreturn');
```

You can also specify an alternate view or element 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.

 # Rendering a specific view

In your controller you may want to render a different view than what would conventionally be done. You can do this by calling `render()` directly. Once you have called `render()` CakePHP will not try to re-render the view.

```
1.      class PostsController extends AppController {
2.          function my_action() {
3.              $this->render('custom_file');
4.          }
5.      }
```

This would render `app/views/posts/custom_file.ctp` instead of `app/views/posts/my_action.ctp`

3.5.4.2 Flow Control

3.5.4.2.1 redirect

`redirect(mixed $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.

```
1.      function placeOrder() {
2.          //Logic for finalizing order goes here
3.          if($success) {
4.              $this->redirect(array('controller' => 'orders', 'action' => 'thanks'));
5.          } else {
```

```
6.                 $this->redirect(array('controller' => 'orders', 'action' => 'confirm'));
7.             }
8.         }
```

You can also use a relative or absolute URL as the $url argument:

```
1.         $this->redirect('/orders/thanks'));
2.         $this->redirect('http://www.example.com');
```

You can also pass data to the action:

```
1.         $this->redirect(array('action' => 'edit', $id));
```

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.

The method will issue an `exit()` after the redirect unless you set the third parameter to `false`.

If you need to redirect to the referer page you can use:

```
1.         $this->redirect($this->referer());
```

3.5.4.2.2 flash

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

Like `redirect()`, 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.

If there's a particular template you'd like your flashed message to use, you may specify the name of that layout in the `$layout` parameter.

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

3.5.4.3 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. It's 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, and after rendering is complete. This is the last controller method to run.

CakePHP also supports callbacks related to scaffolding.

_beforeScaffold($method)

$method name of method called example index, edit, etc.

_afterScaffoldSave($method)

$method name of method called either edit or update.

_afterScaffoldSaveError($method)

$method name of method called either edit or update.

```
_scaffoldError($method)
```

$method name of method called example index, edit, etc.

3.5.4.4 Other Useful Methods

3.5.4.4.1 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 some other outside use, constructClasses() may come in handy.

3.5.4.4.2 referer

```
string referer(mixed $default = null, boolean $local = false)
```

Returns the referring URL for the current request. Parameter `$default` can be used to supply a default URL to use if HTTP_REFERER cannot be read from headers. So, instead of doing this:

```
1.  <?php
2.  class UserController extends AppController {
3.      function delete($id) {
4.          // delete code goes here, and then...
5.          if ($this->referer() != '/') {
6.              $this->redirect($this->referer());
7.          } else {
8.              $this->redirect(array('action' => 'index'));
9.          }
10.     }
11. }
12. ?>
```

you can do this:

```
1.      <?php
2.      class UserController extends AppController {
3.          function delete($id) {
4.              // delete code goes here, and then...
5.              $this->redirect($this->referer(array('action' => 'index')));
6.          }
7.      }
8.      ?>
```

If `$default` is not set, the function defaults to the root of your domain - '/'.

Parameter `$local` if set to `true`, restricts referring URLs to local server.

3.5.4.4.3 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.

The headers sent to this effect are:

- `Expires: Mon, 26 Jul 1997 05:00:00 GMT`
- `Last-Modified: [current datetime] GMT`
- `Cache-Control: no-store, no-cache, must-revalidate`
- `Cache-Control: post-check=0, pre-check=0`
- `Pragma: no-cache`

3.5.4.4.4 postConditions

`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 a 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:

```
1.  function index() {
2.      $conditions = $this->postConditions($this->data);
3.      $orders = $this->Order->find("all",compact('conditions'));
4.      $this->set('orders', $orders);
5.  }
```

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

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

```
1.  /*
2.  Contents of $this->data
3.  array(
4.      'Order' => array(
5.          'num_items' => '4',
6.          'referrer' => 'Ye Olde'
7.      )
8.  )
9.  */
10. //Let's get orders that have at least 4 items and contain 'Ye Olde'
11. $condtions=$this->postConditions(
12.     $this->data,
13.     array(
14.         'num_items' => '>=',
15.         'referrer' => 'LIKE'
```

```
            )
        );
        $orders = $this->Order->find("all",compact('condtions'));
```

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.

3.5.4.4.5 paginate

This method is used for paginating results fetched by your models. You can specify page sizes, model find conditions and more. See the pagination section for more details on how to use paginate.

3.5.4.4.6 requestAction

`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`). To pass extra data to the receiving controller action add to the $options array.

You can use `requestAction()` to retrieve a fully rendered view by passing 'return' in the options: `requestAction($url, array('return'));`. It is important to note that making a requestAction using 'return' from a controller method can cause script and css tags to not work correctly.

If used without caching `requestAction` can lead to poor performance. It is rarely appropriate to use in a controller or model.

`requestAction` is best used in conjunction with (cached) elements – as a way to fetch data for an element before rendering. Let's use the example of putting a "latest comments" element in the layout. First we need to create a controller function that will return the data.

```
        // controllers/comments_controller.php
```

```
2.      class CommentsController extends AppController {
3.          function latest() {
4.              return $this->Comment->find('all', array('order' => 'Comment.created DESC', 'limit' => 10));
5.          }
6.      }
```

If we now create a simple element to call that function:

```
1.      // views/elements/latest_comments.ctp
2.      $comments = $this->requestAction('/comments/latest');
3.      foreach($comments as $comment) {
4.          echo $comment['Comment']['title'];
5.      }
```

We can then place that element anywhere at all to get the output using:

```
1.      echo $this->element('latest_comments');
```

Written in this way, whenever the element is rendered, a request will be made to the controller to get the data, the data will be processed, and returned. However in accordance with the warning above it's best to make use of element caching to prevent needless processing. By modifying the call to element to look like this:

```
1.      echo $this->element('latest_comments', array('cache' => '+1 hour'));
```

The `requestAction` call will not be made while the cached element view file exists and is valid.

In addition, requestAction now takes array based cake style urls:

```
1.      echo $this->requestAction(array('controller' => 'articles', 'action' => 'featured'), array('return'));
```

This allows the requestAction call to bypass the usage of Router::url which can increase performance. The url based arrays are the same as the ones that HtmlHelper::link uses with one difference - if you are using named or passed parameters, you must put them in a second array and wrap them with the correct key. This is because requestAction merges the named args array (requestAction's 2nd parameter) with the Controller::params member array and does not explicitly place the named args array into the key 'named'; Additional members in the $option array will also be made available in the requested action's Controller::params array.

```
1.      echo $this->requestAction('/articles/featured/limit:3');
2.      echo $this->requestAction('/articles/view/5');
```

As an array in the requestAction would then be:

```
1.      echo  $this->requestAction(array('controller'  =>  'articles',  'action'  =>  'featured'),  array('named'  =>
array('limit' => 3)));
2.      echo $this->requestAction(array('controller' => 'articles', 'action' => 'view'), array('pass' => array(5)));
```

Unlike other places where array urls are analogous to string urls, requestAction treats them differently.

When using an array url in conjunction with requestAction() you must specify **all** parameters that you will need in the requested action. This includes parameters like `$this->data` and `$this->params['form']`. In addition to passing all required parameters, named and pass parameters must be done in the second array as seen above.

3.5.4.4.7 loadModel

`loadModel(string $modelClass, mixed $id)`

The `loadModel` function comes handy when you need to use a model which is not the controller's default model or its associated model.

```
1.      $this->loadModel('Article');
2.      $recentArticles = $this->Article->find('all', array('limit' => 5, 'order' => 'Article.created DESC'));
1.      $this->loadModel('User', 2);
```

```
2.        $user = $this->User->read();
```

3.6 Components

3.6.1 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.

3.6.2 Configuring Components

Many of the core components require configuration. Some examples of components requiring configuration are Auth, Cookie and Email. Configuration for these components, and for components in general, is usually done in the `$components` array or your controller's `beforeFilter()` method.

```
1.        var $components = array(
2.            'Auth' => array(
3.                'authorize' => 'controller',
```

```
            'loginAction' => array('controller' => 'users', 'action' => 'login')
        ),
        'Cookie' => array('name' => 'CookieMonster')
    );
```

Would be an example of configuring a component with the `$components` array. All core components allow their configuration settings to be set in this way. In addition you can configure components in your controller's `beforeFilter()` method. This is useful when you need to assign the results of a function to a component property. The above could also be expressed as:

```
function beforeFilter() {
    $this->Auth->authorize = 'controller';
    $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');

    $this->Cookie->name = 'CookieMonster';
}
```

It's possible, however, that a component requires certain configuration options to be set before the controller's `beforeFilter()` is run. To this end, some components allow configuration options be set in the `$components` array.

```
var $components = array('DebugKit.toolbar' => array('panels' => array('history', 'session')));
```

Consult the relevant documentation to determine what configuration options each component provides.

3.6.3 Creating Components

Suppose our online application needs to perform a complex mathematical operation in many different parts of the application. We could create a component to house this shared logic for use in many different controllers.

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

```
<?php
class MathComponent extends Object {
    function doComplexOperation($amount1, $amount2) {
        return $amount1 + $amount2;
    }
}
?>
```

Take notice that our MathComponent extends Object and not Component. Extending Component can create infinite redirect issues, when combined with other Components.

3.6.3.1 Including Components in your Controllers

Once our component is finished, we can use it in the application's controllers by placing the component's name (minus the "Component" part) in the controller's $components array. The controller will automatically be given a new attribute named after the component, through which we can access an instance of it:

```
/* Make the new component available at $this->Math,
as well as the standard $this->Session */
var $components = array('Math', 'Session');
```

Components declared in `AppController` will be merged with those in your other controllers. So there is no need to re-declare the same component twice.

When including Components in a Controller you can also declare a set of parameters that will be passed on to the Component's `initialize()` method. These parameters can then be handled by the Component.

```
var $components = array(
    'Math' => array(
        'precision' => 2,
```

```
4.                 'randomGenerator' => 'srand'
5.             ),
6.             'Session', 'Auth'
7.         );
```

The above would pass the array containing precision and randomGenerator to MathComponent's initialize() method as the second parameter.

This syntax is not implemented by any of the Core Components at this time

3.6.3.2 MVC Class Access Within Components

Components feature a number of callbacks used by the parent controller class. Judicious use of these callbacks can make creating and using components much easier..

initialize(&$controller, $settings=array())

The initialize method is called before the controller's beforeFilter method.

startup(&$controller)

The startup method is called after the controller's beforeFilter method but before the controller executes the current action handler.

beforeRender(&$controller)

The beforeRender method is called after the controller executes the requested action's logic but before the controller's renders views and layout.

shutdown(&$controller)

The shutdown method is called before output is sent to browser.

beforeRedirect(&$controller, $url, $status=null, $exit=true)

The beforeRedirect method is invoked when the controller's redirect method is called but before any further action. If this method returns false the controller will not continue on to redirect the request. The $url, $status and $exit variables have same meaning as for the controller's method. You can also return a string which will be interpreted as the url to redirect to or return associative array with key 'url' and optionally 'status' and 'exit'.

Here is a skeleton component you can use as a template for your own custom components.

```
<?php
class SkeletonComponent extends Object {
    //called before Controller::beforeFilter()
    function initialize(&$controller, $settings = array()) {
        // saving the controller reference for later use
        $this->controller =& $controller;
    }
    //called after Controller::beforeFilter()
    function startup(&$controller) {
    }
    //called after Controller::beforeRender()
    function beforeRender(&$controller) {
    }
    //called after Controller::render()
    function shutdown(&$controller) {
    }
    //called before Controller::redirect()
    function beforeRedirect(&$controller, $url, $status=null, $exit=true) {
    }
    function redirectSomewhere($value) {
        // utilizing a controller method
        $this->controller->redirect($value);
    }
}
?>
```

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.

```
<?php
class MyComponent extends Object {
    // This component uses other components
    var $components = array('Session', 'Math');
    function doStuff() {
        $result = $this->Math->doComplexOperation(1, 2);
        $this->Session->write('stuff', $result);
    }
}
?>
```

To access/use a model in a component is not generally recommended; If you end up needing one, you'll need to instantiate your model class and use it manually. Here's an example:

```
<?php
class MathComponent extends Object {
    function doComplexOperation($amount1, $amount2) {
        return $amount1 + $amount2;
    }
    function doReallyComplexOperation ($amount1, $amount2) {
        $userInstance = ClassRegistry::init('User');
        $totalUsers = $userInstance->find('count');
        return ($amount1 + $amount2) / $totalUsers;
    }
}
?>
```

3.6.3.3 Using other Components in your Component

Sometimes one of your components may need to use another.

You can include other components in your component the exact same way you include them in controllers: Use the `$components` var.

```
<?php
class CustomComponent extends Object {
    var $name = 'Custom'; // the name of your component
    var $components = array('Existing'); // the other component your component uses
    function initialize(&$controller) {
        $this->Existing->foo();
    }
    function bar() {
        // ...
    }
}
?>
<?php
class ExistingComponent extends Object {
    var $name = 'Existing';
    function initialize(&$controller) {
        $this->Parent->bar();
    }

    function foo() {
        // ...
    }
}
?>
```

3.7 Models

Models represent data and are used in CakePHP applications for data access. A model usually represents a database table but can be used to access anything that stores data such as files, LDAP records, iCal events, or rows in a CSV file.

A model can be associated with other models. For example, a Recipe may be associated with the Author of the recipe as well as the Ingredient in the recipe.

This section will explain what features of the model can be automated, how to override those features, and what methods and properties a model can have. It'll explain the different ways to associate your data. It'll describe how to find, save, and delete data. Finally, it'll look at Datasources.

3.7.1 Understanding Models

A Model represents your data model. In object-oriented programming a data model is an object that represents a "thing", like a car, a person, or a house. A blog, for example, may have many blog posts and each blog post may have many comments. The Blog, Post, and Comment are all examples of models, each associated with another.

Here is a simple example of a model definition in CakePHP:

```
1.  <?php
2.  class Ingredient extends AppModel {
3.      var $name = 'Ingredient';
4.  }
5.  ?>
```

With just this simple declaration, the Ingredient model is bestowed with all the functionality you need to create queries along with saving and deleting data. These magic methods come from CakePHP's Model class by the magic of inheritance. The Ingredient model extends the application model, AppModel, which extends CakePHP's internal Model class. It is this core Model class that bestows the functionality onto your Ingredient model.

This intermediate class, AppModel, is empty and if you haven't created your own is taken from within the `/cake/` folder. Overriding the AppModel allows you to define functionality that should be made available to all models within your application. To do so, you need to create your own `app_model.php` file that resides in the root of the `/app/` folder. Creating a project using Bake will automatically generate this file for you.

Create your model PHP file in the `/app/models/` directory or in a subdirectory of `/app/models`. CakePHP will find it anywhere in the directory. By convention it should have the same name as the class; for this example `ingredient.php`.

CakePHP will dynamically create a model object for you if it cannot find a corresponding file in `/app/models`. This also means that if your model file isn't named correctly (i.e. `Ingredient.php` or `ingredients.php`) CakePHP will use a instance of AppModel rather than your missing (from CakePHP's perspective) model file. If you're trying to use a method you've defined in your model, or a behavior attached to your model and you're getting SQL errors that are the name of the method you're calling - it's a sure sign CakePHP can't find your model and you either need to check the file names, clear your tmp files, or both.

See also Behaviors for more information on how to apply similar logic to multiple models.

The `$name` property is necessary for PHP4 but optional for PHP5.

With your model defined, it can be accessed from within your Controller. CakePHP will automatically make the model available for access when its name matches that of the controller. For example, a controller named IngredientsController will automatically initialize the Ingredient model and attach it to the controller at `$this->Ingredient`.

```
1.  <?php
2.  class IngredientsController extends AppController {
3.      function index() {
4.          //grab all ingredients and pass it to the view:
5.          $ingredients = $this->Ingredient->find('all');
6.          $this->set('ingredients', $ingredients);
7.      }
8.  }
9.  ?>
```

Associated models are available through the main model. In the following example, Recipe has an association with the Ingredient model.

```
<?php
class RecipesController extends AppController {
    function index() {
        $ingredients = $this->Recipe->Ingredient->find('all');
        $this->set('ingredients', $ingredients);
    }
}
?>
```

If models have absolutely NO association between them, you can use Controller::loadModel() to get the model.

```
<?php
class RecipesController extends AppController {
    function index() {
        $recipes = $this->Recipe->find('all');

        $this->loadModel('Car');
        $cars = $this->Car->find('all');

        $this->set(compact('recipes', 'cars'));
    }
}
?>
```

Some class names are not usable for model names. For instance "File" cannot be used as "File" is a class already existing in the CakePHP core.

3.7.2 Creating Database Tables

While CakePHP can have datasources that aren't database driven, most of the time, they are. CakePHP is designed to be agnostic and will work with MySQL, MSSQL, Oracle, PostgreSQL and others. You can create your database tables as you normally would. When you create your Model classes, they'll automatically map to the tables that you've created.

Table names are by convention lowercase and pluralized with multi-word table names separated by underscores. For example, a Model name of Ingredient expects the table name ingredients. A Model name of EventRegistration would expect a table name of event_registrations. CakePHP will inspect your tables to determine the data type of each field and uses this information to automate various features such as outputting form fields in the view.

Field names are by convention lowercase and separated by underscores.

Model to table name associations can be overridden with the `useTable` attribute of the model explained later in this chapter.

In the rest of this section, you'll see how CakePHP maps database field types to PHP data types and how CakePHP can automate tasks based on how your fields are defined.

3.7.2.1 Data Type Associations by Database

Every RDBMS defines data types in slightly different ways. Within the datasource class for each database system, CakePHP maps those types to something it recognizes and creates a unified interface, no matter which database system you need to run on.

This breakdown describes how each one is mapped.

3.7.2.1.1 MySQL

CakePHP Type	Field Properties
primary_key	NOT NULL auto_increment
string	varchar(255)
text	text

integer	int(11)
float	float
datetime	datetime
timestamp	datetime
time	time
date	date
binary	blob
boolean	tinyint(1)

A *tinyint(1)* field is considered a boolean by CakePHP.

3.7.2.1.2 MySQLi

CakePHP Type	**Field Properties**
primary_key	DEFAULT NULL auto_increment
string	varchar(255)
text	text
integer	int(11)
float	float

datetime	datetime
timestamp	datetime
time	time
date	date
binary	blob
boolean	tinyint(1)

3.7.2.1.3 ADOdb

CakePHP Type	Field Properties
primary_key	R(11)
string	C(255)
text	X
integer	I(11)
float	N
datetime	T (Y-m-d H:i:s)
timestamp	T (Y-m-d H:i:s)
time	T (H:i:s)
date	T (Y-m-d)

binary	B
boolean	L(1)

3.7.2.1.4 DB2

CakePHP Type	Field Properties
primary_key	not null generated by default as identity (start with 1, increment by 1)
string	varchar(255)
text	clob
integer	integer(10)
float	double
datetime	timestamp (Y-m-d-H.i.s)
timestamp	timestamp (Y-m-d-H.i.s)
time	time (H.i.s)
date	date (Y-m-d)
binary	blob
boolean	smallint(1)

3.7.2.1.5 Firebird/Interbase

CakePHP Type	Field Properties

primary_key	IDENTITY (1, 1) NOT NULL
string	varchar(255)
text	BLOB SUB_TYPE 1 SEGMENT SIZE 100 CHARACTER SET NONE
integer	integer
float	float
datetime	timestamp (d.m.Y H:i:s)
timestamp	timestamp (d.m.Y H:i:s)
time	time (H:i:s)
date	date (d.m.Y)
binary	blob
boolean	smallint

3.7.2.1.6 MS SQL

CakePHP Type	Field Properties
primary_key	IDENTITY (1, 1) NOT NULL
string	varchar(255)
text	text
integer	int

float	numeric
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	datetime (H:i:s)
date	datetime (Y-m-d)
binary	image
boolean	bit

3.7.2.1.7 Oracle

CakePHP Type	Field Properties
primary_key	number NOT NULL
string	varchar2(255)
text	varchar2
integer	numeric
float	float
datetime	date (Y-m-d H:i:s)
timestamp	date (Y-m-d H:i:s)
time	date (H:i:s)

date	date (Y-m-d)
binary	bytea
boolean	boolean
number	numeric
inet	inet

3.7.2.1.8 PostgreSQL

CakePHP Type	Field Properties
primary_key	serial NOT NULL
string	varchar(255)
text	text
integer	integer
float	float
datetime	timestamp (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	time (H:i:s)
date	date (Y-m-d)
binary	bytea

boolean	boolean
number	numeric
inet	inet

3.7.2.1.9 SQLite

CakePHP Type	Field Properties
primary_key	integer primary key
string	varchar(255)
text	text
integer	integer
float	float
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	time (H:i:s)
date	date (Y-m-d)
binary	blob
boolean	boolean

3.7.2.1.10 Sybase

CakePHP Type	Field Properties
primary_key	numeric(9,0) IDENTITY PRIMARY KEY
string	varchar(255)
text	text
integer	int(11)
float	float
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	datetime (H:i:s)
date	datetime (Y-m-d)
binary	image
boolean	bit

3.7.2.2 Titles

An object, in the physical sense, often has a name or a title that refers to it. A person has a name like John or Mac or Buddy. A blog post has a title. A category has a name.

By specifying a `title` or `name` field, CakePHP will automatically use this label in various circumstances:

- Scaffolding — page titles, fieldset labels
- Lists — normally used for `<select>` drop-downs

- TreeBehavior — reordering, tree views

If you have a title *and* name field in your table, the title will be used.

If you want to use something other than the convention set `var $displayField = 'some_field';`. Only one field can be set here.

3.7.2.3 created and modified

By defining a `created` or `modified` field in your database table as `datetime` fields, CakePHP will recognize those fields and populate them automatically whenever a record is created or saved to the database (unless the data being saved already contains a value for these fields).

The `created` and `modified` fields will be set to the current date and time when the record is initially added. The `modified` field will be updated with the current date and time whenever the existing record is saved.

Note: A field named `updated` will exhibit the same behavior as `modified`. These fields need to be datetime fields with the default value set to NULL to be recognized by CakePHP.

If you have `updated`, `created` or `modified` data in your `$this->data` (e.g. from a `Model::read` or `Model::set`) before a `Model::save()` then the values will be taken from `$this->data` and not automagically updated.

Either use `unset($this->data['Model']['modified'])`, etc. Alternatively you can override the `Model::save()` to always do it for you:-

```
1.      class AppModel extends Model {
2.      //
3.      //
4.          function save($data = null, $validate = true, $fieldList = array()) {
5.              //clear modified field value before each save
6.              if (isset($this->data) && isset($this->data[$this->name]))
7.                  unset($this->data[$this->name]['modified']);
8.              if (isset($data) && isset($data[$this->name]))
```

```
                unset($data[$this->name]['modified']);
            return parent::save($data, $validate, $fieldList);
        }
    //
    //
    }
```

3.7.2.4 Using UUIDs as Primary Keys

Primary keys are normally defined as INT fields. The database will automatically increment the field, starting at 1, for each new record that gets added. Alternatively, if you specify your primary key as a CHAR(36) or BINARY(36), CakePHP will automatically generate UUIDs when new records are created.

A UUID is a 32 byte string separated by four hyphens, for a total of 36 characters. For example:

```
550e8400-e29b-41d4-a716-446655440000
```

UUIDs are designed to be unique, not only within a single table, but also across tables and databases. If you require a field to remain unique across systems then UUIDs are a great approach.

3.7.3 Retrieving Your Data

3.7.3.1 find

`find($type, $params)`

Find is the multifunctional workhorse of all model data-retrieval functions. `$type` can be either `'all'`, `'first'`, `'count'`, `'list'`, `'neighbors'` or `'threaded'`. The default find type is `'first'`. Keep in mind that `$type` is case sensitive. Using a upper case character (for example `'All'`) will not produce the expected results.

`$params` is used to pass all parameters to the various finds, and has the following possible keys by default - all of which are optional:

```
    array(
        'conditions' => array('Model.field' => $thisValue), //array of conditions
```

```
            'recursive' => 1, //int
            'fields' => array('Model.field1', 'DISTINCT Model.field2'), //array of field names
            'order' => array('Model.created', 'Model.field3 DESC'), //string or array defining order
            'group' => array('Model.field'), //fields to GROUP BY
            'limit' => n, //int
            'page' => n, //int
            'offset'=>n, //int
            'callbacks' => true //other possible values are false, 'before', 'after'
        )
```

It's also possible to add and use other parameters, as is made use of by some find types, behaviors and of course possible with your own model methods

More information about model callbacks is available here

3.7.3.1.1 find('first')

```
find('first', $params)
```

'first' is the default find type, and will return one result, you'd use this for any use where you expect only one result. Below are a couple of simple (controller code) examples:

```
        function some_function() {
            ...
            $this->Article->order = null; // resetting if it's set
            $semiRandomArticle = $this->Article->find();
            $this->Article->order = 'Article.created DESC'; // simulating the model having a default order
            $lastCreated = $this->Article->find();
            $alsoLastCreated = $this->Article->find('first', array('order' => array('Article.created DESC')));
            $specificallyThisOne = $this->Article->find('first', array('conditions' => array('Article.id' => 1)));
            ...
        }
```

In the first example, no parameters at all are passed to find - therefore no conditions or sort order will be used. The format returned from `find('first')` call is of the form:

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

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

There are no additional parameters used by `find('first')`.

3.7.3.1.2 find('count')

`find('count', $params)`

`find('count', $params)` returns an integer value. Below are a couple of simple (controller code) examples:

```
1.      function some_function() {
2.          ...
3.          $total = $this->Article->find('count');
4.          $pending = $this->Article->find('count', array('conditions' => array('Article.status' => 'pending')));
5.          $authors = $this->Article->User->find('count');
```

```
6.          $publishedAuthors = $this->Article->find('count', array(
7.              'fields' => 'DISTINCT Article.user_id',
8.              'conditions' => array('Article.status !=' => 'pending')
9.          ));
10.         ...
11.     }
```

Don't pass `fields` as an array to `find('count')`. You would only need to specify fields for a DISTINCT count (since otherwise, the count is always the same - dictated by the conditions).

There are no additional parameters used by `find('count')`.

3.7.3.1.3 find('all')

`find('all', $params)`

`find('all')` returns an array of (potentially multiple) results. It is in fact the mechanism used by all `find()` variants, as well as `paginate`. Below are a couple of simple (controller code) examples:

```
1.      function some_function() {
2.          ...
3.          $allArticles = $this->Article->find('all');
4.          $pending = $this->Article->find('all', array('conditions' => array('Article.status' => 'pending')));
5.          $allAuthors = $this->Article->User->find('all');
6.          $allPublishedAuthors = $this->Article->User->find('all', array('conditions' => array('Article.status !='
=> 'pending')));
7.          ...
8.      }
```

In the above example `$allAuthors` will contain every user in the users table, there will be no condition applied to the find as none were passed.

The results of a call to `find('all')` will be of the following form:

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

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

        )
)
```

There are no additional parameters used by `find('all')`.

3.7.3.1.4 find('list')

`find('list', $params)`

`find('list', $params)` returns an indexed array, useful for any use where you would want a list such as for populating input select boxes. Below are a couple of simple (controller code) examples:

```
1.        function some_function() {
2.            ...
3.             $allArticles = $this->Article->find('list');
4.             $pending = $this->Article->find('list', array(
5.                 'conditions' => array('Article.status' => 'pending')
6.             ));
7.             $allAuthors = $this->Article->User->find('list');
8.             $allPublishedAuthors = $this->Article->find('list', array(
9.                 'fields' => array('User.id', 'User.name'),
10.                'conditions' => array('Article.status !=' => 'pending'),
11.                'recursive' => 0
12.            ));
13.           ...
14.       }
```

In the above example `$allAuthors` will contain every user in the users table, there will be no condition applied to the find as none were passed.

The results of a call to `find('list')` will be in the following form:

```
Array
(
    //[id] => 'displayValue',
    [1] => 'displayValue1',
    [2] => 'displayValue2',
    [4] => 'displayValue4',
    [5] => 'displayValue5',
    [6] => 'displayValue6',
    [3] => 'displayValue3',
)
```

When calling `find('list')` the `fields` passed are used to determine what should be used as the array key, value and optionally what to group the results by. By default the primary key for the model is used for the key, and the display field (which can be configured using the model attribute displayField) is used for the value. Some further examples to clarify:.

```
1.      function some_function() {
2.          ...
3.          $justusernames = $this->Article->User->find('list', array('fields' => array('User.username')));
4.          $usernameMap    =    $this->Article->User->find('list',    array('fields'    =>    array('User.username',
'User.first_name')));
5.          $usernameGroups    =    $this->Article->User->find('list',    array('fields'    =>    array('User.username',
'User.first_name', 'User.group')));
6.          ...
7.      }
```

With the above code example, the resultant vars would look something like this:

```
$justusernames = Array
(
    //[id] => 'username',
    [213] => 'AD7six',
    [25] => '_psychic_',
    [1] => 'PHPNut',
    [2] => 'gwoo',
    [400] => 'jperras',
)

$usernameMap = Array
(
    //[username] => 'firstname',
    ['AD7six'] => 'Andy',
    ['_psychic_'] => 'John',
    ['PHPNut'] => 'Larry',
    ['gwoo'] => 'Gwoo',
```

```
    ['jperras'] => 'Joël',
)

$usernameGroups = Array
(
    ['User'] => Array
        (
            ['PHPNut'] => 'Larry',
            ['gwoo'] => 'Gwoo',
        )

    ['Admin'] => Array
        (
            ['_psychic_'] => 'John',
            ['AD7six'] => 'Andy',
            ['jperras'] => 'Joël',
        )

)
```

3.7.3.1.5 find('threaded')

find('threaded', $params)

`find('threaded', $params)` returns a nested array, and is appropriate if you want to use the `parent_id` field of your model data to build nested results. Below are a couple of simple (controller code) examples:

```
1.       function some_function() {
2.          ...
3.          $allCategories = $this->Category->find('threaded');
4.          $aCategory = $this->Category->find('first', array('conditions' => array('parent_id' => 42))); // not the root
5.          $someCategories = $this->Category->find('threaded', array(
6.           'conditions' => array(
7.               'Article.lft >=' => $aCategory['Category']['lft'],
8.               'Article.rght <=' => $aCategory['Category']['rght']
```

```
9.              )
10.           ));
11.           ...
12.       }
```

It is not necessary to use the Tree behavior to use this method - but all desired results must be possible to be found in a single query.

In the above code example, `$allCategories` will contain a nested array representing the whole category structure. The second example makes use of the data structure used by the Tree behavior the return a partial, nested, result for `$aCategory` and everything below it. The results of a call to `find('threaded')` will be of the following form:

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

            [AssociatedModelName] => Array
                (
                    [id] => 1
                    [field1] => value1
                    [field2] => value2
                    [field3] => value3
                )
            [children] => Array
                (
                   [0] => Array
```

```
                        (
                            [ModelName] => Array
                                (
                                    [id] => 42
                                     [parent_id] => 83
                                    [field1] => value1
                                    [field2] => value2
                                    [field3] => value3
                                )

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

The order results appear can be changed as it is influence by the order of processing. For example, if `'order' => 'name ASC'` is passed in the params to `find('threaded')`, the results will appear in name order. Likewise any order can be used, there is no inbuilt requirement of this method for the top result to be returned first.

There are no additional parameters used by `find('threaded')`.

3.7.3.1.6 find('neighbors')

`find('neighbors', $params)`

'neighbors' will perform a find similar to 'first', but will return the row before and after the one you request. Below is a simple (controller code) example:

```
1.        function some_function() {
2.           $neighbors = $this->Article->find('neighbors', array('field' => 'id', 'value' => 3));
3.        }
```

You can see in this example the two required elements of the `$params` array: field and value. Other elements are still allowed as with any other find (Ex: If your model acts as containable, then you can specify 'contain' in `$params`). The format returned from a `find('neighbors')` call is in the form:

```
Array
(
    [prev] => Array
        (
            [ModelName] => Array
                (
                    [id] => 2
                    [field1] => value1
                    [field2] => value2
                    ...
                )
            [AssociatedModelName] => Array
                (
                    [id] => 151
                    [field1] => value1
                    [field2] => value2
                    ...
                )
        )
    [next] => Array
        (
            [ModelName] => Array
                (
                    [id] => 4
                    [field1] => value1
                    [field2] => value2
                    ...
                )
            [AssociatedModelName] => Array
```

```
                (
                    [id] => 122
                    [field1] => value1
                    [field2] => value2
                    ...
                )
        )
)
```

Note how the result always contains only two root elements: prev and next. This function does not honor a model's default recursive var. The recursive setting must be passed in the parameters on each call.

Does not honor the recursive attribute on a model. You must set the recursive param to utilize the recursive feature.

3.7.3.2 findAllBy

```
findAllBy<fieldName>(string $value, array $fields, array $order, int $limit, int $page, int $recursive)
```

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.

PHP5 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->findAllById(7);	Cake.id = 7
$this->User->findAllByUserName('psychic', array(), array('User.user_name => 'asc'));	User.user_name = 'psychic' ORDER BY User.user_name ASC

PHP4 users have to use this function a little differently due to some case-insensitivity in PHP4:

PHP4 findAllBy<x> Example	Corresponding SQL Fragment
$this->Product->findAllByOrder_status('3');	Product.order_status = 3
$this->Recipe->findAllByType('Cookie');	Recipe.type = 'Cookie'
$this->User->findAllByLast_name('Anderson');	User.last_name = 'Anderson'
$this->Cake->findAllById(7);	Cake.id = 7
$this->User->findAllByUser_name('psychic');	User.user_name = 'psychic'

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

3.7.3.3 findBy

```
findBy<fieldName>(string $value);
```

The findBy magic functions also accept some optional parameters:

```
findBy<fieldName>(string $value[, mixed $fields[, mixed $order]]);
```

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.

PHP5 findBy<x> Example	Corresponding SQL Fragment
$this->Product->findByOrderStatus('3');	Product.order_status = 3
$this->Recipe->findByType('Cookie');	Recipe.type = 'Cookie'

$this->User->findByLastName('Anderson');	User.last_name = 'Anderson'
$this->Cake->findById(7);	Cake.id = 7
$this->User->findByUserName('psychic');	User.user_name = 'psychic'

PHP4 users have to use this function a little differently due to some case-insensitivity in PHP4:

PHP4 findBy<x> Example	**Corresponding SQL Fragment**
$this->Product->findByOrder_status('3');	Product.order_status = 3
$this->Recipe->findByType('Cookie');	Recipe.type = 'Cookie'
$this->User->findByLast_name('Anderson');	User.last_name = 'Anderson'
$this->Cake->findById(7);	Cake.id = 7
$this->User->findByUser_name('psychic');	User.user_name = 'psychic'

findBy() functions like find('first',...), while findAllBy() functions like find('all',...).

In either case, the returned result is an array formatted just as it would be from find() or findAll(), respectively.

3.7.3.4 query

```
query(string $query)
```

SQL calls that you can't or don't want to make via other model methods (this should only rarely be necessary) can be made using the model's `query()` method.

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

`query()` does not honour $Model->cachequeries as its functionality is inherently disjoint from that of the calling model. To avoid caching calls to query, supply a second argument of false, ie: `query($query, $cachequeries = false)`

`query()` uses the table name in the query as the array key for the returned data, rather than the model name. For example,

```
1.  $this->Picture->query("SELECT * FROM pictures LIMIT 2;");
```

might return

```
1.  Array
2.  (
3.      [0] => Array
4.          (
5.              [pictures] => Array
6.                  (
7.                      [id] => 1304
8.                      [user_id] => 759
9.                  )
10.         )
11.     [1] => Array
12.         (
13.             [pictures] => Array
14.                 (
15.                     [id] => 1305
16.                     [user_id] => 759
17.                 )
18.         )
```

```
19.         )
```

To use the model name as the array key, and get a result consistent with that returned by the Find methods, the query can be rewritten:

```
1.  $this->Picture->query("SELECT * FROM pictures AS Picture LIMIT 2;");
```

which returns

```
1.  Array
2.  (
3.      [0] => Array
4.          (
5.              [Picture] => Array
6.                  (
7.                      [id] => 1304
8.                      [user_id] => 759
9.                  )
10.         )
11.     [1] => Array
12.         (
13.             [Picture] => Array
14.                 (
15.                     [id] => 1305
16.                     [user_id] => 759
17.                 )
18.         )
19. )
```

This syntax and the corresponding array structure is valid for MySQL only. Cake does not provide any data abstraction when running queries manually, so exact results will vary between databases.

3.7.3.5 field

field(string $name, array $conditions = null, string $order = null)

Returns the value of a single field, specified as `$name`, from the first record matched by $conditions as ordered by $order. If no conditions are passed and the model id is set, will return the field value for the current model result. If no matching record is found returns false.

```
1.        $this->Post->id = 22;
2.        echo $this->Post->field('name'); // echo the name for row id 22
3.        echo $this->Post->field('name', array('created <' => date('Y-m-d H:i:s')), 'created DESC'); // echo the name
of the last created instance
```

3.7.3.6 read()

read($fields, $id)

`read()` is a method used to set the current model data (`Model::$data`)--such as during edits--but it can also be used in other circumstances to retrieve a single record from the database.

`$fields` is used to pass a single field name, as a string, or an array of field names; if left empty, all fields will be fetched.

`$id` specifies the ID of the record to be read. By default, the currently selected record, as specified by `Model::$id`, is used. Passing a different value to `$id` will cause that record to be selected.

`read()` always returns an array (even if only a single field name is requested). Use `field` to retrieve the value of a single field.

```
1.        function beforeDelete($cascade) {
2.            ...
3.            $rating = $this->read('rating'); // gets the rating of the record being deleted.
```

```
4.          $name = $this->read('name', 2); // gets the name of a second record.
5.          $rating = $this->read('rating'); // gets the rating of the second record.
6.          $this->id = 3; //
7.          $this->Article->read(); // reads a third record
8.          $record = $this->data // stores the third record in $record
9.          ...
10.     }
```

Notice that the third call to `read()` fetches the rating of the same record read before. That is because `read()` changes `Model::$id` to any value passed as `$id`. Lines 6-8 demonstrate how `read()` changes the current model data. `read()` will also unset all validation errors on the model. If you would like to keep them, use `find('first')` instead.

3.7.3.7 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 discrete, 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:

```
1.      $conditions = array("Post.title" => "This is a post");
2.      //Example usage with a model:
3.      $this->Post->find('first', array('conditions' => $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":

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

Notice the '<>' that follows the field name. 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 field name and the operator. 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:

```
1.      array(
2.          "Post.title" => array("First post", "Second post", "Third post")
3.      )
```

To do a NOT IN(...) match to find posts where the title is not in the given set of values:

```
1.      array(
2.          "NOT" => array("Post.title" => array("First post", "Second post", "Third post"))
3.      )
```

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

```
1.      array (
2.          "Post.title" => array("First post", "Second post", "Third post"),
3.          "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
4.      )
```

You can also create finds that compare two fields in the database

```
1.      array("Post.created = Post.modified")
```

This above example will return posts where the created date is equal to the modified date (ie it will return posts that have never been modified).

Remember that if you find yourself unable to form a WHERE clause in this method (ex. boolean operations), you can always specify it as a string like:

```
1.      array(
2.          'Model.field & 8 = 1',
3.          //other conditions as usual
4.      )
```

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:

```
1.      array( "OR" => array (
2.          "Post.title" => array("First post", "Second post", "Third post"),
3.          "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
4.          )
5.      )
```

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 belongsTo relationship between Posts and Authors. 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:

```
1.      array (
2.          "Author.name" => "Bob",
3.          "OR" => array (
4.              "Post.title LIKE" => "%magic%",
5.              "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
6.          )
7.      )
```

If you need to set multiple conditions on the same field, like when you want to do a LIKE search with multiple terms, you can do so by using conditions similar to:

```
1.         array(
2.             'OR' => array(
3.             array('Post.title LIKE' => '%one%'),
4.             array('Post.title LIKE' => '%two%')
5.             )
6.         );
```

Cake can also check for null fields. In this example, the query will return records where the post title is not null:

```
1.         array ("NOT" => array (
2.                   "Post.title" => null
3.             )
4.         )
```

To handle BETWEEN queries, you can use the following:

```
1.         array('Post.id BETWEEN ? AND ?' => array(1,10))
```

Note: CakePHP will quote the numeric values depending on the field type in your DB.

How about GROUP BY?

```
1.         array('fields'=>array('Product.type','MIN(Product.price) as price'), 'group' => 'Product.type');
```

The data returned for this would be in the following format:

```
Array
(
    [0] => Array
        (
            [Product] => Array
                (
                    [type] => Clothing
                )
            [0] => Array
                (
                    [price] => 32
                )
        )
    [1] => Array....
```

A quick example of doing a DISTINCT query. You can use other operators, such as MIN(), MAX(), etc., in a similar fashion

```
array('fields'=>array('DISTINCT (User.name) AS my_column_name'), 'order'=>array('User.id DESC'));
```

You can create very complex conditions, by nesting multiple condition arrays:

```
array(
   'OR' => array(
      array('Company.name' => 'Future Holdings'),
      array('Company.city' => 'CA')
   ),
   'AND' => array(
      array(
         'OR'=>array(
            array('Company.status' => 'active'),
```

```
                'NOT'=>array(
                    array('Company.status'=> array('inactive', 'suspended'))
                )
            )
        )
    );
```

Which produces the following SQL:

```
SELECT `Company`.`id`, `Company`.`name`,
`Company`.`description`, `Company`.`location`,
`Company`.`created`, `Company`.`status`, `Company`.`size`
FROM
    `companies` AS `Company`
WHERE
    ((`Company`.`name` = 'Future Holdings')
    OR
    (`Company`.`name` = 'Steel Mega Works'))
AND
    ((`Company`.`status` = 'active')
    OR (NOT (`Company`.`status` IN ('inactive', 'suspended'))))
```

Sub-queries

For the example, imagine we have a "users" table with "id", "name" and "status". The status can be "A", "B" or "C". And we want to get all the users that have status different than "B" using sub-query.

In order to achieve that we are going to get the model data source and ask it to build the query as if we were calling a find method, but it will just return the SQL statement. After that we make an expression and add it to the conditions array.

```
$conditionsSubQuery['`User2`.`status`'] = 'B';
$dbo = $this->User->getDataSource();
$subQuery = $dbo->buildStatement(
    array(
        'fields' => array('`User2`.`id`'),
        'table' => $dbo->fullTableName($this->User),
        'alias' => 'User2',
        'limit' => null,
        'offset' => null,
        'joins' => array(),
        'conditions' => $conditionsSubQuery,
        'order' => null,
        'group' => null
    ),
    $this->User
);
$subQuery = ' `User`.`id` NOT IN (' . $subQuery . ') ';
$subQueryExpression = $dbo->expression($subQuery);
$conditions[] = $subQueryExpression;
$this->User->find('all', compact('conditions'));
```

This should generate the following SQL:

```
SELECT
    `User`.`id` ,
    `User`.`name` ,
    `User`.`status`
FROM
    `users` AS `User`
WHERE
    `User`.`id` NOT IN (
```

```
                SELECT
                    `User2`.`id`
                FROM
                    `users` AS `User2`
                WHERE
                    `User2`.`status` = 'B'
            )
```

Also, if you need to pass just part of your query as raw SQL as the above, datasource **expressions** with raw SQL work for any part of the find query.

3.7.4 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:

```
1.      function edit($id) {
2.          //Has any form data been POSTed?
3.          if(!empty($this->data)) {
4.              //If the form data can be validated and saved...
5.              if($this->Recipe->save($this->data)) {
```

```
6.                    //Set a session flash message and redirect.
7.                    $this->Session->setFlash("Recipe Saved!");
8.                    $this->redirect('/recipes');
9.                }
10.           }
11.
12.           //If no form data, find the recipe to be edited
13.           //and hand it to the view.
14.           $this->set('recipe', $this->Recipe->findById($id));
15.       }
```

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 are being broken.

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

set($one, $two = null)

Model::set() can be used to set one or many fields of data to the data array inside a model. This is useful when using models with the ActiveRecord features offered by Model.

```
1.        $this->Post->read(null, 1);
2.        $this->Post->set('title', 'New title for the article');
3.        $this->Post->save();
```

Is an example of how you can use `set()` to update and save single fields, in an ActiveRecord approach. You can also use `set()` to assign new values to multiple fields.

```
1.        $this->Post->read(null, 1);
2.        $this->Post->set(array(
```

```
3.            'title' => 'New title',
4.            'published' => false
5.        ));
6.        $this->Post->save();
```

The above would update the title and published fields and save them to the database.

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`.

If `$fieldList` is not supplied, a malicious user can add additional fields to the form data (if you are not using Security component), and by this change fields that were not originally intended to be changed.

The save method also has an alternate syntax:

save(array $data = null, array $params = array())

`$params` array can have any of the following available options as keys:

```
1.        array(
2.            'validate' => true,
3.            'fieldList' => array(),
4.            'callbacks' => true //other possible values are false, 'before', 'after'
5.        )
```

More information about model callbacks is available here

If you dont want the updated field to be updated when saving some data add `'updated' => false` to your `$data` array

Once a save has been completed, the ID for the object can be found in the `$id` attribute of the model object - something especially handy when creating new objects.

```
1.      $this->Ingredient->save($newData);
2.      $newIngredientId = $this->Ingredient->id;
```

Creating or updating is controlled by the model's `id` field. If `$Model->id` is set, the record with this primary key is updated. Otherwise a new record is created.

```
1.      //Create: id isn't set or is null
2.      $this->Recipe->create();
3.      $this->Recipe->save($this->data);
4.      //Update: id is set to a numerical value
5.      $this->Recipe->id = 2;
6.      $this->Recipe->save($this->data);
```

When calling save in a loop, don't forget to call `create()`.

`create(array $data = array())`

This method resets the model state for saving new information.

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

If `false` is passed instead of an array, the model instance will not initialize fields from the model schema that are not already set, it will only reset fields that have already been set, and leave the rest unset. Use this to avoid updating fields in the database that were already set and are intended to be updated.

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

Used to save a single field value. Set the ID of the model (`$this->ModelName->id = $id`) just before calling `saveField()`. When using this method, `$fieldName` should only contain the name of the field, not the name of the model and field.

For example, to update the title of a blog post, the call to `saveField` from a controller might look something like this:

```
1.      $this->Post->saveField('title', 'A New Title for a New Day');
```

You cant stop the updated field being updated with this method, you need to use the save() method.

updateAll(array $fields, array $conditions)

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

```
1.      $this_year = date('Y-m-d h:i:s', strtotime('-1 year'));
2.      $this->Baker->updateAll(
3.          array('Baker.approved' => true),
4.          array('Baker.created <=' => $this_year)
5.      );
```

The $fields array accepts SQL expressions. Literal values should be quoted manually.

Even if the modified field exist for the model being updated, it is not going to be updated automatically by the ORM. Just add it manually to the array if you need it to be updated.

For example, to close all tickets that belong to a certain customer:

```
1.      $this->Ticket->updateAll(
2.          array('Ticket.status' => "'closed'"),
3.          array('Ticket.customer_id' => 453)
4.      );
```

By default, updateAll() will automatically join any belongsTo association for databases that support joins. To prevent this, temporarily unbind the associations.

saveAll(array $data = null, array $options = array())

Used to save (a) multiple individual records for a single model or (b) this record, as well as all associated records

The following options may be used:

validate: Set to false to disable validation, true to validate each record before saving, 'first' to validate *all* records before any are saved (default), or 'only' to only validate the records, but not save them.

atomic: If true (default), will attempt to save all records in a single transaction. Should be set to false if database/table does not support transactions. If false, we return an array similar to the $data array passed, but values are set to true/false depending on whether each record saved successfully.

fieldList: Equivalent to the $fieldList parameter in `Model::save()`

For saving multiple records of single model, $data needs to be a numerically indexed array of records like this:

```
Array
```

```
(
    [Article] => Array(
                [0] => Array
                        (
                             [title] => title 1
                         )
                [1] => Array
                        (
                             [title] => title 2
                         )
                )
)
```

The command for saving the above $data array would look like this:

```
1.        $this->Article->saveAll($data['Article']);
```

Note that we are passing `$data['Article']` instead of usual `$data`. When saving multiple records of same model the records arrays should be just numerically indexed without the model key.

For saving a record along with its related record having a hasOne or belongsTo association, the data array should be like this:

```
Array
(
    [User] => Array
        (
            [username] => billy
        )
    [Profile] => Array
        (
            [sex] => Male
           [occupation] => Programmer
        )
)
```

The command for saving the above $data array would look like this:

```
1.        $this->Article->saveAll($data);
```

For saving a record along with its related records having hasMany association, the data array should be like this:

```
Array
(
    [Article] => Array
        (
            [title] => My first article
        )
    [Comment] => Array
        (
            [0] => Array
                (
                    [comment] => Comment 1
                    [user_id] => 1
                )
            [1] => Array
                (
                    [comment] => Comment 2
                    [user_id] => 2
                )
        )
)
```

The command for saving the above $data array would look like this:

```
1.        $this->Article->saveAll($data);
```

Saving related data with `saveAll()` will only work for directly associated models. If successful, `last_insert_id()`'s will be stored in the related models id field, i.e. `$this->RelatedModel->id`.

Calling a saveAll before another saveAll has completed will cause the first saveAll to return false. One or both of the saveAll calls must have atomic set to false to correct this behavior.

3.7.4.1 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.

```
1.  <?php
2.  function add() {
3.      if (!empty($this->data)) {
4.          // We can save the User data:
5.          // it should be in $this->data['User']
6.
7.          $user = $this->User->save($this->data);
8.          // If the user was saved, Now we add this information to the data
9.          // and save the Profile.
10.
11.         if (!empty($user)) {
12.             // The ID of the newly created user has been set
13.             // as $this->User->id.
14.             $this->data['Profile']['user_id'] = $this->User->id;
15.             // Because our User hasOne Profile, we can access
16.             // the Profile model through the User model:
17.             $this->User->Profile->save($this->data);
```

```
18.                 }
19.             }
20.         }
21.         ?>
```

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 the `$id` attribute of the model class 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.

To supplement the basic approach used above, CakePHP also offers a very handy method `saveAll()`, which allows you to validate and save multiple models in one shot. In addition, `saveAll()` provides transactional support to ensure data integrity in your database (i.e. if one model fails to save, the other models will not be saved either).

For transactions to work correctly in MySQL your tables must use InnoDB engine. Remember that MyISAM tables do not support transactions.

Let's see how we can use `saveAll()` to save Company and Account models at the same time.

First, you need to build your form for both Company and Account models (we'll assume that Company hasMany Account).

```
1.         echo $form->create('Company', array('action'=>'add'));
2.         echo $form->input('Company.name', array('label'=>'Company name'));
3.         echo $form->input('Company.description');
4.         echo $form->input('Company.location');
5.         echo $form->input('Account.0.name', array('label'=>'Account name'));
6.         echo $form->input('Account.0.username');
7.         echo $form->input('Account.0.email');
8.         echo $form->end('Add');
```

Take a look at the way we named the form fields for the Account model. If Company is our main model, `saveAll()` will expect the related model's (Account) data to arrive in a specific format. And having `Account.0.fieldName` is exactly what we need.

The above field naming is required for a hasMany association. If the association between the models is hasOne, you have to use ModelName.fieldName notation for the associated model.

Now, in our companies_controller we can create an `add()` action:

```
1.      function add() {
2.         if(!empty($this->data)) {
3.            //Use the following to avoid   validation errors:
4.            unset($this->Company->Account->validate['company_id']);
5.            $this->Company->saveAll($this->data, array('validate'=>'first'));
6.         }
7.      }
```

That's all there is to it. Now our Company and Account models will be validated and saved all at the same time. A quick thing to point out here is the use of `array('validate'=>'first')`; this option will ensure that both of our models are validated. Note that `array('validate'=>'first')` is the default option on cakephp 1.3.

3.7.4.1.1 counterCache - Cache your count()

This function helps you cache the count of related data. Instead of counting the records manually via `find('count')`, the model itself tracks any addition/deleting towards the associated `$hasMany` model and increases/decreases a dedicated integer field within the parent model table.

The name of the field consists of the singular model name followed by a underscore and the word "count".

```
1.      my_model_count
```

Let's say you have a model called `ImageComment` and a model called `Image`, you would add a new INT-field to the `image` table and name it `image_comment_count`.

Here are some more examples:

Model	Associated Model	Example
User	Image	users.image_count
Image	ImageComment	images.image_comment_count
BlogEntry	BlogEntryComment	blog_entries.blog_entry_comment_count

Once you have added the counter field you are good to go. Activate counter-cache in your association by adding a `counterCache` key and set the value to `true`.

```
1.  class Image extends AppModel {
2.      var $belongsTo = array(
3.          'ImageAlbum' => array('counterCache' => true)
4.      );
5.  }
```

From now on, every time you add or remove a `Image` associated to `ImageAlbum`, the number within `image_count` is adjusted automatically.

You can also specify `counterScope`. It allows you to specify a simple condition which tells the model when to update (or when not to, depending on how you look at it) the counter value.

Using our Image model example, we can specify it like so:

```
1.  class Image extends AppModel {
```

```
2.            var $belongsTo = array(
3.                'ImageAlbum' => array(
4.                    'counterCache' => true,
5.                    'counterScope' => array('Image.active' => 1) // only count if "Image" is active = 1
6.            ));
7.        }
```

3.7.4.2 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, you need to set the ID of the associated model in your data array. We'll build a form that creates a new tag and associates it on the fly with some recipe.

The simplest form might look something like this (we'll assume that $recipe_id is already set to something):

```
1.        <?php echo $form->create('Tag');?>
2.            <?php echo $form->input(
3.                'Recipe.id',
4.                array('type'=>'hidden', 'value' => $recipe_id)); ?>
5.            <?php echo $form->input('Tag.name'); ?>
6.            <?php echo $form->end('Add Tag'); ?>
```

In this example, you can see the `Recipe.id` hidden field whose value is set to the ID of the recipe we want to link the tag to.

When the `save()` method is invoked within the controller, it'll automatically save the HABTM data to the database.

```
1.        function add() {
2.
3.            //Save the association
```

```
4.            if ($this->Tag->save($this->data)) {
5.                //do something on success
6.            }
7.        }
```

With the preceding code, our new Tag is created and associated with a Recipe, whose ID was set in $this->data['Recipe']['id'].

Other ways we might want to present our associated data can include a select drop down list. The data can be pulled from the model using the `find('list')` method and assigned to a view variable of the model name. An input with the same name will automatically pull in this data into a `<select>`.

```
1.        // in the controller:
2.        $this->set('tags', $this->Recipe->Tag->find('list'));
3.        // in the view:
4.        $form->input('tags');
```

A more likely scenario with a HABTM relationship would include a `<select>` set to allow multiple selections. For example, a Recipe can have multiple Tags assigned to it. In this case, the data is pulled out of the model the same way, but the form input is declared slightly different. The tag name is defined using the `ModelName` convention.

```
1.        // in the controller:
2.        $this->set('tags', $this->Recipe->Tag->find('list'));
3.        // in the view:
4.        $form->input('Tag');
```

Using the preceding code, a multiple select drop down is created, allowing for multiple choices to automatically be saved to the existing Recipe being added or saved to the database.

What to do when HABTM becomes complicated?

By default when saving a HasAndBelongsToMany relationship, Cake will delete all rows on the join table before saving new ones. For example if you have a Club that has 10 Children associated. You then update the Club with 2 children. The Club will only have 2 Children, not 12.

Also note that if you want to add more fields to the join (when it was created or meta information) this is possible with HABTM join tables, but it is important to understand that you have an easy option.

HasAndBelongsToMany between two models is in reality shorthand for three models associated through both a hasMany and a belongsTo association.

Consider this example:

```
1.      Child hasAndBelongsToMany Club
```

Another way to look at this is adding a Membership model:

```
1.      Child hasMany Membership
2.      Membership belongsTo Child, Club
3.      Club hasMany Membership.
```

These two examples are almost the exact same. They use the same amount and named fields in the database and the same amount of models. The important differences are that the "join" model is named differently and its behavior is more predictable.

When your join table contains extra fields besides two foreign keys, in most cases it's easier to make a model for the join table and setup hasMany, belongsTo associations as shown in example above instead of using HABTM association.

3.7.5 Deleting Data

These methods can be used to remove data.

3.7.5.1 delete

```
delete(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 (User 'hasMany' or 'hasAndBelongsToMany' Recipes):

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

3.7.5.2 deleteAll

```
deleteAll(mixed $conditions, $cascade = true, $callbacks = false)
```

Same as with `delete()` 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.

conditions	Conditions	to match
cascade	Boolean, Set to true to	delete records that depend on this record
callbacks	Boolean,	Run callbacks

Return boolean True on success, false on failure

3.7.6 Associations: Linking Models Together

One of the most powerful features of CakePHP is the ability to link 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.

While data can come from a variety of sources, the most common form of storage in web applications is a relational database. Most of what this section covers will be in that context.

For information on associations with Plugin models, see Plugin Models.

3.7.6.1 Relationship Types

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

Relationship	Association Type	Example
one to one	hasOne	A user has one profile.
one to many	hasMany	A user can have multiple recipes.
many to one	belongsTo	Many recipes belong to a user.
many to many	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.

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $hasOne = 'Profile';
5.      var $hasMany = array(
6.          'Recipe' => array(
7.              'className'  => 'Recipe',
8.              'conditions' => array('Recipe.approved' => '1'),
9.              'order'      => 'Recipe.created DESC'
```

```
            )
        );
    }
    ?>
```

In the above example, the first instance of the word 'Recipe' is what is termed an 'Alias'. This is an identifier for the relationship and can be anything you choose. Usually, you will choose the same name as the class that it references. However, **aliases for each model must be unique app wide**. E.g. it is appropriate to have

```
<?php
class User extends AppModel {
    var $name = 'User';
    var $hasMany = array(
        'MyRecipe' => 'Recipe',
    );
    var $hasAndBelongsToMany => array('Member' => 'User');
}
class Group extends AppModel {
    var $name = 'Group';
    var $hasMany = array(
        'MyRecipe' => array(
            'className'  => 'Recipe',
        )
    );
    var $hasAndBelongsToMany => array('MemberOf' => 'Group');
}
?>
```

but the following will not work well in all circumstances:

```
<?php
class User extends AppModel {
```

```
3.              var $name = 'User';
4.              var $hasMany = array(
5.                  'MyRecipe' => 'Recipe',
6.              );
7.              var $hasAndBelongsToMany => array('Member' => 'User');
8.          }
9.          class Group extends AppModel {
10.             var $name = 'Group';
11.             var $hasMany = array(
12.                 'MyRecipe' => array(
13.                     'className'  => 'Recipe',
14.                 )
15.             );
16.             var $hasAndBelongsToMany => array('Member' => 'Group');
17.         }
18.         ?>
```

because here we have the alias 'Member' referring to both the User (in Group) and the Group (in User) model in the HABTM associations. Choosing non-unique names for model aliases across models can cause unexpected behavior.

Cake will automatically create links between associated model objects. So for example in your `User` model you can access the `Recipe` model as

```
1.          $this->Recipe->someFunction();
```

Similarly in your controller you can access an associated model simply by following your model associations and without adding it to the `$uses` array:

```
1.          $this->User->Recipe->someFunction();
```

Remember that associations are defined 'one way'. If you define User hasMany Recipe that has no effect on the Recipe Model. You need to define Recipe belongsTo User to be able to access the User model from your Recipe model

3.7.6.2 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.

Relation	Schema
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.

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $hasOne = 'Profile';
5.  }
6.  ?>
```

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 limit the association to include only certain records.

```
<?php
class User extends AppModel {
    var $name = 'User';
    var $hasOne = array(
        'Profile' => array(
            'className'    => 'Profile',
            'conditions'   => array('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 current model, suffixed with '_id'. In the example above it would default to 'user_id'.
- **conditions**: An SQL fragment used to 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.
- **order**: An SQL fragment that defines the sorting order for the returned associated rows.
- **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
        )
)
```

3.7.6.3 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.

Relation	Schema
Banana belongsTo Apple	bananas.apple_id
Profile belongsTo User	profiles.user_id
Mentor belongsTo Doctor	mentors.doctor_id

If a model(table) contains a foreign key, it belongsTo the other model(table).

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 $belongsTo = '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 current 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 to 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."
- **type**: the type of the join to use in the SQL query, default is LEFT which may not fit your needs in all situations, INNER may be helpful when you want everything from your main and associated models or nothing at all!(effective when used with some conditions of course). **(NB: type value is in lower case - i.e. left, inner)**
- **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.
- **counterCache**: If set to true the associated Model will automatically increase or decrease the "[singular_model_name]_count" field in the foreign table whenever you do a save() or delete(). If its a string then its the field name to use. The value in the counter field represents the number of related rows.
- **counterScope**: Optional conditions array to use for updating counter cache field.

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
        )
)
```

3.7.6.4 hasMany

Next step: defining a “User hasMany Comment” association. A hasMany association 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.

Relation	Schema
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:

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $hasMany = 'Comment';
5.  }
6.  ?>
```

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

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $hasMany = array(
5.          'Comment' => array(
```

```
                'className'     => 'Comment',
                'foreignKey'    => 'user_id',
                'conditions'    => array('Comment.status' => '1'),
                'order'    => 'Comment.created DESC',
                'limit'        => '5',
                'dependent'=> true
            )
        );
    }
    ?>
```

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 actual model, suffixed with '_id'.
- **conditions**: An SQL fragment used to 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.
- **exclusive**: When exclusive is set to true, recursive model deletion does the delete with a deleteAll() call, instead of deleting each entity separately. This greatly improves performance, but may not be ideal for all circumstances.
- **finderQuery**: A complete SQL query CakePHP can use to fetch associated model records. This should be used in situations that require very custom results.

If a query you're building requires a reference to the associated model ID, use the special `{$__cakeID__$}` marker in the query. For example, if your Apple model hasMany Orange, the query should look something like this:

```
1.        SELECT Orange.* from oranges as Orange WHERE Orange.apple_id = {$__cakeID__$};
```

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] => 124
                    [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.

3.7.6.5 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, and separated with an underscore (_). The contents of the table should be two fields, each foreign keys (which should be integers) pointing to both of the primary keys of the involved models. To avoid any issues - don't define a combined primary key for these two fields, if your application requires it you can define a unique index. If you plan to add any extra information to this table, it's a good idea to add an additional primary key field (by convention 'id') to make acting on the table as easy as any other model.

HABTM requires a separate join table that includes both *model* names.

Relation	Schema (HABTM table in bold)
Recipe HABTM Tag	**recipes_tags.**id, **recipes_tags.**recipe_id, **recipes_tags.**tag_id
Cake HABTM Fan	**cakes_fans.**id, **cakes_fans.**cake_id, **cakes_fans.**fan_id
Foo HABTM Bar	**bars_foos.**id, **bars_foos.**foo_id, **bars_foos.**bar_id

Table names are by convention in alphabetical order.

Make sure primary keys in tables **cakes** and **recipes** have "id" fields as assumed by convention. If they're different than assumed, it has to be changed in model

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:

```
1.  <?php
2.  class Recipe extends AppModel {
3.      var $name = 'Recipe';
4.      var $hasAndBelongsToMany = array(
5.          'Tag' =>
6.              array(
7.                  'className'              => 'Tag',
8.                  'joinTable'              => 'recipes_tags',
9.                  'foreignKey'             => 'recipe_id',
10.                 'associationForeignKey'  => 'tag_id',
11.                 'unique'                 => true,
12.                 'conditions'             => '',
13.                 'fields'                 => '',
14.                 'order'                  => '',
15.                 'limit'                  => '',
```

```
16.                     'offset'                => '',
17.                     'finderQuery'           => '',
18.                     'deleteQuery'           => '',
19.                     'insertQuery'           => ''
20.                 )
21.         );
22.     }
23.     ?>
```

Possible keys for HABTM association arrays include:

- **className**: the classname of the model being associated to the current model. If you're defining a 'Recipe HABTM Tag' relationship, the className key should equal 'Tag.'
- **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).
- **with**: Defines the name of the model for the join table. By default CakePHP will auto-create a model for you. Using the example above it would be called RecipesTag. By using this key you can override this default name. The join table model can be used just like any "regular" model to access the join table directly.
- **foreignKey**: 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'.
- **associationForeignKey**: 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'.
- **unique**: If true (default value) cake will first delete existing relationship records in the foreign keys table before inserting new ones, when updating a record. So existing associations need to be passed again when updating.
- **conditions**: An SQL fragment used to 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, deleteQuery, insertQuery**: A complete SQL query CakePHP can use to fetch, delete, or create new 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] => 124
                    [name] => Dessert
                )
           [2] => Array
                (
                    [id] => 125
                    [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.

It is also possible to execute custom find queries based on HABTM relationships. Consider the following examples:

Assuming the same structure in the above example (Recipe HABTM Tag), let's say we want to fetch all Recipes with the tag 'Dessert', one potential (wrong) way to achieve this would be to apply a condition to the association itself:

```
1.        $this->Recipe->bindModel(array(
2.            'hasAndBelongsToMany' => array(
3.                'Tag' => array('conditions'=>array('Tag.name'=>'Dessert'))
4.        )));
5.        $this->Recipe->find('all');
```

```
//Data Returned
Array
(
    0 => 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] => 124
                        [name] => Dessert
                    )
            )
    )
    1 => Array
        {
        [Recipe] => Array
            (
                [id] => 2745
```

```
                [name] => Crab Cakes
                [created] => 2008-05-01 10:31:01
                [user_id] => 2349
            )
        [Tag] => Array
            (
            )
        }
}
```

Notice that this example returns ALL recipes but only the "Dessert" tags. To properly achieve our goal, there are a number of ways to do it. One option is to search the Tag model (instead of Recipe), which will also give us all of the associated Recipes.

```
1.      $this->Recipe->Tag->find('all', array('conditions'=>array('Tag.name'=>'Dessert')));
```

We could also use the join table model (which CakePHP provides for us), to search for a given ID.

```
1.      $this->Recipe->bindModel(array('hasOne' => array('RecipesTag')));
2.      $this->Recipe->find('all', array(
3.              'fields' => array('Recipe.*'),
4.              'conditions'=>array('RecipesTag.tag_id'=>124) // id of Dessert
5.      ));
```

It's also possible to create an exotic association for the purpose of creating as many joins as necessary to allow filtering, for example:

```
1.      $this->Recipe->bindModel(array(
2.          'hasOne' => array(
3.              'RecipesTag',
4.              'FilterTag' => array(
5.                  'className' => 'Tag',
6.                  'foreignKey' => false,
7.                  'conditions' => array('FilterTag.id = RecipesTag.tag_id')
```

```
8.        ))));
9.        $this->Recipe->find('all', array(
10.               'fields' => array('Recipe.*'),
11.               'conditions'=>array('FilterTag.name'=>'Dessert')
12.       ));
```

Both of which will return the following data:

```
//Data Returned
Array
(
    0 => 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] => 124
                    [name] => Dessert
                )
           [2] => Array
                (
                    [id] => 125
                    [name] => Heart Disease
```

```
                    )
            )
}
```

The same binding trick can be used to easily paginate your HABTM models. Just one word of caution: since paginate requires two queries (one to count the records and one to get the actual data), be sure to supply the `false` parameter to your `bindModel();` which essentially tells CakePHP to keep the binding persistent over multiple queries, rather than just one as in the default behavior. Please refer to the API for more details.

For more information on saving HABTM objects see Saving Related Model Data (HABTM)

For more information on binding model associations on the fly see Creating and destroying associations on the fly

Mix and match techniques to achieve your specific objective.

3.7.6.6 hasMany through (The Join Model)

It is sometimes desirable to store additional data with a many to many association. Consider the following

Student hasAndBelongsToMany Course Course hasAndBelongsToMany Student

In other words, a Student can take many Courses and a Course can be taken my many Students. This is a simple many to many association demanding a table such as this

```
id | student_id | course_id
```

Now what if we want to store the number of days that were attended by the student on the course and their final grade? The table we'd want would be

```
id | student_id | course_id | days_attended | grade
```

The trouble is, hasAndBelongsToMany will not support this type of scenario because when hasAndBelongsToMany associations are saved, the association is deleted first. You would lose the extra data in the columns as it is not replaced in the new insert.

The way to implement our requirement is to use a **join model**, otherwise known (in Rails) as a **hasMany through** association. That is, the association is a model itself. So, we can create a new model CourseMembership. Take a look at the following models.

```
1.  student.php
2.
3.  class Student extends AppModel
4.  {
5.      public $hasMany = array(
6.          'CourseMembership'
7.      );
8.      public $validate = array(
9.          'first_name' => array(
10.             'rule' => 'notEmpty',
11.             'message' => 'A first name is required'
12.         ),
13.         'last_name' => array(
14.             'rule' => 'notEmpty',
15.             'message' => 'A last name is required'
16.         )
17.     );
18. }
19.
20. course.php
21.
22. class Course extends AppModel
23. {
24.     public $hasMany = array(
25.         'CourseMembership'
26.     );
27.     public $validate = array(
28.         'name' => array(
```

```
                        'rule' => 'notEmpty',
                        'message' => 'A course name is required'
                    )
                );
            }

            course_membership.php
            class CourseMembership extends AppModel
            {
                public $belongsTo = array(
                    'Student', 'Course'
                );
                public $validate = array(
                    'days_attended' => array(
                        'rule' => 'numeric',
                        'message' => 'Enter the number of days the student attended'
                    ),
                    'grade' => array(
                        'rule' => 'notEmpty',
                        'message' => 'Select the grade the student received'
                    )
                );
            }
```

The CourseMembership join model uniquely identifies a given Student's participation on a Course in addition to extra meta-information.

Working with join model data

Now that the models have been defined, let's see how we can save all of this. Let's say the Head of Cake School has asked us the developer to write an application that allows him to log a student's attendance on a course with days attended and grade. Take a look at the following code.

```
1.      controllers/course_membership_controller.php
2.
3.          class CourseMembershipsController extends AppController
4.          {
5.              public $uses = array('CourseMembership');
6.
7.              public function index() {
8.                  $this->set('course_memberships_list', $this->CourseMembership->find('all'));
9.              }
10.
11.             public function add() {
12.
13.                 if (! empty($this->data)) {
14.
15.                     if ($this->CourseMembership->saveAll(
16.                         $this->data, array('validate' => 'first'))) {
17.
18.                         $this->redirect(array('action' => 'index'));
19.                     }
20.                 }
21.             }
22.         }
23.
24.         views/course_memberships/add.ctp
25.         <?php echo $form->create('CourseMembership'); ?>
26.             <?php echo $form->input('Student.first_name'); ?>
27.             <?php echo $form->input('Student.last_name'); ?>
28.             <?php echo $form->input('Course.name'); ?>
29.             <?php echo $form->input('CourseMembership.days_attended'); ?>
30.             <?php echo $form->input('CourseMembership.grade'); ?>
31.             <button type="submit">Save</button>
```

```
32.          <?php echo $form->end(); ?>
```

You can see that the form uses the form helper's dot notation to build up the data array for the controller's save which looks a bit like this when submitted.

```
Array
(
    [Student] => Array
        (
            [first_name] => Joe
            [last_name] => Bloggs
        )

    [Course] => Array
        (
            [name] => Cake
        )

    [CourseMembership] => Array
        (
            [days_attended] => 5
            [grade] => A
        )

)
```

Cake will happily be able to save the lot together and assigning the foreign keys of the Student and Course into CourseMembership with a saveAll call with this data structure. If we run the index action of our CourseMembershipsController the data structure received now from a find('all') is:

```
Array
(
    [0] => Array
        (
            [CourseMembership] => Array
                (
                    [id] => 1
                    [student_id] => 1
```

```
                            [course_id] => 1
                            [days_attended] => 5
                            [grade] => A
                        )

                    [Student] => Array
                        (
                            [id] => 1
                            [first_name] => Joe
                            [last_name] => Bloggs
                        )

                    [Course] => Array
                        (
                            [id] => 1
                            [name] => Cake
                        )

            )

    )
```

There are of course many ways to work with a join model. The version above assumes you want to save everything at-once. There will be cases where you want to create the Student and Course independently and at a later point associate the two together with a CourseMembership. So you might have a form that allows selection of existing students and courses from picklists or ID entry and then the two meta-fields for the CourseMembership, e.g.

```
1.      views/course_memberships/add.ctp
2.
3.          <?php echo $form->create('CourseMembership'); ?>
4.              <?php echo $form->input('Student.id', array('type' => 'text', 'label' => 'Student ID', 'default' =>
    1)); ?>
5.              <?php echo $form->input('Course.id', array('type' => 'text', 'label' => 'Course ID', 'default' =>
    1)); ?>
6.              <?php echo $form->input('CourseMembership.days_attended'); ?>
7.              <?php echo $form->input('CourseMembership.grade'); ?>
```

```
8.              <button type="submit">Save</button>
9.          <?php echo $form->end(); ?>
```

And the resultant POST

```
Array
(
    [Student] => Array
        (
            [id] => 1
        )

    [Course] => Array
        (
            [id] => 1
        )

    [CourseMembership] => Array
        (
            [days_attended] => 10
            [grade] => 5
        )

)
```

Again Cake is good to us and pulls the Student id and Course id into the CourseMembership with the saveAll.

Join models are pretty useful things to be able to use and Cake makes it easy to do so with its built-in hasMany and belongsTo associations and saveAll feature.

3.7.6.7 Creating and Destroying Associations on the Fly

Sometimes it becomes necessary to create and destroy model associations on the fly. This may be for any number of reasons:

- You want to reduce the amount of associated data fetched, but all your associations are on the first level of recursion.

- You want to change the way an association is defined in order to sort or filter associated data.

This association creation and destruction is done using the CakePHP model bindModel() and unbindModel() methods. (There is also a very helpful behavior called "Containable", please refer to manual section about Built-in behaviors for more information). Let's set up a few models so we can see how bindModel() and unbindModel() work. We'll start with two models:

```
1.  <?php
2.  class Leader extends AppModel {
3.      var $name = 'Leader';
4.
5.      var $hasMany = array(
6.          'Follower' => array(
7.              'className' => 'Follower',
8.              'order'     => 'Follower.rank'
9.          )
10.     );
11. }
12. ?>
13.
14. <?php
15. class Follower extends AppModel {
16.     var $name = 'Follower';
17. }
18. ?>
```

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

```
1.  function someAction() {
```

```
        // This fetches Leaders, and their associated Followers
        $this->Leader->find('all');

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

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

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

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

Removing or adding associations using bind- and unbindModel() only works for the *next* find operation only unless the second parameter has been set to false. If the second parameter has been set to *false*, the bind remains in place for the remainder of the request.

Here's the basic usage pattern for unbindModel():

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

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

```
1.      function anotherAction() {
2.          // There is no Leader hasMany Principles in
3.          // the leader.php model file, so a find here,
4.          // only fetches Leaders.
5.          $this->Leader->find('all');
6.
7.          // Let's use bindModel() to add a new association
8.          // to the Leader model:
9.          $this->Leader->bindModel(
10.             array('hasMany' => array(
11.                     'Principle' => array(
12.                         'className' => 'Principle'
13.                     )
14.                 )
15.             )
16.         );
17.
18.         // Now that we're associated correctly,
19.         // we can use a single find function to fetch
20.         // Leaders with their associated principles:
21.         $this->Leader->find('all');
22.     }
```

There you have it. The basic usage for bindModel() is the encapsulation of a normal association array inside an array whose key is named after the type of association you are trying to create:

```
1.        $this->Model->bindModel(
2.                array('associationName' => array(
3.                        'associatedModelClassName' => array(
4.                            // normal association keys go here...
5.                        )
6.                    )
7.                )
8.            );
```

Even though the newly bound model doesn't need any sort of association definition in its model file, it will still need to be correctly keyed in order for the new association to work properly.

3.7.6.8 Multiple relations to the same model

There are cases where a Model has more than one relation to another Model. For example you might have a Message model that has two relations to the User model. One relation to the user that sends a message, and a second to the user that receives the message. The messages table will have a field user_id, but also a field recipient_id. Now your Message model can look something like:

```
1.        <?php
2.        class Message extends AppModel {
3.            var $name = 'Message';
4.            var $belongsTo = array(
5.                'Sender' => array(
6.                    'className' => 'User',
7.                    'foreignKey' => 'user_id'
8.                ),
9.                'Recipient' => array(
10.                   'className' => 'User',
11.                   'foreignKey' => 'recipient_id'
12.               )
13.           );
```

```
14.         }
15.         ?>
```

Recipient is an alias for the User model. Now let's see what the User model would look like.

```
1.          <?php
2.          class User extends AppModel {
3.              var $name = 'User';
4.              var $hasMany = array(
5.                  'MessageSent' => array(
6.                      'className' => 'Message',
7.                      'foreignKey' => 'user_id'
8.                  ),
9.                  'MessageReceived' => array(
10.                     'className' => 'Message',
11.                     'foreignKey' => 'recipient_id'
12.                 )
13.             );
14.         }
15.         ?>
```

It is also possible to create self associations as shown below.

```
1.          <?php
2.          class Post extends AppModel {
3.              var $name = 'Post';
4.
5.              var $belongsTo = array(
6.                  'Parent' => array(
7.                      'className' => 'Post',
```

```
                    'foreignKey' => 'parent_id'
                )
        );
        var $hasMany = array(
            'Children' => array(
                'className' => 'Post',
                'foreignKey' => 'parent_id'
            )
        );
    }
    ?>
```

An alternate method of associating a model with itself (without assuming a parent/child relationship) is to have both the `$belongsTo` and `$hasMany` relationships of a model each to declare an identical alias, className, and foreignKey [property].

```
    <?php
    class MySchema extends CakeSchema {
        public $users = array (
            'id' => array ('type' => 'integer', 'default' => null, 'key' => 'primary'),
            'username' => array ('type' => 'string', 'null' => false, 'key' => 'index'),
            // more schema properties...
            'last_user_id' => array ('type' => 'integer', 'default' => null, 'key' => 'index'),
            'indexes' => array (
                'PRIMARY' => array ('column' => 'id', 'unique' => true),
                // more keys...
                'last_user' => array ('column' => 'last_user_id', 'unique' => false)
            )
        );
    }
    class User extends AppModel {
```

```
16.         public $hasMany = array (
17.             'Tag' => array (
18.                 'foreignKey' => 'last_user_id'
19.             ),
20.             // more hasMany relationships...
21.             'LastUser' => array (
22.                 'className' => 'User',
23.                 'foreignKey' => 'last_user_id'
24.             )
25.         );
26.         public $belongsTo = array (
27.             // in most cases this would be the only belongsTo relationship for this model
28.             'LastUser' => array (
29.                 'className' => 'User',
30.                 'foreignKey' => 'last_user_id',
31.                 'dependent' => true
32.             )
33.         );
34.     }
35.     ?>
```

Reasoning [for this particular self-association method]: Say there are many models which contain the property `$modelClass.lastUserId`. Each model has the foreign key `last_user_id`, a reference to the last user that updated/modified the record in question. The model `User` *also contains* the same property (last_user_id), since it may be neat to know if someone has committed a security breach through the modification of any User record other than their own (you could also use strict ACL behaviors).

Fetching a nested array of associated records:

If your table has `parent_id` field you can also use `find('threaded')` to fetch nested array of records using a single query without setting up any associations.

3.7.6.9 Joining tables

In SQL you can combine related tables using the JOIN statement. This allows you to perform complex searches across multiples tables (i.e: search posts given several tags).

In CakePHP some associations (belongsTo and hasOne) performs automatic joins to retrieve data, so you can issue queries to retrieve models based on data in the related one.

But this is not the case with hasMany and hasAndBelongsToMany associations. Here is where forcing joins comes to the rescue. You only have to define the necessary joins to combine tables and get the desired results for your query.

Remember you need to set the recursion to -1 for this to work. I.e: $this->Channel->recursive = -1;

To force a join between tables you need to use the "modern" syntax for Model::find(), adding a `'joins'` key to the `$options` array. For example:

```
1.      $options['joins'] = array(
2.          array('table' => 'channels',
3.              'alias' => 'Channel',
4.              'type' => 'LEFT',
5.              'conditions' => array(
6.                  'Channel.id = Item.channel_id',
7.              )
8.          )
9.      );
10.     $Item->find('all', $options);
```

Note that the 'join' arrays are not keyed.

In the above example, a model called Item is left joined to the channels table. You can alias the table with the Model name, so the retrieved data complies with the CakePHP data structure.

The keys that define the join are the following:

- **table**: The table for the join.
- **alias**: An alias to the table. The name of the model associated with the table is the best bet.
- **type**: The type of join: inner, left or right.
- **conditions**: The conditions to perform the join.

With joins, you could add conditions based on Related model fields:

```
1.      $options['joins'] = array(
2.          array('table' => 'channels',
3.              'alias' => 'Channel',
4.              'type' => 'LEFT',
5.              'conditions' => array(
6.                  'Channel.id = Item.channel_id',
7.              )
8.          )
9.      );
10.     $options['conditions'] = array(
11.         'Channel.private' => 1
12.     );
13.     $privateItems = $Item->find('all', $options);
```

You could perform several joins as needed in hasBelongsToMany:

Suppose a Book hasAndBelongsToMany Tag association. This relation uses a books_tags table as join table, so you need to join the books table to the books_tags table, and this with the tags table:

```
1.      $options['joins'] = array(
2.          array('table' => 'books_tags',
3.              'alias' => 'BooksTag',
```

```
4.                'type' => 'inner',
5.                'conditions' => array(
6.                    'Books.id = BooksTag.books_id'
7.                )
8.            ),
9.            array('table' => 'tags',
10.               'alias' => 'Tag',
11.               'type' => 'inner',
12.               'conditions' => array(
13.                   'BooksTag.tag_id = Tag.id'
14.               )
15.           )
16.       );
17.       $options['conditions'] = array(
18.           'Tag.tag' => 'Novel'
19.       );
20.       $books = $Book->find('all', $options);
```

Using joins with Containable behavior could lead to some SQL errors (duplicate tables), so you need to use the joins method as an alternative for Containable if your main goal is to perform searches based on related data. Containable is best suited to restricting the amount of related data brought by a find statement.

3.7.7 Callback Methods

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.

3.7.7.1 beforeFind

`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 `$queryData` options), return *false*. Otherwise, return the possibly modified `$queryData`, or anything you want to get passed to find and its counterparts.

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

3.7.7.2 afterFind

`afterFind(array $results, bool $primary)`

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, i.e. something like:

```
1.        $results = array(
2.          0 => array(
3.            'ModelName' => array(
4.              'field1' => 'value1',
5.              'field2' => 'value2',
6.            ),
7.          ),
8.        );
```

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

The `$primary` parameter indicates whether or not the current model was the model that the query originated on or whether or not this model was queried as an association. If a model is queried as an assocation the format of `$results` can differ; instead of the result you would normally get from a find operation, you may get this:

```
1.        $results = array(
2.          'field_1' => 'value1',
3.          'field_2' => 'value2'
4.        );
```

Code expecting `$primary` to be true will probably get a "Cannot use string offset as an array" fatal error from PHP if a recursive find is used.

Below is an example of how afterfind can be used for date formating.

```
1.      function afterFind($results) {
2.          foreach ($results as $key => $val) {
3.              if (isset($val['Event']['begindate'])) {
4.                  $results[$key]['Event']['begindate'] = $this->dateFormatAfterFind($val['Event']['begindate']);
5.              }
6.          }
7.          return $results;
8.      }
9.      function dateFormatAfterFind($dateString) {
10.         return date('d-m-Y', strtotime($dateString));
11.     }
```

3.7.7.3 beforeValidate

`beforeValidate()`

Use this callback to modify model data before it is validated, or to modify validation rules if required. This function must also return *true*, otherwise the current save() execution will abort.

3.7.7.4 beforeSave

`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.

Below is an example of how beforeSave can be used for date conversion. The code in the example is used for an application with a begindate formatted like YYYY-MM-DD in the database and is displayed like DD-MM-YYYY in the application. Of course this can be changed very easily. Use the code below in the appropriate model.

```
1.      function beforeSave() {
2.          if (!empty($this->data['Event']['begindate']) && !empty($this->data['Event']['enddate'])) {
3.                  $this->data['Event']['begindate']              =              $this->dateFormatBeforeSave($this-
>data['Event']['begindate']);
4.                  $this->data['Event']['enddate'] = $this->dateFormatBeforeSave($this->data['Event']['enddate']);
5.          }
6.          return true;
7.      }
8.      function dateFormatBeforeSave($dateString) {
9.          return date('Y-m-d', strtotime($dateString)); // Direction is from
10.     }
```

Be sure that beforeSave() returns true, or your save is going to fail.

3.7.7.5 afterSave

`afterSave(boolean $created)`

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

The value of `$created` will be true if a new record was created (rather than an update).

3.7.7.6 beforeDelete

`beforeDelete(boolean $cascade)`

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.

The value of `$cascade` will be `true` if records that depend on this record will also be deleted.

Be sure that beforeDelete() returns true, or your delete is going to fail.

```
1.      // using app/models/ProductCategory.php
2.      // In the following example, do not let a product category be deleted if it still contains products.
3.      // A call of $this->Product->delete($id) from ProductsController.php has set $this->id .
4.      // Assuming 'ProductCategory hasMany Product', we can access $this->Product in the model.
5.      function beforeDelete()
6.      {
7.          $count = $this->Product->find("count", array(
8.              "conditions" => array("product_category_id" => $this->id)
9.          ));
10.         if ($count == 0) {
11.             return true;
12.         } else {
13.             return false;
14.         }
15.     }
```

3.7.7.7 afterDelete

`afterDelete()`

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

3.7.7.8 onError

`onError()`

Called if any problems occur.

3.7.8 Model Attributes

Model attributes allow you to set properties that can override the default model behavior.

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

3.7.8.1 useDbConfig

The `useDbConfig` property is a string that specifies the name of the database connection to use to bind your model class to the related database table. You can set it to any of the database connections defined within your database configuration file. The database configuration file is stored in `/app/config/database.php`.

The `useDbConfig` property is defaulted to the 'default' database connection.

Example usage:

```
1.      class Example extends AppModel {
2.          var $useDbConfig = 'alternate';
3.      }
```

3.7.8.2 useTable

The `useTable` property specifies the database table name. By default, the model uses the lowercase, plural form of the model's class name. Set this attribute to the name of an alternate table, or set it to `false` if you wish the model to use no database table.

Example usage:

```
1.      class Example extends AppModel {
2.          var $useTable = false; // This model does not use a database table
3.      }
```

Alternatively:

```
1.      class Example extends AppModel {
2.          var $useTable = 'exmp'; // This model uses a database table 'exmp'
3.      }
```

3.7.8.3 tablePrefix

The name of the table prefix used for the model. The table prefix is initially set in the database connection file at `/app/config/database.php`. The default is no prefix. You can override the default by setting the `tablePrefix` attribute in the model.

Example usage:

```
1.      class Example extends AppModel {
2.          var $tablePrefix = 'alternate_'; // will look for 'alternate_examples'
3.      }
```

3.7.8.4 primaryKey

Each table normally has a primary key, `id`. You may change which field name the model uses as its primary key. This is common when setting CakePHP to use an existing database table.

Example usage:

```
1.      class Example extends AppModel {
2.          var $primaryKey = 'example_id'; // example_id is the field name in the database
3.      }
```

3.7.8.5 displayField

The `displayField` attribute specifies which database field should be used as a label for the record. The label is used in scaffolding and in `find('list')` calls. The model will use `name` or `title`, by default.

For example, to use the `username` field:

```
1.      class User extends AppModel {
2.          var $displayField = 'username';
3.      }
```

Multiple field names cannot be combined into a single display field. For example, you cannot specify, `array('first_name', 'last_name')` as the display field. Instead create a virtual field with the Model attribute virtualFields

3.7.8.6 recursive

The recursive property defines how deep CakePHP should go to fetch associated model data via `find()`, `findAll()` and `read()` methods.

Imagine your application features Groups which belong to a domain and 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:

Depth	Description
-1	Cake fetches Group data only, no joins.
0	Cake fetches Group data and its domain
1	Cake fetches a Group, its domain and its associated Users
2	Cake fetches a Group, its domain, 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 slows your app unnecessarily. Also note that the default recursive level is 1.

If you want to combine $recursive with the `fields` functionality, you will have to add the columns containing the required foreign keys to the `fields` array manually. In the example above, this could mean adding `domain_id`.

3.7.8.7 order

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

```
1.        $order = "field"
2.        $order = "Model.field";
3.        $order = "Model.field asc";
4.        $order = "Model.field ASC";
5.        $order = "Model.field DESC";
6.        $order = array("Model.field" => "asc", "Model.field2" => "DESC");
```

3.7.8.8 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.

3.7.8.9 _schema

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

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

Example Usage:

```
1.        var $_schema = array(
2.            'first_name' => array(
3.                'type' => 'string',
4.                'length' => 30
5.            ),
6.            'last_name' => array(
7.                'type' => 'string',
```

```
8.                'length' => 30
9.            ),
10.           'email' => array(
11.               'type' => 'string',
12.               'length' => 30
13.           ),
14.           'message' => array('type' => 'text')
15.       );
```

3.7.8.10 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.

It is not necessary to call validate() before save() as save() will automatically validate your data before actually saving.

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

3.7.8.11 virtualFields

Array of virtual fields this model has. Virtual fields are aliased SQL expressions. Fields added to this property will be read as other fields in a model but will not be saveable.

Example usage for MySQL:

```
1.        var $virtualFields = array(
2.            'name' => "CONCAT(User.first_name, ' ', User.last_name)"
3.        );
```

In subsequent find operations, your User results would contain a `name` key with the result of the concatenation. It is not advisable to create virtual fields with the same names as columns on the database, this can cause SQL errors.

For more information on the `virtualFields` property, its proper usage, as well as limitations, see the section on virtual fields.

3.7.8.12 name

As you saw earlier in this chapter, the name attribute is a compatibility feature for PHP4 users and is set to the same value as the model name.

Example usage:

```
1.      class Example extends AppModel {
2.          var $name = 'Example';
3.      }
```

3.7.8.13 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.

3.7.9 Additional Methods and Properties

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 or define your own properties.

Any operation that handles the saving and fetching of data is best housed in your model classes. This concept is often referred to as the fat model.

```
1.      class Example extends AppModel {
2.          function getRecent() {
3.              $conditions = array(
4.                  'created BETWEEN (curdate() - interval 7 day) and (curdate() - interval 0 day))'
5.              );
6.              return $this->find('all', compact('conditions'));
7.          }
8.      }
```

This `getRecent()` method can now be used within the controller.

```
1.      $recent = $this->Example->getRecent();
```

3.7.9.1 Using virtualFields

Virtual fields are a new feature in the Model for CakePHP 1.3. Virtual fields allow you to create arbitrary SQL expressions and assign them as fields in a Model. These fields cannot be saved, but will be treated like other model fields for read operations. They will be indexed under the model's key alongside other model fields.

How to create virtual fields

Creating virtual fields is easy. In each model you can define a $virtualFields property that contains an array of `field => expressions`. An example of virtual field definitions would be:

```
1.      var $virtualFields = array(
2.          'name' => 'CONCAT(User.first_name, " ", User.last_name)'
3.      );
```

In subsequent find operations, your User results would contain a `name` key with the result of the concatenation. It is not advisable to create virtual fields with the same names as columns on the database, this can cause SQL errors.

Using virtual fields

Creating virtual fields is straightforward and easy, interacting with virtual fields can be done through a few different methods.

`Model::hasField()`

`Model::hasField()` has been updated so that it can will return true if the model has a `virtualField` with the correct name. By setting the second parameter of `hasField` to true, `virtualFields` will also be checked when checking if a model has a field. Using the example field above,

```
1.      $this->User->hasField('name'); // Will return false, as there is no concrete field called name
```

```
2.        $this->User->hasField('name', true); // Will return true as there is a virtual field called name
```

Model::isVirtualField()

This method can be used to check if a field/column is a virtual field or a concrete field. Will return true if the column is virtual.

```
1.        $this->User->isVirtualField('name'); //true
2.        $this->User->isVirtualField('first_name'); //false
```

Model::getVirtualField()

This method can be used to access the SQL expression that comprises a virtual field. If no argument is supplied it will return all virtual fields in a Model.

```
1.        $this->User->getVirtualField('name'); //returns 'CONCAT(User.first_name, ' ', User.last_name)'
```

Model::find() and virtual fields

As stated earlier `Model::find()` will treat virtual fields much like any other field in a model. The value of a virtual field will be placed under the model's key in the resultset. Unlike the behavior of calculated fields in 1.2

```
1.        $results = $this->User->find('first');
2.        // results contains the following
3.        array(
4.            'User' => array(
5.                'first_name' => 'Mark',
6.                'last_name' => 'Story',
7.                'name' => 'Mark Story',
8.                //more fields.
9.            )
10.       );
```

Pagination and virtual fields

Since virtual fields behave much like regular fields when doing find's, `Controller::paginate()` has been updated to allows sorting by virtual fields.

3.7.10 Virtual fields

Virtual fields are a new feature in the Model for CakePHP 1.3. Virtual fields allow you to create arbitrary SQL expressions and assign them as fields in a Model. These fields cannot be saved, but will be treated like other model fields for read operations. They will be indexed under the model's key alongside other model fields.

3.7.10.1 Creating virtual fields

Creating virtual fields is easy. In each model you can define a `$virtualFields` property that contains an array of field => expressions. An example of a virtual field definition using MySQL would be:

```
1.      var $virtualFields = array(
2.          'full_name' => 'CONCAT(User.first_name, " ", User.last_name)'
3.      );
```

And with PostgreSQL:

```
1.      var $virtualFields = array(
2.          'name' => 'User.first_name || \' \' || User.last_name'
3.      );
```

In subsequent find operations, your User results would contain a `name` key with the result of the concatenation. It is not advisable to create virtual fields with the same names as columns on the database, this can cause SQL errors.

It is not always useful to have **User.first_name** fully qualified. If you do not follow the convention (i.e. you have multiple relations to other tables) this would result in an error. In this case it may be better to just use **first_name || \'\' || last_name** without the Model Name.

3.7.10.2 Using virtual fields

Creating virtual fields is straightforward and easy, interacting with virtual fields can be done through a few different methods.

Model::hasField()

Model::hasField() has been updated so that it can return true if the model has a virtualField with the correct name. By setting the second parameter of hasField to true, virtualFields will also be checked when checking if a model has a field. Using the example field above,

```
1.  $this->User->hasField('name'); // Will return false, as there is no concrete field called name
2.  $this->User->hasField('name', true); // Will return true as there is a virtual field called name
```

Model::isVirtualField()

This method can be used to check if a field/column is a virtual field or a concrete field. Will return true if the column is virtual.

```
1.  $this->User->isVirtualField('name'); //true
2.  $this->User->isVirtualField('first_name'); //false
```

Model::getVirtualField()

This method can be used to access the SQL expression that comprises a virtual field. If no argument is supplied it will return all virtual fields in a Model.

```
1.  $this->User->getVirtualField('name'); //returns 'CONCAT(User.first_name, ' ', User.last_name)'
```

Model::find() and virtual fields

As stated earlier `Model::find()` will treat virtual fields much like any other field in a model. The value of a virtual field will be placed under the model's key in the resultset. Unlike the behavior of calculated fields in 1.2

```
1.  $results = $this->User->find('first');
2.  // results contains the following
3.  array(
4.      'User' => array(
5.          'first_name' => 'Mark',
6.          'last_name' => 'Story',
```

```
                'name' => 'Mark Story',
                //more fields.
            )
        );
```

Pagination and virtual fields

Since virtual fields behave much like regular fields when doing find's, `Controller::paginate()` has been updated to allows sorting by virtual fields.

3.7.10.3 Virtual fields and model aliases

When you are using virtualFields and models with aliases that are not the same as their name, you can run into problems as virtualFields do not update to reflect the bound alias. If you are using virtualFields in models that have more than one alias it is best to define the virtualFields in your model's constructor

```
        function __construct($id = false, $table = null, $ds = null) {
            parent::__construct($id, $table, $ds);
            $this->virtualFields['name'] = sprintf('CONCAT(%s.first_name, " ", %s.last_name)', $this->alias, $this->alias);
        }
```

This will allow your virtualFields to work for any alias you give a model.

3.7.10.4 Limitations of virtualFields

The implementation of `virtualFields` in 1.3 has a few limitations. First you cannot use `virtualFields` on associated models for conditions, order, or fields arrays. Doing so will generally result in an SQL error as the fields are not replaced by the ORM. This is because it difficult to estimate the depth at which an associated model might be found.

A common workaround for this implementation issue is to copy `virtualFields` from one model to another at runtime when you need to access them.

```
        $this->virtualFields['full_name'] = $this->Author->virtualFields['full_name'];
```

Alternatively, you can define `$virtualFields` in your model's constructor, using `$this->alias`, like so:

```
1.        public function __construct($id=false,$table=null,$ds=null){
2.          parent::__construct($id,$table,$ds);
3.          $this->virtualFields = array(
4.            'name'=>"CONCAT(`{$this->alias}`.`first_name`,' ',`{$this->alias}`.`last_name`)"
5.          );
6.        }
```

3.7.11 Transactions

To perform a transaction, a model's tables must be of a type that supports transactions.

All transaction methods must be performed on a model's DataSource object. To get a model's DataSource from within the model, use:

```
1.            $dataSource = $this->getDataSource();
```

You can then use the data source to start, commit, or roll back transactions.

```
1.            $dataSource->begin($this);
2.
3.            //Perform some tasks
4.            if(/*all's well*/) {
5.                $dataSource->commit($this);
6.            } else {
7.                $dataSource->rollback($this);
8.            }
```

Nested transactions are currently not supported. If a nested transaction is started, a commit will return false on the parent transaction.

3.8 Behaviors

Model behaviors are a way to organize some of the functionality defined in CakePHP models. They allow us to separate logic that may not be directly related to a model, but needs to be there. By providing a simple yet powerful way to extend models, behaviors allow us to attach functionality to models by defining a simple class variable. That's how behaviors allow models to get rid of all the extra weight that might not be part of the business contract they are modeling, or that is also needed in different models and can then be extrapolated.

As an example, consider a model that gives us access to a database table which stores structural information about a tree. Removing, adding, and migrating nodes in the tree is not as simple as deleting, inserting, and editing rows in the table. Many records may need to be updated as things move around. Rather than creating those tree-manipulation methods on a per model basis (for every model that needs that functionality), we could simply tell our model to use the TreeBehavior, or in more formal terms, we tell our model to behave as a Tree. This is known as attaching a behavior to a model. With just one line of code, our CakePHP model takes on a whole new set of methods that allow it to interact with the underlying structure.

CakePHP already includes behaviors for tree structures, translated content, access control list interaction, not to mention the community-contributed behaviors already available in the CakePHP Bakery (http://bakery.cakephp.org). In this section, we'll cover the basic usage pattern for adding behaviors to models, how to use CakePHP's built-in behaviors, and how to create our own.

In essence, Behaviors are Mixins with callbacks.

3.8.1 Using Behaviors

Behaviors are attached to models through the $actsAs model class variable:

```
1.  <?php
2.  class Category extends AppModel {
3.      var $name    = 'Category';
4.      var $actsAs = array('Tree');
5.  }
6.  ?>
```

This example shows how a Category model could be managed in a tree structure using the TreeBehavior. Once a behavior has been specified, use the methods added by the behavior as if they always existed as part of the original model:

```
1.        // Set ID
2.        $this->Category->id = 42;
3.        // Use behavior method, children():
4.        $kids = $this->Category->children();
```

Some behaviors may require or allow settings to be defined when the behavior is attached to the model. Here, we tell our TreeBehavior the names of the "left" and "right" fields in the underlying database table:

```
1.        <?php
2.        class Category extends AppModel {
3.            var $name    = 'Category';
4.            var $actsAs = array('Tree' => array(
5.                'left'  => 'left_node',
6.                'right' => 'right_node'
7.            ));
8.        }
9.        ?>
```

We can also attach several behaviors to a model. There's no reason why, for example, our Category model should only behave as a tree, it may also need internationalization support:

```
1.        <?php
2.        class Category extends AppModel {
3.            var $name    = 'Category';
4.            var $actsAs = array(
5.                'Tree' => array(
6.                  'left'  => 'left_node',
```

```
                'right' => 'right_node'
            ),
            'Translate'
        );
    }
    ?>
```

So far we have been adding behaviors to models using a model class variable. That means that our behaviors will be attached to our models throughout the model's lifetime. However, we may need to "detach" behaviors from our models at runtime. Let's say that on our previous Category model, which is acting as a Tree and a Translate model, we need for some reason to force it to stop acting as a Translate model:

```
    // Detach a behavior from our model:
    $this->Category->Behaviors->detach('Translate');
```

That will make our Category model stop behaving as a Translate model from thereon. We may need, instead, to just disable the Translate behavior from acting upon our normal model operations: our finds, our saves, etc. In fact, we are looking to disable the behavior from acting upon our CakePHP model callbacks. Instead of detaching the behavior, we then tell our model to stop informing of these callbacks to the Translate behavior:

```
    // Stop letting the behavior handle our model callbacks
    $this->Category->Behaviors->disable('Translate');
```

We may also need to find out if our behavior is handling those model callbacks, and if not we then restore its ability to react to them:

```
    // If our behavior is not handling model callbacks
    if (!$this->Category->Behaviors->enabled('Translate')) {
        // Tell it to start doing so
        $this->Category->Behaviors->enable('Translate');
    }
```

Just as we could completely detach a behavior from a model at runtime, we can also attach new behaviors. Say that our familiar Category model needs to start behaving as a Christmas model, but only on Christmas day:

```
1.      // If today is Dec 25
2.      if (date('m/d') == '12/25') {
3.          // Our model needs to behave as a Christmas model
4.          $this->Category->Behaviors->attach('Christmas');
5.      }
```

We can also use the attach method to override behavior settings:

```
1.      // We will change one setting from our already attached behavior
2.      $this->Category->Behaviors->attach('Tree', array('left' => 'new_left_node'));
```

There's also a method to obtain the list of behaviors a model has attached. If we pass the name of a behavior to the method, it will tell us if that behavior is attached to the model, otherwise it will give us the list of attached behaviors:

```
1.      // If the Translate behavior is not attached
2.      if (!$this->Category->Behaviors->attached('Translate')) {
3.          // Get the list of all behaviors the model has attached
4.          $behaviors = $this->Category->Behaviors->attached();
5.      }
```

3.8.2 Creating Behaviors

Behaviors that are attached to Models get their callbacks called automatically. The callbacks are similar to those found in Models: beforeFind, afterFind, beforeSave, afterSave, beforeDelete, afterDelete and onError - see Callback Methods.

Your behaviors should be placed in `app/models/behaviors`. It's often helpful to use a core behavior as a template when creating your own. Find them in `cake/libs/model/behaviors/`.

Every callback takes a reference to the model it is being called from as the first parameter.

Besides implementing the callbacks, you can add settings per behavior and/or model behavior attachment. Information about specifying settings can be found in the chapters about core behaviors and their configuration.

A quick example that illustrates how behavior settings can be passed from the model to the behavior:

```
1.      class Post extends AppModel {
2.          var $name = 'Post'
3.          var $actsAs = array(
4.              'YourBehavior' => array(
5.                  'option1_key' => 'option1_value'));
6.      }
```

As of 1.2.8004, CakePHP adds those settings once per model/alias only. To keep your behavior upgradable you should respect aliases (or models).

An upgrade-friendly function setup would look something like this:

```
1.      function setup(&$Model, $settings) {
2.          if (!isset($this->settings[$Model->alias])) {
3.              $this->settings[$Model->alias] = array(
4.                  'option1_key' => 'option1_default_value',
5.                  'option2_key' => 'option2_default_value',
6.                  'option3_key' => 'option3_default_value',
7.              );
8.          }
9.          $this->settings[$Model->alias] = array_merge(
10.             $this->settings[$Model->alias], (array)$settings);
```

```
11.         }
```

3.8.3 Creating behavior methods

Behavior methods are automatically available on any model acting as the behavior. For example if you had:

```
1.          class Duck extends AppModel {
2.              var $name = 'Duck';
3.              var $actsAs = array('Flying');
4.          }
```

You would be able to call FlyingBehavior methods as if they were methods on your Duck model. When creating behavior methods you automatically get passed a reference of the calling model as the first parameter. All other supplied parameters are shifted one place to the right. For example

```
1.          $this->Duck->fly('toronto', 'montreal');
```

Although this method takes two parameters, the method signature should look like:

```
1.          function fly(&$Model, $from, $to) {
2.              // Do some flying.
3.          }
```

Keep in mind that methods called in a `$this->doIt()` fashion

3.8.4 Behavior callbacks

Model Behaviors can define a number of callbacks that are triggered before/after the model callbacks of the same name. Behavior callbacks allow your behaviors to capture events in attached models and augment the parameters or splice in additional behavior.

The available callbacks are:

- `beforeValidate` is fired before a model's beforeValidate

- `beforeFind` is fired before a model's beforeFind
- `afterFind` is fired before a model's afterFind
- `beforeSave` is fired before a model's beforeSave
- `afterSave` is fired before a model's afterSave
- `beforeDelete` is fired after a model's beforeDelete
- `afterDelete` is fired before a model's afterDelete

3.8.5 Creating a behavior callback

Model behavior callbacks are defined as simple methods in your behavior class. Much like regular behavior methods, they receive a `$Model` parameter as the first argument. This parameter is the model that the behavior method was invoked on.

function beforeFind(&$model, $query)

If a behavior's beforeFind returns false it will abort the find(). Returning an array will augment the query parameters used for the find operation.

afterFind(&$model, $results, $primary)

You can use the afterFind to augment the results of a find. The return value will be passed on as the results to either the next behavior in the chain or the model's afterFind.

beforeDelete(&$model, $cascade = true)

You can return false from a behavior's beforeDelete to abort the delete. Return true to allow it continue.

afterDelete(&$model)

You can use afterDelete to perform clean up operations related to your behavior.

beforeSave(&$model)

You can return false from a behavior's beforeSave to abort the save. Return true to allow it continue.

afterSave(&$model, $created)

You can use afterSave to perform clean up operations related to your behavior. $created will be true when a record is created, and false when a record is updated.

beforeValidate(&$model)

You can use beforeValidate to modify a model's validate array or handle any other pre-validation logic. Returning false from a beforeValidate callback will abort the validation and cause it to fail.

3.9 DataSources

DataSources are the link between models and the source of data that models represent. In many cases, the data is retrieved from a relational database such as MySQL, PostgreSQL or MSSQL. CakePHP is distributed with several database-specific datasources (see the dbo_* class files in `cake/libs/model/datasources/dbo/`), a summary of which is listed here for your convenience:

- dbo_mssql.php
- dbo_mysql.php
- dbo_mysqli.php
- dbo_oracle.php
- dbo_postgres.php
- dbo_sqlite.php

Additional DataSources and those that were removed from the core in 1.3 can be found in the community-maintained CakePHP DataSources repository at github.

When specifying a database connection configuration in `app/config/database.php`, CakePHP transparently uses the corresponding database datasource for all model operations. So, even though you might not have known about datasources, you've been using them all along.

All of the above sources derive from a base `DboSource` class, which aggregates some logic that is common to most relational databases. If you decide to write a RDBMS datasource, working from one of these (e.g. `dbo_mysql.php` or `dbo_mssql.php` is your best bet.

Most people, however, are interested in writing datasources for external sources of data, such as remote REST APIs or even an LDAP server. So that's what we're going to look at now.

3.9.1 Basic API For DataSources

A datasource can, and *should* implement at least one of the following methods: `create`, `read`, `update` and/or `delete` (the actual method signatures & implementation details are not important for the moment, and will be described later). You need not implement more of the methods listed above than necessary - if you need a read-only datasource, there's no reason to implement `create`, `update`, and `delete`.

Methods that must be implemented

- `describe($model)`
- `listSources()`
- At least one of:
  - `create($model, $fields = array(), $values = array())`
  - `read($model, $queryData = array())`
  - `update($model, $fields = array(), $values = array())`
  - `delete($model, $id = null)`

It is also possible (and sometimes quite useful) to define the `$_schema` class attribute inside the datasource itself, instead of in the model.

And that's pretty much all there is to it. By coupling this datasource to a model, you are then able to use `Model::find()/save()` as you would normally, and the appropriate data and/or parameters used to call those methods will be passed on to the datasource itself, where you can decide to implement whichever features you need (e.g. Model::find options such as `'conditions'` parsing, `'limit'` or even your own custom parameters).

3.9.2 An Example

Here is a simple example of how to use Datasources and `HttpSocket` to implement a very basic Twitter source that allows querying the Twitter API as well as posting new status updates to a configured account.

This example will only work in PHP 5.2 and above, due to the use of `json_decode` for the parsing of JSON formatted data.

You would place the Twitter datasource in `app/models/datasources/twitter_source.php`:

```
1.  <?php
2.  /**
3.   * Twitter DataSource
4.   *
5.   * Used for reading and writing to Twitter, through models.
6.   *
7.   * PHP Version 5.x
8.   *
9.   * CakePHP(tm) : Rapid Development Framework (http://www.cakephp.org)
10.  * Copyright 2005-2009, Cake Software Foundation, Inc. (http://www.cakefoundation.org)
11.  *
12.  * Licensed under The MIT License
13.  * Redistributions of files must retain the above copyright notice.
14.  *
15.  * @filesource
16.  * @copyright     Copyright 2009, Cake Software Foundation, Inc. (http://www.cakefoundation.org)
17.  * @link          http://cakephp.org CakePHP(tm) Project
18.  * @license       http://www.opensource.org/licenses/mit-license.php The MIT License
19.  */
20. App::import('Core', 'HttpSocket');
21. class TwitterSource extends DataSource {
22.     protected $_schema = array(
23.         'tweets' => array(
24.             'id' => array(
25.                 'type' => 'integer',
26.                 'null' => true,
```

```
                    'key' => 'primary',
                    'length' => 11,
                ),
                'text' => array(
                    'type' => 'string',
                    'null' => true,
                    'key' => 'primary',
                    'length' => 140
                ),
                'status' => array(
                    'type' => 'string',
                    'null' => true,
                    'key' => 'primary',
                    'length' => 140
                ),
            )
        );
        public function __construct($config) {
            $auth = "{$config['login']}:{$config['password']}";
            $this->connection = new HttpSocket(
                "http://{$auth}@twitter.com/"
            );
            parent::__construct($config);
        }
        public function listSources() {
            return array('tweets');
        }
        public function read($model, $queryData = array()) {
            if (!isset($queryData['conditions']['username'])) {
                $queryData['conditions']['username'] = $this->config['login'];
            }
```

```
58.             $url = "/statuses/user_timeline/";
59.             $url .= "{$queryData['conditions']['username']}.json";
60.
61.             $response = json_decode($this->connection->get($url), true);
62.             $results = array();
63.
64.             foreach ($response as $record) {
65.                 $record = array('Tweet' => $record);
66.                 $record['User'] = $record['Tweet']['user'];
67.                 unset($record['Tweet']['user']);
68.                 $results[] = $record;
69.             }
70.             return $results;
71.         }
72.         public function create($model, $fields = array(), $values = array()) {
73.             $data = array_combine($fields, $values);
74.             $result = $this->connection->post('/statuses/update.json', $data);
75.             $result = json_decode($result, true);
76.             if (isset($result['id']) && is_numeric($result['id'])) {
77.                 $model->setInsertId($result['id']);
78.                 return true;
79.             }
80.             return false;
81.         }
82.         public function describe($model) {
83.             return $this->_schema['tweets'];
84.         }
85.     }
86.     ?>
```

Your model implementation could be as simple as:

```
1.      <?php
2.      class Tweet extends AppModel {
3.          public $useDbConfig = 'twitter';
4.      }
5.      ?>
```

If we had not defined our schema in the datasource itself, you would get an error message to that effect here.

And the configuration settings in your `app/config/database.php` would resemble something like this:

```
1.      <?php
2.          var $twitter = array(
3.              'datasource' => 'twitter',
4.              'login' => 'username',
5.              'password' => 'password',
6.          );
7.      ?>
```

Using the familiar model methods from a controller:

```
1.      <?php
2.      // Will use the username defined in the $twitter as shown above:
3.      $tweets = $this->Tweet->find('all');
4.      // Finds tweets by another username
5.      $conditions= array('username' => 'caketest');
6.      $otherTweets = $this->Tweet->find('all', compact('conditions'));
7.      ?>
```

Similarly, saving a new status update:

```
<?php
$this->Tweet->save(array('status' => 'This is an update'));
?>
```

3.9.3 Plugin DataSources and Datasource Drivers

Plugin Datasources

You can also package Datasources into plugins.

Simply place your datasource file into `plugins/[your_plugin]/models/datasources/[your_datasource]_source.php` and refer to it using the plugin notation:

```
var $twitter = array(
    'datasource' => 'Twitter.Twitter',
    'username' => 'test@example.com',
    'password' => 'hi_mom',
);
```

Plugin DBO Drivers

In addition, you can also add to the current selection of CakePHP's dbo drivers in plugin form.

Simply add your drivers to `plugins/[your_plugin]/models/datasources/dbo/[your_driver].php` and again use plugin notation:

```
var $twitter = array(
    'driver' => 'Twitter.Twitter',
    ...
);
```

Combining the Two

Finally, you're also able to bundle together your own DataSource and respective drivers so that they can share functionality. First create your main class you plan to extend:

```
plugins/[social_network]/models/datasources/[social_network]_source.php :
<?php
class SocialNetworkSource extends DataSource {
    // general functionality here
}
?>
```

And now create your drivers in a sub folder:

```
plugins/[social_network]/models/datasources/[social_network]/[twitter].php
<?php
class Twitter extends SocialNetworkSource {
    // Unique functionality here
}
?>
```

And finally setup your `database.php` settings accordingly:

```
var $twitter = array(
    'driver' => 'SocialNetwork.Twitter',
    'datasource' => 'SocialNetwork.SocialNetwork',
);
var $facebook = array(
    'driver' => 'SocialNetwork.Facebook',
    'datasource' => 'SocialNetwork.SocialNetwork',
);
```

Just like that, all your files are included **Automagically!** No need to place `App::import()` at the top of all your files

3.10 Views

-
- Comments (0)
- History

Views are the **V** in MVC. Views are responsible for generating the specific output required for the request. Often this is in the form of HTML, XML, or JSON, but streaming files and creating PDF's that users can download are also responsibilities of the View Layer.

3.10.1 View Templates

The view layer of CakePHP is how you speak to your users. Most of the time your views will be showing (X)HTML documents to browsers, but you might also need to serve AMF data to a Flash object, reply to a remote application via SOAP, or output a CSV file for a user.

CakePHP view files are written in plain PHP and have a default extension of .ctp (CakePHP Template). These files contain all the presentational logic needed to get the data it received from the controller in a format that is ready for the audience you're serving to.

View files are stored in /app/views/, in a folder named after the controller that uses the files, and named after the action it corresponds to. For example, the view file for the Products controller's "view()" action, would normally be found in /app/views/products/view.ctp.

The view layer in CakePHP can be made up of a number of different parts. Each part has different uses, and will be covered in this chapter:

- **layouts**: view files that contain presentational code that is found wrapping many interfaces in your application. Most views are rendered inside of a layout.
- **elements**: smaller, reusable bits of view code. Elements are usually rendered inside of views.
- **helpers**: these classes encapsulate view logic that is needed in many places in the view layer. Among other things, helpers in CakePHP can help you build forms, build AJAX functionality, paginate model data, or serve RSS feeds.

3.10.2 Layouts

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

Layout files should be placed in /app/views/layouts. CakePHP's default layout can be overridden by creating a new default layout at /app/views/layouts/default.ctp. Once a new default layout has been created, controller-rendered view code is placed inside of the default layout when the page is rendered.

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

```
1.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2.  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml">
4.  <head>
5.  <title><?php echo $title_for_layout?></title>
6.  <link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
7.  <!-- Include external files and scripts here (See HTML helper for more info.) -->
8.  <?php echo $scripts_for_layout ?>
9.  </head>
10. <body>
11. <!-- If you'd like some sort of menu to
12. show up on all of your views, include it here -->
13. <div id="header">
14.     <div id="menu">...</div>
15. </div>
16. <!-- Here's where I want my views to be displayed -->
17. <?php echo $content_for_layout ?>
18.
19. <!-- Add a footer to each displayed page -->
20. <div id="footer">...</div>
21. </body>
22. </html>
```

`$scripts_for_layout` contains any external files and scripts included with the built-in HTML helper. Useful for including javascript and CSS files from views.

When using `$html->css()` or `$javascript->link()` in view files, specify 'false' for the 'in-line' argument to place the html source in `$scripts_for_layout`. (See API for more details on usage).

`$content_for_layout` contains the view. This is where the view code will be placed.

`$title_for_layout` contains the page title.

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

```
1.  <?php
2.  class UsersController extends AppController {
3.      function viewActive() {
4.          $this->set('title_for_layout', 'View Active Users');
5.      }
6.  }
7.  ?>
```

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

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

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

```
1.  <?php
2.  class UsersController extends AppController {
```

```
3.              function viewActive() {
4.                  $this->set('title_for_layout', 'View Active Users');
5.                  $this->layout = 'default_small_ad';
6.              }
7.              function viewImage() {
8.                  $this->layout = 'image';
9.                  //output user image
10.             }
11.         }
12.         ?>
```

CakePHP features two core layouts (besides CakePHP's default layout) you can use in your own application: 'ajax' and 'flash'. The Ajax layout is handy for crafting Ajax responses - it's an empty layout (most ajax calls only require a bit of markup in return, rather than a fully-rendered interface). The flash layout is used for messages shown by the controllers flash() method.

Three other layouts xml, js, and rss exist in the core for a quick and easy way to serve up content that isn't text/html.

3.10.3 Elements

Many applications have small blocks of presentation code that need to be repeated from page to page, sometimes in different places in the layout. CakePHP can help you repeat parts of your website that need to be reused. These reusable parts are called Elements. Ads, help boxes, navigational controls, extra menus, login forms, and callouts are often implemented in CakePHP as elements. An element is basically a mini-view that can be included in other views, in layouts, and even within other elements. Elements can be used to make a view more readable, placing the rendering of repeating elements in its own file. They can also help you re-use content fragments in your application.

Elements live in the `/app/views/elements/` folder, and have the .ctp filename extension. They are output using the element method of the view.

```
1.          <?php echo $this->element('helpbox'); ?>
```

3.10.3.1 Passing Variables into an Element

You can pass data to an element through the element's second argument:

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

Inside the element file, all the passed variables are available as members of the parameter array (in the same way that `set()` in the controller works with view files). In the above example, the `/app/views/elements/helpbox.ctp` file can use the `$helptext` variable.

```
<?php
echo $helptext; //outputs "Oh, this text is very helpful."
?>
```

The `element()` function combines options for the element with the data for the element to pass. The two options are 'cache' and 'plugin'. An example:

```
<?php echo
$this->element('helpbox',
    array(
        "helptext" => "This is passed to the element as $helptext",
        "foobar" => "This is passed to the element as $foobar",
        "cache" => "+2 days", //sets the caching to +2 days.
        "plugin" => "" //to render an element from a plugin
    )
);
?>
```

To cache different versions of the same element in an application, provide a unique cache key value using the following format:

```
<?php
$this->element('helpbox',
    array(
```

```
4.                "cache" => array('time'=> "+7 days",'key'=>'unique value')
5.            )
6.        );
7.        ?>
```

You can take full advantage of elements by using `requestAction()`. The `requestAction()` function fetches view variables from a controller action and returns them as an array. This enables your elements to perform in true MVC style. Create a controller action that prepares the view variables for your elements, then call `requestAction()` inside the second parameter of `element()` to feed the element the view variables from your controller.

To do this, in your controller add something like the following for the Post example.

```
1.        <?php
2.        class PostsController extends AppController {
3.            ...
4.            function index() {
5.                $posts = $this->paginate();
6.                if (isset($this->params['requested'])) {
7.                    return $posts;
8.                } else {
9.                    $this->set('posts', $posts);
10.               }
11.           }
12.       }
13.       ?>
```

And then in the element we can access the paginated posts model. To get the latest five posts in an ordered list we would do something like the following:

```
1.        <h2>Latest Posts</h2>
2.        <?php $posts = $this->requestAction('posts/index/sort:created/direction:asc/limit:5'); ?>
3.        <?php foreach($posts as $post): ?>
```

```
4.      <ol>
5.          <li><?php echo $post['Post']['title']; ?></li>
6.      </ol>
7.      <?php endforeach; ?>
```

3.10.3.2 Caching Elements

You can take advantage of CakePHP view caching if you supply a cache parameter. If set to true, it will cache for 1 day. Otherwise, you can set alternative expiration times. See Caching for more information on setting expiration.

```
1.      <?php echo $this->element('helpbox', array('cache' => true)); ?>
```

If you render the same element more than once in a view and have caching enabled be sure to set the 'key' parameter to a different name each time. This will prevent each succesive call from overwriting the previous element() call's cached result. E.g.

```
1.      <?php
2.      echo $this->element('helpbox', array('cache' => array('key' => 'first_use', 'time' => '+1 day'), 'var' => $var));
3.      echo $this->element('helpbox', array('cache' => array('key' => 'second_use', 'time' => '+1 day'), 'var' => $differentVar));
4.      ?>
```

The above will ensure that both element results are cached separately.

3.10.3.3 Requesting Elements from a Plugin

If you are using a plugin and wish to use elements from within the plugin, just specify the plugin parameter. If the view is being rendered for a plugin controller/action, it will automatically point to the element for the plugin. If the element doesn't exist in the plugin, it will look in the main `APP` folder.

```
1.      <?php echo $this->element('helpbox', array('plugin' => 'pluginname')); ?>
```

3.10.4 View methods

View methods are accessible in all view, element and layout files. To call any view method use `$this->method()`

3.10.4.1 set()

```
set(string $var, mixed $value)
```

Views have a `set()` method that is analogous to the `set()` found in Controller objects. It allows you to add variables to the viewVars. Using set() from your view file will add the variables to the layout and elements that will be rendered later. See Controller::set() for more information on using set().

In your view file you can do

```
1.          $this->set('activeMenuButton', 'posts');
```

Then in your layout the `$activeMenuButton` variable will be available and contain the value 'posts'.

3.10.4.2 getVar()

```
getVar(string $var)
```

Gets the value of the viewVar with the name $var

3.10.4.3 getVars()

```
getVars()
```

Gets a list of all the available view variables in the current rendering scope. Returns an array of variable names.

3.10.4.4 error()

```
error(int $code, string $name, string $message)
```

Displays an error page to the user. Uses layouts/error.ctp to render the page.

```
1.          $this->error(404, 'Not found', 'This page was not found, sorry');
```

This will render an error page with the title and messages specified. Its important to note that script execution is not stopped by `View::error()` So you will have to stop code execution yourself if you want to halt the script.

3.10.4.5 element()

element(string $elementPath, array $data, bool $loadHelpers)

Renders an element or view partial. See the section on View Elements for more information and examples.

3.10.4.6 uuid

uuid(string $object, mixed $url)

Generates a unique non-random DOM ID for an object, based on the object type and url. This method is often used by helpers that need to generate unique DOM ID's for elements such as the AjaxHelper.

```
1.          $uuid = $this->uuid('form', array('controller' => 'posts', 'action' => 'index'));
2.          //$uuid contains 'form0425fe3bad'
```

3.10.4.7 addScript()

addScript(string $name, string $content)

Adds content to the internal scripts buffer. This buffer is made available in the layout as `$scripts_for_layout`. This method is helpful when creating helpers that need to add javascript or css directly to the layout. Keep in mind that scripts added from the layout, or elements in the layout will not be added to `$scripts_for_layout`. This method is most often used from inside helpers, like the Javascript and Html Helpers.

3.10.5 Themes

You can take advantage of themes, making it easy to switch the look and feel of your page quickly and easily.

To use themes, you need to tell your controller to use the ThemeView class instead of the default View class.

```
1.      class ExampleController extends AppController {
2.          var $view = 'Theme';
3.      }
```

To declare which theme to use by default, specify the theme name in your controller.

```
1.      class ExampleController extends AppController {
2.          var $view = 'Theme';
3.          var $theme = 'example';
4.      }
```

You can also set or change the theme name within an action or within the `beforeFilter` or `beforeRender` callback functions.

```
1.      $this->theme = 'another_example';
```

Theme view files need to be within the /app/views/themed/ folder. Within the themed folder, create a folder using the same name as your theme name. Beyond that, the folder structure within the /app/views/themed/example/ folder is exactly the same as /app/views/.

For example, the view file for an edit action of a Posts controller would reside at /app/views/themed/example/posts/edit.ctp. Layout files would reside in /app/views/themed/example/layouts/.

If a view file can't be found in the theme, CakePHP will try to locate the view file in the /app/views/ folder. This way, you can create master view files and simply override them on a case-by-case basis within your theme folder.

Theme assets

In previous versions themes needed to be split into their view and asset parts. New for 1.3 is a webroot directory as part of a theme. This webroot directory can contain any static assets that are included as part of your theme. Allowing the theme webroot to exist inside the views directory allows themes to be packaged far easier than before.

Linking to static assets is slightly different from 1.2. You can still use the existing `app/webroot/themed` and directly link to those static files. It should be noted that you will need to use the **full** path to link to assets in `app/webroot/themed`. If you want to keep your theme assets inside app/webroot it is recommended that you rename `app/webroot/themed` to `app/webroot/theme`. This will allow you to leverage the core helper path finding. As well as keep the performance benefits of not serving assets through PHP.

To use the new theme webroot create directories like `theme/<theme_name>/webroot<path_to_file>` in your theme. The Dispatcher will handle finding the correct theme assets in your view paths.

All of CakePHP's built-in helpers are aware of themes and will create the correct paths automatically. Like view files, if a file isn't in the theme folder, it'll default to the main webroot folder.

```
1.      //When in a theme with the name of 'purple_cupcake'
2.      $this->Html->css('main.css');
3.
4.      //creates a path like
5.      /theme/purple_cupcake/css/main.css
6.
7.      //and links to
8.      app/views/themed/purple_cupcake/webroot/css/main.css
```

3.10.5.1 Increasing performance of plugin and theme assets

Its a well known fact that serving assets through PHP is guaranteed to be slower than serving those assets without invoking PHP. And while the core team has taken steps to make plugin and theme asset serving as fast as possible, there may be situations where more performance is required. In these situations its recommended that you either symlink or copy out plugin/theme assets to directories in `app/webroot` with paths matching those used by cakephp.

- `app/plugins/debug_kit/webroot/js/my_file.js` becomes `app/webroot/debug_kit/js/my_file.js`
- `app/views/themed/navy/webroot/css/navy.css` becomes `app/webroot/theme/navy/css/navy.css`

3.10.6 Media Views

Media views allow you to send binary files to the user. For example, you may wish to have a directory of files outside of the webroot to prevent users from direct linking them. You can use the Media view to pull the file from a special folder within /app/, allowing you to perform authentication before delivering the file to the user.

To use the Media view, you need to tell your controller to use the MediaView class instead of the default View class. After that, just pass in additional parameters to specify where your file is located.

```
1.      class ExampleController extends AppController {
2.          function download () {
3.              $this->view = 'Media';
4.              $params = array(
5.                      'id' => 'example.zip',
6.                      'name' => 'example',
7.                      'download' => true,
8.                      'extension' => 'zip',
9.                      'path' => APP . 'files' . DS
10.             );
11.             $this->set($params);
12.         }
13.     }
```

Here's an example of rendering a file whose mime type is not included in the MediaView's `$mimeType` array.

```
1.      function download () {
2.          $this->view = 'Media';
3.          $params = array(
```

```
4.                  'id' => 'example.docx',
5.                  'name' => 'example',
6.                  'extension' => 'docx',
7.                  'mimeType'          =>          array('docx'          =>          'application/vnd.openxmlformats-officedocument.wordprocessingml.document'),
8.                  'path' => APP . 'files' . DS
9.          );
10.         $this->set($params);
11.     }
```

Parameters	Description
id	The ID is the file name as it resides on the file server including the file extension.
name	The name allows you to specify an alternate file name to be sent to the user. Specify the name without the file extension.
download	A boolean value indicating whether headers should be set to force download. Note that your controller's autoRender option should be set to false for this to work correctly.
extension	The file extension. This is matched against an internal list of acceptable mime types. If the mime type specified is not in the list (or sent in the mimeType parameter array), the file will not be downloaded.
path	The folder name, including the final directory separator. The path should be absolute, but can be relative to the APP/webroot folder.
mimeType	An array with additional mime types to be merged with MediaView internal list of acceptable mime types.
cache	A boolean or integer value - If set to true it will allow browsers to cache the file (defaults to false if not set); otherwise set it to the number of seconds in the future for when the cache should expire.

3.11 Helpers

Helpers are the component-like classes for the presentation layer of your application. They contain presentational logic that is shared between many views, elements, or layouts. This chapter will show you how to create your own helpers, and outline the basic tasks CakePHP's core helpers can help you accomplish. For more information on core helpers, check out Core Helpers.

3.11.1 Using Helpers

You use helpers in CakePHP by making a controller aware of them. Each controller has a $helpers property that lists the helpers to be made available in the view. To enable a helper in your view, add the name of the helper to the controller's $helpers array.

```
1.  <?php
2.  class BakeriesController extends AppController {
3.      var $helpers = array('Form', 'Html', 'Javascript', 'Time');
4.  }
5.  ?>
```

You can also add helpers from within an action, so they will only be available to that action and not the other actions in the controller. This saves processing power for the other actions that do not use the helper as well as help keep the controller better organized.

```
1.  <?php
2.  class BakeriesController extends AppController {
3.      function bake {
4.          $this->helpers[] = 'Time';
5.      }
6.      function mix {
7.          // The Time helper is not loaded here and thus not available
8.      }
9.  }
10. ?>
```

If you need to enable a helper for all controllers add the name of the helper to the $helpers array in */app/app_controller.php* (or create if not present). Remember to include the default Html and Form helpers.

```
<?php
class AppController extends Controller {
    var $helpers = array('Form', 'Html', 'Javascript', 'Time');
}
?>
```

You can pass options to helpers. These options can be used to set attribute values or modify behavior of a helper.

```
<?php
class AwesomeHelper extends AppHelper {
    function __construct($options = null) {
        parent::__construct($options);
        debug($options);
    }
}
?>
<?php
class AwesomeController extends AppController {
    var $helpers = array('Awesome' => array('option1' => 'value1'));
}
?>
```

3.11.2 Creating Helpers

If a core helper (or one showcased on Cakeforge or the Bakery) doesn't fit your needs, helpers are easy to create.

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

```
<?php
/* /app/views/helpers/link.php */
class LinkHelper extends AppHelper {
    function make($title, $url) {
        // Logic to create specially formatted link goes here...
    }
}
?>
```

3.11.2.1 Including other Helpers

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

```
<?php
/* /app/views/helpers/link.php (using other helpers) */
class LinkHelper extends AppHelper {
    var $helpers = array('Html');
    function make($title, $url) {
        // Use the HTML helper to output
        // formatted data:
        $link = $this->Html->link($title, $url, array('class' => 'edit'));
        return "<div class=\"editOuter\">$link</div>";
    }
}
?>
```

3.11.2.2 Callback method

Helpers feature a callback used by the parent controller class.

beforeRender()

The beforeRender method is called after the controller's beforeRender method but before the controller's renders views and layout.

3.11.2.3 Using your Helper

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

Once your controller has been made aware of this new class, you can use it in your views by accessing an object named after the helper:

```
1.      <!-- make a link using the new helper -->
2.      <?php echo $this->Link->make('Change this Recipe', '/recipes/edit/5'); ?>
```

This is the new syntax introduced in 1.3. You can also access helpers using the form `$link->make()`, however the newer format allows view variables and helpers to share names and not create collisions.

The Html, Form and Session (If sessions are enabled) helpers are always available.

3.11.3 Creating Functionality for All Helpers

All helpers extend a special class, AppHelper (just like models extend AppModel and controllers extend AppController). To create functionality that would be available to all helpers, create `/app/app_helper.php`.

```
1.      <?php
2.      class AppHelper extends Helper {
3.          function customMethod () {
4.          }
5.      }
6.      ?>
```

3.11.4 Core Helpers

CakePHP features a number of helpers that aid in view creation. They assist in creating well-formed markup (including forms), aid in formatting text, times and numbers, and can even speed up Ajax functionality. Here is a summary of the built-in helpers. For more information, check out Core Helpers.

CakePHP Helper	Description
Ajax	Used in tandem with the Prototype JavaScript library to create Ajax functionality in views. Contains shortcut methods for drag/drop, ajax forms & links, observers, and more.
Cache	Used by the core to cache view content.
Form	Creates HTML forms and form elements that self populate and handle validation problems.
Html	Convenience methods for crafting well-formed markup. Images, links, tables, header tags and more.
Js	Used to create Javascript compatible with various Javascript libraries. Replaces JavascriptHelper and AjaxHelper with a more flexible solution.
Javascript	Used to escape values for use in JavaScripts, write out data to JSON objects, and format code blocks.
Number	Number and currency formatting.
Paginator	Model data pagination and sorting.
Rss	Convenience methods for outputting RSS feed XML data.
Session	Access for reading session values in views.
Text	Smart linking, highlighting, word smart truncation.
Time	Proximity detection (is this next year?), nice string formatting(Today, 10:30 am) and time zone conversion.
Xml	Convenience methods for creating XML headers and elements.

3.12 Scaffolding

Application scaffolding is a technique that allows a developer to define and create a basic application that can create, retrieve, update and delete objects. Scaffolding in CakePHP also allows developers to define how objects are related to each other, and to create and break those links.

All that's needed to create a scaffold is a model and its controller. Once you set the $scaffold variable in the controller, you're up and running.

CakePHP's scaffolding is pretty cool. It allows you to get a basic CRUD application up and going in minutes. So cool that you'll want to use it in production apps. Now, we think its cool too, but please realize that scaffolding is... well... just scaffolding. It's a loose structure you throw up real quick during the beginning of a project in order to get started. It isn't meant to be completely flexible, it's meant as a temporary way to get up and going. If you find yourself really wanting to customize your logic and your views, its time to pull your scaffolding down in order to write some code. CakePHP's Bake console, covered in the next section, is a great next step: it generates all the code that would produce the same result as the most current scaffold.

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

To add scaffolding to your application, in the controller, add the $scaffold variable:

```
1.  <?php
2.  class CategoriesController extends AppController {
3.      var $scaffold;
4.  }
5.  ?>
```

Assuming you've created even the most basic Category model class file (in /app/models/category.php), you're ready to go. Visit http://example.com/categories to see your new scaffold.

Creating methods in controllers that are scaffolded can cause unwanted results. For example, if you create an index() method in a scaffolded controller, your index method will be rendered rather than the scaffolding functionality.

Scaffolding is knowledgeable about model associations, so if your Category model belongsTo a User, you'll see related User IDs in the Category listings. While scaffolding "knows" about model associations, you will not see any related records in the scaffold views until you manually add the association code to the model. For example, if Group hasMany User and User belongsTo Group, you have to manually add the following code in your User and Group models. Before you add the following code, the view displays an empty select input for Group in the New User form. After you add the following code, the view displays a select input populated with IDs or names from the Group table in the New User form.

```
1.      // In group.php
2.      var $hasMany = 'User';
3.      // In user.php
4.      var $belongsTo = 'Group';
```

If you'd rather see something besides an ID (like the user's first name), you can set the $displayField variable in the model. Let's set the $displayField variable in our User class so that users related to categories will be shown by first name rather than just an ID in scaffolding. This feature makes scaffolding more readable in many instances.

```
1.      <?php
2.      class User extends AppModel {
3.          var $name = 'User';
4.          var $displayField = 'first_name';
5.      }
6.      ?>
```

3.12.1 Creating a simple admin interface with scaffolding

If you have enabled admin routing in your app/config/core.php, with `Configure::write('Routing.prefixes', array('admin'));` you can use scaffolding to generate an admin interface.

Once you have enabled admin routing assign your admin prefix to the scaffolding variable.

```
1.        var $scaffold = 'admin';
```

You will now be able to access admin scaffolded actions:

```
1.        http://example.com/admin/controller/index
2.        http://example.com/admin/controller/view
3.        http://example.com/admin/controller/edit
4.        http://example.com/admin/controller/add
5.        http://example.com/admin/controller/delete
```

This is an easy way to create a simple backend interface quickly. Keep in mind that you cannot have both admin and non-admin methods scaffolded at the same time. As with normal scaffolding you can override individual methods and replace them with your own.

```
1.        function admin_view($id = null) {
2.          //custom code here
3.        }
```

Once you have replaced a scaffolded action you will need to create a view file for the action as well.

3.12.2 Customizing Scaffold Views

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

Customization is done by creating view templates:

```
Custom scaffolding views for a specific controller
(PostsController in this example) should be placed like so:

/app/views/posts/scaffold.index.ctp
/app/views/posts/scaffold.show.ctp
/app/views/posts/scaffold.edit.ctp
```

```
/app/views/posts/scaffold.new.ctp

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

/app/views/scaffolds/index.ctp
/app/views/scaffolds/show.ctp
/app/views/scaffolds/edit.ctp
/app/views/scaffolds/new.ctp
/app/views/scaffolds/add.ctp
```

3.13 The CakePHP Console

This section provides an introduction into CakePHP at the command-line. If you've ever needed access to your CakePHP MVC classes in a cron job or other command-line script, this section is for you.

PHP provides a powerful CLI client that makes interfacing with your file system and applications much smoother. The CakePHP console provides a framework for creating shell scripts. The Console uses a dispatcher-type setup to load a shell or task, and hand it its parameters.

A command-line (CLI) build of PHP must be available on the system if you plan to use the Console.

Before we get into specifics, let's make sure we can run the CakePHP Console. First, you'll need to bring up a system shell. The examples shown in this section will be in bash, but the CakePHP Console is Windows-compatible as well. Let's execute the Console program from bash. This example assumes that the user is currently logged into a bash prompt and is currently at the root of a CakePHP installation.

You can technically run the console using something like this:

```
$ cd /my/cake/app_folder
$ ../cake/console/cake
```

But the preferred usage is adding the console directory to your path so you can use the cake command anywhere:

```
$ cake
```

Running the Console with no arguments produces this help message:

```
Hello user,

Welcome to CakePHP v1.2 Console
---------------------------------------------------------------
Current Paths:
 -working: /path/to/cake/
 -root: /path/to/cake/
 -app: /path/to/cake/app/
 -core: /path/to/cake/

Changing Paths:
your working path should be the same as your application path
to change your path use the '-app' param.
Example: -app relative/path/to/myapp or -app /absolute/path/to/myapp

Available Shells:

 app/vendors/shells/:
        - none

 vendors/shells/:
        - none

 cake/console/libs/:
        acl
        api
        bake
        console
        extract

To run a command, type 'cake shell_name [args]'
To get help on a specific command, type 'cake shell_name help'
```

The first information printed relates to paths. This is especially helpful if you're running the Console from different parts of the filesystem.

Many users add the CakePHP Console to their system's path so it can be accessed easily. Printing out the working, root, app, and core paths allows you to see where the Console will be making changes. To change the app folder you wish to work with, you can supply its path as the first argument to the cake command. This next example shows how to specify an app folder, assuming you've already added the console folder to your PATH:

```
$ cake -app /path/to/app
```

The path supplied can be relative to the current working directory or supplied as an absolute path.

3.13.1 Creating Shells & Tasks

3.13.1.1 Creating Your Own Shells

Let's create a shell for use in the Console. For this example, we'll create a 'report' shell that prints out some model data. First, create report.php in /vendors/shells/.

```
1.  <?php
2.  class ReportShell extends Shell {
3.      function main() {}
4.  }
5.  ?>
```

From this point, we can run the shell, but it won't do much. Let's add some models to the shell so that we can create a report of some sort. This is done just as it is in the controller: by adding the names of models to the $uses variable.

```
1.  <?php
2.  class ReportShell extends Shell {
3.      var $uses = array('Order');
4.      function main() {
5.      }
6.  }
7.  ?>
```

Once we've added our model to the $uses array, we can use it in the main() method. In this example, our Order model should now be accessible as $this->Order in the main() method of our new shell.

Here's a simple example of the logic we might use in this shell:

```
1.      class ReportShell extends Shell {
2.          var $uses = array('Order');
3.          function main() {
4.              //Get orders shipped in the last    month
5.              $month_ago = date('Y-m-d H:i:s',    strtotime('-1 month'));
6.              $orders =    $this->Order->find("all",array('conditions'=>"Order.shipped >= '$month_ago'"));
7.              //Print out each order's information
8.              foreach($orders as $order) {
9.                  $this->out('Order date:    ' .    $order['Order']['created'] . "\n");
10.                 $this->out('Amount: $' .    number_format($order['Order']['amount'], 2) . "\n");
11.                 $this->out('----------------------------------------' .    "\n");
12.
13.                 $total += $order['Order']['amount'];
14.             }
15.             //Print out total for the selected orders
16.             $this->out("Total: $" .    number_format($total, 2) . "\n");
17.         }
18.     }
```

You would be able to run this report by executing this command (if the cake command is in your PATH):

```
$ cake report
```

where report is the name of the shell file in /vendor/shells/ without the .php extension. This should yield something like:

```
Hello user,
   Welcome to    CakePHP v1.2 Console
```

```
-----------------------------------------------------------------
App : app
Path:    /path/to/cake/app
-----------------------------------------------------------------
Order date:    2007-07-30 10:31:12
Amount:    $42.78
---------------------------------------
Order date:    2007-07-30 21:16:03
Amount:    $83.63
---------------------------------------
Order date:    2007-07-29 15:52:42
Amount:    $423.26
---------------------------------------
Order date:    2007-07-29 01:42:22
Amount:    $134.52
---------------------------------------
Order date:    2007-07-29 01:40:52
Amount:    $183.56
---------------------------------------
Total:    $867.75
```

3.13.1.2 Tasks

Tasks are small extensions to shells. They allow logic to be shared between shells, and are added to shells by using the special $tasks class variable. For example in the core bake shell, there are a number of tasks defined:

```php
<?php
class BakeShell extends Shell {
   var $tasks = array('Project', 'DbConfig', 'Model', 'View', 'Controller');
}
?>
```

Tasks are stored in /vendors/shells/tasks/ in files named after their classes. So if we were to create a new 'cool' task. Class CoolTask (which extends Shell) would be placed in /vendors/shells/tasks/cool.php. Class VeryCoolTask (which extends Shell) would be placed in /vendors/shells/tasks/very_cool.php.

Each task must at least implement an execute() method - shells will call this method to start the task logic.

```
1.  <?php
2.  class SoundTask extends Shell {
3.     var $uses = array('Model'); // same as controller var $uses
4.     function execute() {}
5.  }
6.  ?>
```

You can access tasks inside your shell classes and execute them there:

```
1.  <?php
2.  class SeaShell extends Shell { // found in /vendors/shells/sea.php
3.     var $tasks = array('Sound'); //found in /vendors/shells/tasks/sound.php
4.     function main() {
5.         $this->Sound->execute();
6.     }
7.  }
8.  ?>
```

You can also access tasks directly from the command line:

```
$ cake sea sound
```

In order to access tasks directly from the command line, the task **must** be included in the shell class' $tasks property. Therefore, be warned that a method called "sound" in the SeaShell class would override the ability to access the functionality in the Sound task specified in the $tasks array.

3.13.2 Running Shells as cronjobs

A common thing to do with a shell is making it run as a cronjob to clean up the database once in a while or send newsletters. However, when you have added the console path to the PATH variable via `~/.profile`, it will be unavailable to the cronjob.

The following BASH script will call your shell and append the needed paths to $PATH. Copy and save this to your vendors folder as 'cakeshell' and don't forget to make it executable. (`chmod +x cakeshell`)

```
#!/bin/bash
TERM=dumb
export TERM
cmd="cake"
while [ $# -ne 0 ]; do
        if [ "$1" = "-cli" ] || [ "$1" = "-console" ]; then
                PATH=$PATH:$2
                shift
        else
                cmd="${cmd} $1"
        fi
        shift
done
$cmd
```

You can call it like:

```
$ ./vendors/cakeshell myshell myparam -cli /usr/bin -console /cakes/1.2.x.x/cake/console
```

The `-cli` parameter takes a path which points to the php cli executable and the `-console` parameter takes a path which points to the CakePHP console.

As a cronjob this would look like:

```
# m h dom mon dow command
*/5 *   *   *   * /full/path/to/cakeshell myshell myparam -cli /usr/bin -console /cakes/1.2.x.x/cake/console -app
/full/path/to/app
```

A simple trick to debug a crontab is to set it up to dump it's output to a logfile. You can do this like:

```
# m h dom mon dow command
*/5 *    *    *    * /full/path/to/cakeshell myshell myparam -cli /usr/bin -console /cakes/1.2.x.x/cake/console -app
/full/path/to/app >> /path/to/log/file.log
```

3.14 Plugins

CakePHP allows you to set up a combination of controllers, models, and views and release them as a packaged application plugin that others can use in their CakePHP applications. Have a sweet user management module, simple blog, or web services module in one of your applications? Package it as a CakePHP plugin so you can pop it into other applications.

The main tie between a plugin and the application it has been installed into, is the application's configuration (database connection, etc.). Otherwise, it operates in its own little space, behaving much like it would if it were an application on its own.

3.14.1 Creating a Plugin

As a working example, let's create a new plugin that orders pizza for you. To start out, we'll need to place our plugin files inside the /app/plugins folder. The name of the parent folder for all the plugin files is important, and will be used in many places, so pick wisely. For this plugin, let's use the name '**pizza**'. This is how the setup will eventually look:

```
/app
    /plugins
        /pizza
            /controllers                <- plugin controllers go    here
            /models                     <- plugin models go    here
            /views                      <- plugin views go    here
            /pizza_app_controller.php   <- plugin's AppController
            /pizza_app_model.php        <- plugin's AppModel
```

If you want to be able to access your plugin with a URL, defining an AppController and AppModel for a plugin is required. These two special classes are named after the plugin, and extend the parent application's AppController and AppModel. Here's what they should look like for our pizza example:

```
1.      // /app/plugins/pizza/pizza_app_controller.php:
```

```
2.        <?php
3.        class PizzaAppController extends AppController {
4.             //...
5.        }
6.        ?>
1.        // /app/plugins/pizza/pizza_app_model.php:
2.        <?php
3.        class PizzaAppModel extends AppModel {
4.               //...
5.        }
6.        ?>
```

If you forgot to define these special classes, CakePHP will hand you "Missing Controller" errors until you've done so.

Please note that the process of creating plugins can be greatly simplified by using the Cake shell.

In order to bake a plugin please use the following command:

```
user@host$ cake bake plugin pizza
```

Now you can bake using the same conventions which apply to the rest of your app. For example - baking controllers:

```
user@host$ cake bake plugin pizza controller ingredients
```

Please refer to the chapter dedicated to bake if you have any problems with using the command line.

3.14.2 Plugin Controllers

Controllers for our pizza plugin will be stored in /app/plugins/pizza/controllers/. Since the main thing we'll be tracking is pizza orders, we'll need an OrdersController for this plugin.

While it isn't required, it is recommended that you name your plugin controllers something relatively unique in order to avoid namespace conflicts with parent applications. Its not a stretch to think that a parent application might have a UsersController, OrdersController, or ProductsController: so you might want to be creative with controller names, or prepend the name of the plugin to the classname (PizzaOrdersController, in this case).

So, we place our new PizzaOrdersController in /app/plugins/pizza/controllers and it looks like so:

```
1.      // /app/plugins/pizza/controllers/pizza_orders_controller.php
2.      class PizzaOrdersController extends PizzaAppController {
3.          var $name = 'PizzaOrders';
4.          var $uses = array('Pizza.PizzaOrder');
5.          function index() {
6.                //...
7.          }
8.      }
```

This controller extends the plugin's AppController (called PizzaAppController) rather than the parent application's AppController.

Also note how the name of the model is prefixed with the name of the plugin. This line of code is added for clarity but is not necessary for this example.

If you want to access what we've got going thus far, visit /pizza/pizza_orders. You should get a "Missing Model" error because we don't have a PizzaOrder model defined yet.

3.14.3 Plugin Models

Models for the plugin are stored in /app/plugins/pizza/models. We've already defined a PizzaOrdersController for this plugin, so let's create the model for that controller, called PizzaOrder. PizzaOrder is consistent with our previously defined naming scheme of pre-pending all of our plugin classes with Pizza.

```
1.      // /app/plugins/pizza/models/pizza_order.php:
2.      class PizzaOrder extends PizzaAppModel {
3.          var $name = 'PizzaOrder';
```

```
4.      }
5.      ?>
```

Visiting /pizza/pizzaOrders now (given you've got a table in your database called 'pizza_orders') should give us a "Missing View" error. Let's create that next.

If you need to reference a model within your plugin, you need to include the plugin name with the model name, separated with a dot.

For example:

```
1.      // /app/plugins/pizza/models/example_model.php:
2.      class ExampleModel extends PizzaAppModel {
3.          var $name = 'ExampleModel';
4.              var $hasMany = array('Pizza.PizzaOrder');
5.      }
6.      ?>
```

If you would prefer that the array keys for the association not have the plugin prefix on them, use the alternative syntax:

```
1.      // /app/plugins/pizza/models/example_model.php:
2.      class ExampleModel extends PizzaAppModel {
3.          var $name = 'ExampleModel';
4.              var $hasMany = array(
5.                      'PizzaOrder' => array(
6.                              'className' => 'Pizza.PizzaOrder'
7.                      )
8.              );
9.      }
10.     ?>
```

3.14.4 Plugin Views

Views behave exactly as they do in normal applications. Just place them in the right folder inside of the /app/plugins/[plugin]/views/ folder. For our pizza ordering plugin, we'll need a view for our PizzaOrdersController::index() action, so let's include that as well:

```
1.      // /app/plugins/pizza/views/pizza_orders/index.ctp:
2.      <h1>Order A Pizza</h1>
3.      <p>Nothing goes better with Cake than a good pizza!</p>
4.      <!-- An order form of some sort might go here....-->
```

For information on how to use elements from a plugin, look up Requesting Elements from a Plugin

Overriding plugin views from inside your application

You can override any plugin views from inside your app using special paths. If you have a plugin called 'Pizza' you can override the view files of the plugin with more application specific view logic by creating files using the following template "app/views/plugins/$plugin/$controller/$view.ctp". For the pizza controller you could make the following file:

```
1.      /app/views/plugins/pizza/pizza_orders/index.ctp
```

Creating this file, would allow you to override "/app/plugins/pizza/views/pizza_orders/index.ctp".

3.14.5 Components, Helpers and Behaviors

A plugin can have Components, Helpers and Behaviors just like a regular CakePHP application. You can even create plugins that consist only of Components, Helpers or Behaviors and can be a great way to build reusable components that can easily be dropped into any project.

Building these components is exactly the same as building it within a regular application, with no special naming convention. Referring to your components from within the plugin also does not require any special reference.

```
1.      // Component
```

```
class ExampleComponent extends Object {
}
// within your Plugin controllers:
var $components = array('Example');
```

To reference the Component from outside the plugin requires the plugin name to be referenced.

```
var $components = array('PluginName.Example');
var $components = array('Pizza.Example'); // references ExampleComponent in Pizza plugin.
```

The same technique applies to Helpers and Behaviors.

3.14.6 Plugin assets

New for 1.3 is an improved and simplified plugin webroot directory. In the past plugins could have a vendors directory containing `img`, `js`, and `css`. Each of these directories could only contain the type of file they shared a name with. In 1.3 both plugins and themes can have a `webroot` directory. This directory should contain any and all public accessible files for your plugin

```
app/plugins/debug_kit/webroot/
                                    css/
                                    js/
                                    img/
                                    flash/
                                    pdf/
```

And so on. You are no longer restricted to the three directories in the past, and you may put any type of file in any directory, just like a regular webroot. The only restriction is that `MediaView` needs to know the mime-type of that asset.

Linking to assets in plugins

The urls to plugin assets remains the same. In the past you used `/debug_kit/js/my_file.js` to link to `app/plugins/debug_kit/vendors/js/my_file.js`. It now links to `app/plugins/debug_kit/webroot/js/my_file.js`

It is important to note the **/your_plugin/** prefix before the img, js or css path. That makes the magic happen!

3.14.7 Plugin Tips

So, now that you've built everything, it should be ready to distribute (though we'd suggest you also distribute a few extras like a readme or SQL file).

Once a plugin has been installed in /app/plugins, you can access it at the URL /pluginname/controllername/action. In our pizza ordering plugin example, we'd access our PizzaOrdersController at /pizza/pizzaOrders.

Some final tips on working with plugins in your CakePHP applications:

- When you don't have a [Plugin]AppController and [Plugin]AppModel, you'll get missing Controller errors when trying to access a plugin controller.
- You can have a default controller with the name of your plugin. If you do that, you can access its index action via /[plugin]. Unlike 1.2 only the index action route comes built in. Other shortcuts that were accessible in 1.2 will need to have routes made for them. This was done to fix a number of workarounds inside CakePHP
- You can define your own layouts for plugins, inside app/plugins/[plugin]/views/layouts. Otherwise, plugins will use the layouts from the /app/views/layouts folder by default.
- You can do inter-plugin communication by using $this->requestAction('/plugin/controller/action'); in your controllers.
- If you use requestAction, make sure controller and model names are as unique as possible. Otherwise you might get PHP "redefined class ..." errors.

3.15 Global Constants and Functions

While most of your day-to-day work in CakePHP will be utilizing core classes and methods, CakePHP features a number of global convenience functions that may come in handy. Many of these functions are for use with CakePHP classes (loading model or component classes), but many others make working with arrays or strings a little easier.

We'll also cover some of the constants available in CakePHP applications. Using these constants will help make upgrades more smooth, but are also convenient ways to point to certain files or directories in your CakePHP application.

3.15.1 Global Functions

Here are CakePHP's globally available functions. Many of them are convenience wrappers for long-named PHP functions, but some of them (like `uses()`) can be used to include code or perform other useful functions. Chances are if you're constantly wanting a function to accomplish an oft-used task, it's here.

3.15.1.1 __

```
__(string $string_id, boolean $return = false)
```

This function handles localization in CakePHP applications. The `$string_id` identifies the ID for a translation, and the second parameter allows you to have the function automatically echo the string (the default behavior), or return it for further processing (pass a boolean true to enable this behavior).

> Check out the Localization & Internationalization section for more information.

3.15.1.2 a

```
a(mixed $one, $two, $three...)
```

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

```
1.      print_r(a('foo', 'bar'));
2.      // output:
3.      array(
4.          [0] => 'foo',
5.          [1] => 'bar'
6.      )
```

This has been Deprecated and will be removed in 2.0 version. Use **array()** instead.

3.15.1.3 aa

```
aa(string $one, $two, $three...)
```

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

```
print_r(aa('a','b'));
// output:
array(
    'a' => 'b'
)
```

This has been Deprecated and will be removed in 2.0 version.

3.15.1.4 am

```
am(array $one, $two, $three...)
```

Merges all the arrays passed as parameters and returns the merged array.

3.15.1.5 config

Can be used to load files from your application `config`-folder via include_once. Function checks for existance before include and returns boolean. Takes an optional number of arguments.

Example: `config('some_file', 'myconfig');`

3.15.1.6 convertSlash

```
convertSlash(string $string)
```

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

3.15.1.7 debug

```
debug(mixed $var, boolean $showHtml = false)
```

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

Also see Basic Debugging

3.15.1.8 e

```
e(mixed $data)
```

Convenience wrapper for `echo()`.

This has been Deprecated and will be removed in 2.0 version. Use **echo()** instead

3.15.1.9 env

```
env(string $key)
```

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

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

3.15.1.10 fileExistsInPath

```
fileExistsInPath(string $file)
```

Checks to make sure that the supplied file is within the current PHP include_path. Returns a boolean result.

3.15.1.11 h

```
h(string $text, string $charset = null)
```

Convenience wrapper for `htmlspecialchars()`.

3.15.1.12 ife

```
ife($condition, $ifNotEmpty, $ifEmpty)
```

Used for ternary-like operations. If the `$condition` is non-empty, `$ifNotEmpty` is returned, else `$ifEmpty` is returned.

This has been Deprecated and will be removed in 2.0 version.

3.15.1.13 low

```
low(string $string)
```

Convenience wrapper for `strtolower()`.

This has been Deprecated and will be removed in 2.0 version. Use **strtolower()** instead

3.15.1.14 pr

```
pr(mixed $var)
```

Convenience wrapper for `print_r()`, with the addition of wrapping <pre> tags around the output.

3.15.1.15 r

```
r(string $search, string $replace, string $subject)
```

Convenience wrapper for `str_replace()`.

This has been Deprecated and will be removed in 2.0 version. Use **str_replace()** instead

3.15.1.16 stripslashes_deep

```
stripslashes_deep(array $value)
```

Recursively strips slashes from the supplied `$value`. Returns the modified array.

3.15.1.17 up

```
up(string $string)
```

Convenience wrapper for `strtoupper()`.

This has been Deprecated and will be removed in 2.0 version. Use **strtoupper()** instead

3.15.1.18 uses

```
uses(string $lib1, $lib2, $lib3...)
```

Used to load CakePHP's core libraries (found in `cake/libs/`). Supply the name of the library's file name without the '`.php`' extension.

This has been Deprecated and will be removed in 2.0 version.

3.15.2 Core Definition Constants

constant	Absolute path to the application's...
APP	root directory.
APP_PATH	app directory.
CACHE	cache files directory.
CAKE	cake directory.
COMPONENTS	components directory.
CONFIGS	configuration files directory.
CONTROLLER_TESTS	controller tests directory.
CONTROLLERS	controllers directory.
CSS	CSS files directory.
DS	Short for PHP's DIRECTORY_SEPARATOR, which is / on Linux and \ on windows.
ELEMENTS	elements directory.

HELPER_TESTS	helper tests directory.
HELPERS	helpers directory.
IMAGES	images directory.
JS	JavaScript files directory (in the webroot).
LAYOUTS	layouts directory.
LIB_TESTS	CakePHP Library tests directory.
LIBS	CakePHP libs directory.
LOGS	logs directory (in app).
MODEL_TESTS	model tests directory.
MODELS	models directory.
SCRIPTS	Cake scripts directory.
TESTS	tests directory (parent for the models, controllers, etc. test directories)
TMP	tmp directory.
VENDORS	vendors directory.
VIEWS	views directory.
WWW_ROOT	full path to the webroot.

3.16 Vendor packages

Vendor file information goes here.

3.16.1 Vendor assets

Support for vendor assets have been removed for 1.3. It is recommended that you take any vendor assets you have and repackage them into plugins. See Plugin assets for more information.

4 Common Tasks With CakePHP

4.1 Data Validation

Data validation is an important part of any application, as it helps to make sure that the data in a Model conforms to the business rules of the application. For example, you might want to make sure that passwords are at least eight characters long, or ensure that usernames are unique. Defining validation rules makes form handling much, much easier.

There are many different aspects to the validation process. What we'll cover in this section is the model side of things. Essentially: what happens when you call the save() method of your model. For more information about how to handle the displaying of validation errors, check out the section covering FormHelper.

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

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $validate = array();
5.  }
6.  ?>
```

In the example above, the $validate array is added to the User Model, but the array contains no validation rules. Assuming that the users table has login, password, email and born fields, the example below shows some simple validation rules that apply to those fields:

```
1.  <?php
2.  class User extends AppModel {
3.      var $name = 'User';
4.      var $validate = array(
```

```
5.                'login' => 'alphaNumeric',
6.                'email' => 'email',
7.                'born' => 'date'
8.            );
9.        }
10.       ?>
```

This last example shows how validation rules can be added to model fields. For the login field, only letters and numbers will be accepted, the email should be valid, and born should be a valid date. Defining validation rules enables CakePHP's automagic showing of error messages in forms if the data submitted does not follow the defined rules.

CakePHP has many validation rules and using them can be quite easy. Some of the built-in rules allow you to verify the formatting of emails, URLs, and credit card numbers – but we'll cover these in detail later on.

Here is a more complex validation example that takes advantage of some of these built-in validation rules:

```
1.        <?php
2.        class User extends AppModel {
3.            var $name = 'User';
4.            var $validate = array(
5.                'login' => array(
6.                    'alphaNumeric' => array(
7.                        'rule' => 'alphaNumeric',
8.                        'required' => true,
9.                        'message' => 'Alphabets and numbers only'
10.                       ),
11.                   'between' => array(
12.                       'rule' => array('between', 5, 15),
13.                       'message' => 'Between 5 to 15 characters'
14.                   )
15.               ),
```

```
            'password' => array(
                'rule' => array('minLength', '8'),
                'message' => 'Mimimum 8 characters long'
            ),
            'email' => 'email',
            'born' => array(
                'rule' => 'date',
                'message' => 'Enter a valid date',
                'allowEmpty' => true
            )
        );
    }
    ?>
```

Two validation rules are defined for login: it should contain letters and numbers only, and its length should be between 5 and 15. The password field should be a minimum of 8 characters long. The email should be a valid email address, and born should be a valid date. Also, notice how you can define specific error messages that CakePHP will use when these validation rules fail.

As the example above shows, a single field can have multiple validation rules. And if the built-in rules do not match your criteria, you can always add your own validation rules as required.

Now that you've seen the big picture on how validation works, let's look at how these rules are defined in the model. There are three different ways that you can define validation rules: simple arrays, single rule per field, and multiple rules per field.

4.1.1 Simple Rules

As the name suggests, this is the simplest way to define a validation rule. The general syntax for defining rules this way is:

```
    var $validate = array('fieldName' => 'ruleName');
```

Where, 'fieldName' is the name of the field the rule is defined for, and 'ruleName' is a pre-defined rule name, such as 'alphaNumeric', 'email' or 'isUnique'.

For example, to ensure that the user is giving a well formatted email address, you could use this rule:

```
1.      var $validate = array('user_email' => 'email');
```

4.1.2 One Rule Per Field

This definition technique allows for better control of how the validation rules work. But before we discuss that, let's see the general usage pattern adding a rule for a single field:

```
1.      var $validate = array(
2.          'fieldName1' => array(
3.              'rule' => 'ruleName', // or: array('ruleName', 'param1', 'param2' ...)
4.              'required' => true,
5.              'allowEmpty' => false,
6.              'on' => 'create', // or: 'update'
7.              'message' => 'Your Error Message'
8.          )
9.      );
```

The 'rule' key is required. If you only set 'required' => true, the form validation will not function correctly. This is because 'required' is not actually a rule.

As you can see here, each field (only one field shown above) is associated with an array that contains five keys: 'rule', 'required', 'allowEmpty', 'on' and 'message'. Let's have a closer look at these keys.

4.1.2.1 rule

The 'rule' key defines the validation method and takes either a single value or an array. The specified 'rule' may be the name of a method in your model, a method of the core Validation class, or a regular expression. For more information on the rules available by default, see Core Validation Rules.

If the rule does not require any parameters, 'rule' can be a single value e.g.:

```
1.      var $validate = array(
2.          'login' => array(
3.              'rule' => 'alphaNumeric'
4.          )
5.      );
```

If the rule requires some parameters (like the max, min or range), 'rule' should be an array:

```
1.      var $validate = array(
2.          'password' => array(
3.              'rule' => array('minLength', 8)
4.          )
5.      );
```

Remember, the 'rule' key is required for array-based rule definitions.

4.1.2.2 required

This key should be assigned to a boolean value. If ‘required’ is true, the field must be present in the data array. For example, if the validation rule has been defined as follows:

```
1.      var $validate = array(
2.          'login' => array(
3.              'rule' => 'alphaNumeric',
4.              'required' => true
5.          )
6.      );
```

The data sent to the model’s save() method must contain data for the login field. If it doesn’t, validation will fail. The default value for this key is boolean false.

`required => true` does not mean the same as the validation rule `notEmpty()`. `required => true` indicates that the array *key* must be present - it does not mean it must have a value. Therefore validation will fail if the field is not present in the dataset, but may (depending on the rule) succeed if the value submitted is empty ('').

4.1.2.3 allowEmpty

If set to `false`, the field value must be **nonempty**, where "nonempty" is defined as `!empty($value) || is_numeric($value)`. The numeric check is so that CakePHP does the right thing when `$value` is zero.

The difference between `required` and `allowEmpty` can be confusing. `'required' => true` means that you cannot save the model without the *key* for this field being present in `$this->data` (the check is performed with `isset`); whereas, `'allowEmpty' => false` makes sure that the current field *value* is nonempty, as described above.

4.1.2.4 on

The 'on' key can be set to either one of the following values: 'update' or 'create'. This provides a mechanism that allows a certain rule to be applied either during the creation of a new record, or during update of a record.

If a rule has defined 'on' => 'create', the rule will only be enforced during the creation of a new record. Likewise, if it is defined as 'on' => 'update', it will only be enforced during the updating of a record.

The default value for 'on' is null. When 'on' is null, the rule will be enforced during both creation and update.

4.1.2.5 message

The 'message' key allows you to define a custom validation error message for the rule:

```
1.      var $validate = array(
2.          'password' => array(
3.              'rule' => array('minLength', 8),
4.              'message' => 'Password must be at least 8 characters long'
5.          )
6.      );
```

4.1.3 Multiple Rules per Field

The technique outlined above gives us much more flexibility than simple rules assignment, but there's an extra step we can take in order to gain more fine-grained control of data validation. The next technique we'll outline allows us to assign multiple validation rules per model field.

If you would like to assign multiple validation rules to a single field, this is basically how it should look:

```
1.
2.      var $validate = array(
3.          'fieldName' => array(
4.              'ruleName' => array(
5.                  'rule' => 'ruleName',
6.                  // extra keys like on, required, etc. go here...
7.              ),
8.              'ruleName2' => array(
9.                  'rule' => 'ruleName2',
10.                 // extra keys like on, required, etc. go here...
11.             )
12.         )
13.     );
```

As you can see, this is quite similar to what we did in the previous section. There, for each field we had only one array of validation parameters. In this case, each 'fieldName' consists of an array of rule indices. Each 'ruleName' contains a separate array of validation parameters.

This is better explained with a practical example:

```
1.      var $validate = array(
2.          'login' => array(
3.              'loginRule-1' => array(
4.                  'rule' => 'alphaNumeric',
5.                  'message' => 'Only alphabets and numbers allowed',
```

```
                    'last' => true
                ),
                'loginRule-2' => array(
                    'rule' => array('minLength', 8),
                    'message' => 'Minimum length of 8 characters'
                )
            )
    );
```

The above example defines two rules for the login field: loginRule-1 and loginRule-2. As you can see, each rule is identified with an arbitrary name.

By default CakePHP tries to validate a field using all the validation rules declared for it and returns the error message for the last failing rule. But if the key `last` is set to `true` for a rule and it fails, then the error message for that rule is returned and further rules are not validated. So if you prefer to show the error message for the first failing rule then set `'last' => true` for each rule.

When using multiple rules per field the 'required' and 'allowEmpty' keys need to be used only once in the first rule.

If you plan on using internationalized error messages, you may want to specify error messages in your view instead:

```
    echo $form->input('login', array(
        'label' => __('Login', true),
        'error' => array(
                'loginRule-1' => __('Only alphabets and numbers allowed', true),
                'loginRule-2' => __('Minimum length of 8 characters', true)
            )
        )
    );
```

The field is now fully internationalized, and you are able to remove the messages from the model. For more information on the __() function, see "Localization & Internationalization"

4.1.4 Core Validation Rules

The Validation class in CakePHP contains many validation rules that can make model data validation much easier. This class contains many oft-used validation techniques you won't need to write on your own. Below, you'll find a complete list of all the rules, along with usage examples.

4.1.4.1 alphaNumeric

The data for the field must only contain letters and numbers.

```
1.      var $validate = array(
2.          'login' => array(
3.              'rule' => 'alphaNumeric',
4.              'message' => 'Usernames must only contain letters and numbers.'
5.          )
6.      );
```

4.1.4.2 between

The length of the data for the field must fall within the specified numeric range. Both minimum and maximum values must be supplied. Uses <= not < .

```
1.      var $validate = array(
2.          'password' => array(
3.              'rule' => array('between', 5, 15),
4.              'message' => 'Passwords must be between 5 and 15 characters long.'
5.          )
6.      );
```

The length of data is "the number of bytes in the string representation of the data". Be careful that it may be larger than the number of characters when handling non-ASCII characters.

4.1.4.3 blank

This rule is used to make sure that the field is left blank or only white space characters are present in its value. White space characters include space, tab, carriage return, and newline.

```
1.      var $validate = array(
2.          'id' => array(
3.              'rule' => 'blank',
4.              'on' => 'create'
5.          )
6.      );
```

4.1.4.4 boolean

The data for the field must be a boolean value. Valid values are true or false, integers 0 or 1 or strings '0' or '1'.

```
1.      var $validate = array(
2.          'myCheckbox' => array(
3.              'rule' => array('boolean'),
4.              'message' => 'Incorrect value for myCheckbox'
5.          )
6.      );
```

4.1.4.5 cc

This rule is used to check whether the data is a valid credit card number. It takes three parameters: ‘type’, ‘deep’ and ‘regex’.

The ‘type’ key can be assigned to the values of ‘fast’, ‘all’ or any of the following:

- amex
- bankcard
- diners
- disc
- electron

- enroute
- jcb
- maestro
- mc
- solo
- switch
- visa
- voyager

If ‘type’ is set to ‘fast’, it validates the data against the major credit cards’ numbering formats. Setting ‘type’ to ‘all’ will check with all the credit card types. You can also set ‘type’ to an array of the types you wish to match.

The ‘deep’ key should be set to a boolean value. If it is set to true, the validation will check the Luhn algorithm of the credit card (http://en.wikipedia.org/wiki/Luhn_algorithm). It defaults to false.

The ‘regex’ key allows you to supply your own regular expression that will be used to validate the credit card number.

```
1.      var $validate = array(
2.          'ccnumber' => array(
3.              'rule' => array('cc', array('visa', 'maestro'), false, null),
4.              'message' => 'The credit card number you supplied was invalid.'
5.          )
6.      );
```

4.1.4.6 comparison

Comparison is used to compare numeric values. It supports “is greater”, “is less”, “greater or equal”, “less or equal”, “equal to”, and “not equal”. Some examples are shown below:

```
1.      var $validate = array(
2.          'age' => array(
```

```
            'rule' => array('comparison', '>=', 18),
            'message' => 'Must be at least 18 years old to qualify.'
        )
    );
    var $validate = array(
        'age' => array(
            'rule' => array('comparison', 'greater or equal', 18),
            'message' => 'Must be at least 18 years old to qualify.'
        )
    );
```

4.1.4.7 date

This rule ensures that data is submitted in valid date formats. A single parameter (which can be an array) can be passed that will be used to check the format of the supplied date. The value of the parameter can be one of the following:

- 'dmy' e.g. 27-12-2006 or 27-12-06 (separators can be a space, period, dash, forward slash)
- 'mdy' e.g. 12-27-2006 or 12-27-06 (separators can be a space, period, dash, forward slash)
- 'ymd' e.g. 2006-12-27 or 06-12-27 (separators can be a space, period, dash, forward slash)
- 'dMy' e.g. 27 December 2006 or 27 Dec 2006
- 'Mdy' e.g. December 27, 2006 or Dec 27, 2006 (comma is optional)
- 'My' e.g. (December 2006 or Dec 2006)
- 'my' e.g. 12/2006 or 12/06 (separators can be a space, period, dash, forward slash)

If no keys are supplied, the default key that will be used is 'ymd'.

```
    var $validate = array(
        'born' => array(
            'rule' => array('date','ymd'),
            'message' => 'Enter a valid date in YY-MM-DD format.',
            'allowEmpty' => true
```

```
            )
        );
```

While many data stores require a certain date format, you might consider doing the heavy lifting by accepting a wide-array of date formats and trying to convert them, rather than forcing users to supply a given format. The more work you can do for your users, the better.

4.1.4.8 decimal

This rule ensures that the data is a valid decimal number. A parameter can be passed to specify the number of digits required after the decimal point. If no parameter is passed, the data will be validated as a scientific float, which will cause validation to fail if no digits are found after the decimal point.

```
        var $validate = array(
            'price' => array(
                'rule' => array('decimal', 2)
            )
        );
```

4.1.4.9 email

This checks whether the data is a valid email address. Passing a boolean true as the second parameter for this rule will also attempt to verify that the host for the address is valid.

```
        var $validate = array('email' => array('rule' => 'email'));

        var $validate = array(
            'email' => array(
                'rule' => array('email', true),
                'message' => 'Please supply a valid email address.'
            )
        );
```

4.1.4.10 equalTo

This rule will ensure that the value is equal to, and of the same type as the given value.

```
1.      var $validate = array(
2.          'food' => array(
3.              'rule' => array('equalTo', 'cake'),
4.              'message' => 'This value must be the string cake'
5.          )
6.      );
```

4.1.4.11 extension

This rule checks for valid file extensions like .jpg or .png. Allow multiple extensions by passing them in array form.

```
1.      var $validate = array(
2.          'image' => array(
3.              'rule' => array('extension', array('gif', 'jpeg', 'png', 'jpg')),
4.              'message' => 'Please supply a valid image.'
5.          )
6.      );
```

4.1.4.12 file

This rule ensures that the value is a valid file name. This validation rule is currently non-functional.

4.1.4.13 ip

This rule will ensure that a valid IPv4 or IPv6 address has been submitted. Accepts as option 'both' (default), 'IPv4' or 'IPv6'.

```
1.      var $validate = array(
2.          'clientip' => array(
3.              'rule' => array('ip', 'IPv4'), // or 'IPv6' or 'both' (default)
```

```
4.                'message' => 'Please supply a valid IP address.'
5.            )
6.        );
```

4.1.4.14 isUnique

The data for the field must be unique, it cannot be used by any other rows.

```
1.        var $validate = array(
2.            'login' => array(
3.                'rule' => 'isUnique',
4.                'message' => 'This username has already been taken.'
5.            )
6.        );
```

4.1.4.15 minLength

This rule ensures that the data meets a minimum length requirement.

```
1.        var $validate = array(
2.            'login' => array(
3.                'rule' => array('minLength', 8),
4.                'message' => 'Usernames must be at least 8 characters long.'
5.            )
6.        );
```

The length here is "the number of bytes in the string representation of the data". Be careful that it may be larger than the number of characters when handling non-ASCII characters.

4.1.4.16 maxLength

This rule ensures that the data stays within a maximum length requirement.

```
1.        var $validate = array(
2.            'login' => array(
3.                'rule' => array('maxLength', 15),
4.                'message' => 'Usernames must be no larger than 15 characters long.'
5.            )
6.        );
```

The length here is "the number of bytes in the string representation of the data". Be careful that it may be larger than the number of characters when handling non-ASCII characters.

4.1.4.17 money

This rule will ensure that the value is in a valid monetary amount.

Second parameter defines where symbol is located (left/right).

```
1.        var $validate = array(
2.            'salary' => array(
3.                'rule' => array('money', 'left'),
4.                'message' => 'Please supply a valid monetary amount.'
5.            )
6.        );
```

4.1.4.18 multiple

Use this for validating a multiple select input. It supports parameters "in", "max" and "min".

```
1.        var $validate = array(
2.            'multiple' => array(
3.                'rule' => array('multiple', array('in' => array('do', 'ray', 'me', 'fa', 'so', 'la', 'ti'), 'min' =>
   1, 'max' => 3)),
```

```
            'message' => 'Please select one, two or three options'
        )
    );
```

4.1.4.19 inList

This rule will ensure that the value is in a given set. It needs an array of values. The field is valid if the field's value matches one of the values in the given array.

Example:

```
        var $validate = array(
          'function' => array(
              'allowedChoice' => array(
                  'rule' => array('inList', array('Foo', 'Bar')),
                  'message' => 'Enter either Foo or Bar.'
              )
          )
        );
```

4.1.4.20 numeric

Checks if the data passed is a valid number.

```
    var $validate = array(
        'cars' => array(
            'rule' => 'numeric',
            'message' => 'Please supply the number of cars.'
        )
    );
```

4.1.4.21 notEmpty

The basic rule to ensure that a field is not empty.

```
1.      var $validate = array(
2.          'title' => array(
3.              'rule' => 'notEmpty',
4.              'message' => 'This field cannot be left blank'
5.          )
6.      );
```

Do not use this for a multiple select input as it will cause an error. Instead, use "multiple".

4.1.4.22 phone

Phone validates US phone numbers. If you want to validate non-US phone numbers, you can provide a regular expression as the second parameter to cover additional number formats.

```
1.      var $validate = array(
2.          'phone' => array(
3.              'rule' => array('phone', null, 'us')
4.          )
5.      );
```

4.1.4.23 postal

Postal is used to validate ZIP codes from the U.S. (us), Canada (ca), U.K (uk), Italy (it), Germany (de) and Belgium (be). For other ZIP code formats, you may provide a regular expression as the second parameter.

```
1.      var $validate = array(
2.          'zipcode' => array(
3.              'rule' => array('postal', null, 'us')
4.          )
5.      );
```

4.1.4.24 range

This rule ensures that the value is in a given range. If no range is supplied, the rule will check to ensure the value is a legal finite on the current platform.

```
1.      var $validate = array(
2.          'number' => array(
3.              'rule' => array('range', -1, 11),
4.              'message' => 'Please enter a number between 0 and 10'
5.          )
6.      );
```

The above example will accept any value which is larger than 0 (e.g., 0.01) and less than 10 (e.g., 9.99). Note: The range lower/upper are not inclusive!!!

4.1.4.25 ssn

Ssn validates social security numbers from the U.S. (us), Denmark (dk), and the Netherlands (nl). For other social security number formats, you may provide a regular expression.

```
1.      var $validate = array(
2.          'ssn' => array(
3.              'rule' => array('ssn', null, 'us')
4.          )
5.      );
```

4.1.4.26 url

This rule checks for valid URL formats. Supports http(s), ftp(s), file, news, and gopher protocols.

```
1.      var $validate = array(
2.          'website' => array(
3.              'rule' => 'url'
4.          )
```

```
5.        );
```

To ensure that a protocol is in the url, strict mode can be enabled like so.

```
1.        var $validate = array(
2.            'website' => array(
3.                'rule' => array('url', true)
4.            )
5.        );
```

4.1.5 Custom Validation Rules

If you haven't found what you need thus far, you can always create your own validation rules. There are two ways you can do this: by defining custom regular expressions, or by creating custom validation methods.

4.1.5.1 Custom Regular Expression Validation

If the validation technique you need to use can be completed by using regular expression matching, you can define a custom expression as a field validation rule.

```
1.        var $validate = array(
2.            'login' => array(
3.                'rule' => '/^[a-z0-9]{3,}$/i',
4.                'message' => 'Only letters and integers, min 3 characters'
5.            )
6.        );
```

The example above checks if the login contains only letters and integers, with a minimum of three characters.

The regular expression in the `rule` must be delimited by slashes. The optional trailing 'i' after the last slash means the reg-exp is case *i*nsensitive.

4.1.5.2 Adding your own Validation Methods

Sometimes checking data with regular expression patterns is not enough. For example, if you want to ensure that a promotional code can only be used 25 times, you need to add your own validation function, as shown below:

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

    var $validate = array(
        'promotion_code' => array(
            'rule' => array('limitDuplicates', 25),
            'message' => 'This code has been used too many times.'
        )
    );

    function limitDuplicates($check, $limit){
        //$check will have value: array('promomotion_code' => 'some-value')
        //$limit will have value: 25
        $existing_promo_count = $this->find( 'count', array('conditions' => $check, 'recursive' => -1) );
        return $existing_promo_count < $limit;
    }
}
?>
```

The current field to be validated is passed into the function as first parameter as an associated array with field name as key and posted data as value.

If you want to pass extra parameters to your validation function, add elements onto the 'rule' array, and handle them as extra params (after the main `$check` param) in your function.

Your validation function can be in the model (as in the example above), or in a behavior that the model implements. This includes mapped methods.

Model/behavior methods are checked first, before looking for a method on the `Validation` class. This means that you can override existing validation methods (such as `alphaNumeric()`) at an application level (by adding the method to `AppModel`), or at model level.

When writing a validation rule which can be used by multiple fields, take care to extract the field value from the $check array. The $check array is passed with the form field name as its key and the field value as its value. The full record being validated is stored in $this->data member variable.

```
1.  <?php
2.  class Post extends AppModel {
3.    var $name = 'Post';
4.
5.    var $validate = array(
6.      'slug' => array(
7.        'rule' => 'alphaNumericDashUnderscore',
8.        'message' => 'Slug can only be letters, numbers, dash and underscore'
9.        )
10.     );
11.
12.     function alphaNumericDashUnderscore($check) {
13.       // $data array is passed using the form field name as the key
14.       // have to extract the value to make the function generic
15.       $value = array_values($check);
16.       $value = $value[0];
17.
18.       return preg_match('|^[0-9a-zA-Z_-]*$|', $value);
19.     }
20. }
21. ?>
```

4.1.6 Validating Data from the Controller

While normally you would just use the save method of the model, there may be times where you wish to validate the data without saving it. For example, you may wish to display some additional information to the user before actually saving the data to the database. Validating data requires a slightly different process than just saving the data.

First, set the data to the model:

```
1.        $this->ModelName->set( $this->data );
```

Then, to check if the data validates, use the validates method of the model, which will return true if it validates and false if it doesn't:

```
1.        if ($this->ModelName->validates()) {
2.            // it validated logic
3.        } else {
4.            // didn't validate logic
5.        }
```

It may be desirable to validate your model only using a subset of the validations specified in your model. For example say you had a User model with fields for first_name, last_name, email and password. In this instance when creating or editing a user you would want to validate all 4 field rules. Yet when a user logs in you would validate just email and password rules. To do this you can pass an options array specifying the fields to validate. e.g.

```
1.        if ($this->User->validates(array('fieldList' => array('email', 'password')))) {
2.            // valid
3.        } else {
4.            // invalid
5.        }
```

The validates method invokes the invalidFields method which populates the validationErrors property of the model. The invalidFields method also returns that data as the result.

```
1.        $errors = $this->ModelName->invalidFields(); // contains validationErrors array
```

It is important to note that the data must be set to the model before the data can be validated. This is different from the save method which allows the data to be passed in as a parameter. Also, keep in mind that it is not required to call validates prior to calling save as save will automatically validate the data before actually saving.

To validate multiple models, the following approach should be used:

```
1.        if ($this->ModelName->saveAll($this->data, array('validate' => 'only'))) {
2.          // validates
3.        } else {
4.          // does not validate
5.        }
```

If you have validated data before save, you can turn off validation to avoid second check.

```
1.        if ($this->ModelName->saveAll($this->data, array('validate' => false))) {
2.            // saving wihout validation
3.        }
```

4.2 Data Sanitization

The CakePHP Sanitize class can be used to rid user-submitted data of malicious data and other unwanted information. Sanitize is a core library, so it can be used anywhere inside of your code, but is probably best used in controllers or models.

CakePHP already protects you against SQL Injection **if** you use CakePHP's ORM methods (such as find() and save()) and proper array notation (ie. array('field' => $value)) instead of raw SQL. For sanitization against XSS its generally better to save raw HTML in database without modification and sanitize at the time of output/display.

All you need to do is include the Sanitize core library (e.g. before the controller class definition):

```
App::import('Sanitize');
class MyController extends AppController {
    ...
    ...
}
```

Once you've done that, you can make calls to Sanitize statically.

4.2.1 paranoid

paranoid(string $string, array $allowedChars);

This function strips anything out of the target $string that is not a plain-jane alphanumeric character. The function can be made to overlook certain characters by passing them in $allowedChars array.

```
$badString = ";:<script><html><   // >@@#";
echo Sanitize::paranoid($badString);
// output: scripthtml
echo Sanitize::paranoid($badString, array(' ', '@'));
// output: scripthtml    @@
```

4.2.2 html

html(string $string, array $options = array())

This method prepares user-submitted data for display inside HTML. This is especially useful if you don't want users to be able to break your layouts or insert images or scripts inside of your HTML pages. If the $remove option is set to true, HTML content detected is removed rather than rendered as HTML entities.

```
$badString = '<font size="99" color="#FF0000">HEY</font><script>...</script>';
echo Sanitize::html($badString);
```

```
3.      //                      output:                         &lt;font                      size=&quot;99&quot;
color=&quot;#FF0000&quot;&gt;HEY&lt;/font&gt;&lt;script&gt;...&lt;/script&gt;
4.      echo Sanitize::html($badString, array('remove' => true));
5.      // output: HEY...
```

4.2.3 escape

escape(string $string, string $connection)

Used to escape SQL statements by adding slashes, depending on the system's current magic_quotes_gpc setting. $connection is the name of the database to quote the string for, as named in your app/config/database.php file.

4.2.4 clean

```
Sanitize::clean(mixed $data, mixed $options)
```

This function is an industrial-strength, multi-purpose cleaner, meant to be used on entire arrays (like $this->data, for example). The function takes an array (or string) and returns the clean version. The following cleaning operations are performed on each element in the array (recursively):

- Odd spaces (including 0xCA) are replaced with regular spaces.
- Double-checking special chars and removal of carriage returns for increased SQL security.
- Adding of slashes for SQL (just calls the sql function outlined above).
- Swapping of user-inputted backslashes with trusted backslashes.

The $options argument can either be a string or an array. When a string is provided it's the database connection name. If an array is provided it will be merged with the following options:

- connection
- odd_spaces
- encode
- dollar
- carriage

- unicode
- escape
- backslash
- remove_html (must be used in conjunction with the encode parameter)

Usage of clean() with options looks something like the following:

```
1.        $this->data = Sanitize::clean($this->data, array('encode' => false));
```

4.3 Error Handling

In the event of an unrecoverable error in your application, it is common to stop processing and show an error page to the user. To save you from having to code error handling for this in each of your controllers and components, you can use the provided method:

```
$this->cakeError(string $errorType [, array $parameters]);
```

Calling this method will show an error page to the user and halt any further processing in your application.

`parameters` must be an array of strings. If the array contains objects (including Exception objects), they will be cast into strings.

CakePHP pre-defines a set of error-types, but at the time of writing, most are only really useful by the framework itself. One that is more useful to the application developer is the good old 404 error. This can be called with no parameters as follows:

```
1.        $this->cakeError('error404');
```

Or alternatively, you can cause the page to report the error was at a specific URL by passing the `url` parameter:

```
1.        $this->cakeError('error404', array('url' => 'some/other.url'));
```

This all starts being a lot more useful by extending the error handler to use your own error-types. Application error handlers are largely like controller actions; You typically will set() any passed parameters to be available to the view and then render a view file from your `app/views/errors` directory.

Create a file `app/app_error.php` with the following definition.

```
<?php
class AppError extends ErrorHandler {
}
?>
```

Handlers for new error-types can be implemented by adding methods to this class. Simply create a new method with the name you want to use as your error-type.

Let's say we have an application that writes a number of files to disk and that it is appropriate to report write errors to the user. We don't want to add code for this all over the different parts of our application, so this is a great case for using a new error type.

Add a new method to your `AppError` class. We'll take one parameter called `file` that will be the path to the file we failed to write.

```
function cannotWriteFile($params) {
  $this->controller->set('file', $params['file']);
  $this->_outputMessage('cannot_write_file');
}
```

Create the view in `app/views/errors/cannot_write_file.ctp`

```
<h2>Unable to write file</h2>
<p>Could not write file <?php echo $file ?> to the disk.</p>
```

and throw the error in your controller/component

```
1.        $this->cakeError('cannotWriteFile', array('file'=>'somefilename'));
```

The default implementation of `$this->_outputMessage(<view-filename>)` will just display the view in `views/errors/<view-filename>.ctp`. If you wish to override this behaviour, you can redefine `_outputMessage($template)` in your AppError class.

4.4 Debugging

Debugging is an inevitable and necessary part of any development cycle. While CakePHP doesn't offer any tools that directly connect with any IDE or editor, CakePHP does provide several tools to assist in debugging and exposing what is running under the hood of your application.

4.4.2 Using the Debugger Class

To use the debugger, first ensure that Configure::read('debug') is set to a value greater than 0.

dump($var)

Dump prints out the contents of a variable. It will print out all properties and methods (if any) of the supplied variable.

```
1.            $foo = array(1,2,3);
2.
3.            Debugger::dump($foo);
4.
5.            //outputs
6.            array(
7.                1,
8.                2,
9.                3
10.           )
11.
12.           //simple object
13.           $car = new Car();
```

```
14.
15.         Debugger::dump($car);
16.
17.         //outputs
18.         Car::
19.         Car::colour = 'red'
20.         Car::make = 'Toyota'
21.         Car::model = 'Camry'
22.         Car::mileage = '15000'
23.         Car::acclerate()
24.         Car::decelerate()
25.         Car::stop()
```

log($var, $level = 7)

Creates a detailed stack trace log at the time of invocation. The log() method prints out data similar to that done by Debugger::dump(), but to the debug.log instead of the output buffer. Note your app/tmp directory (and its contents) must be writable by the web server for log() to work correctly.

trace($options)

Returns the current stack trace. Each line of the trace includes the calling method, including which file and line the call originated from.

```
1.          //In PostsController::index()
2.          pr( Debugger::trace() );
3.
4.          //outputs
5.          PostsController::index() - APP/controllers/downloads_controller.php, line 48
6.          Dispatcher::_invoke() - CORE/cake/dispatcher.php, line 265
7.          Dispatcher::dispatch() - CORE/cake/dispatcher.php, line 237
8.          [main] - APP/webroot/index.php, line 84
```

Above is the stack trace generated by calling Debugger::trace() in a controller action. Reading the stack trace bottom to top shows the order of currently running functions (stack frames). In the above example, index.php called Dispatcher::dispatch(), which in-turn called Dispatcher::_invoke(). The _invoke() method then called PostsController::index(). This information is useful when working with recursive operations or deep stacks, as it identifies which functions are currently running at the time of the trace().

excerpt($file, $line, $context)

Grab an excerpt from the file at $path (which is an absolute filepath), highlights line number $line with $context number of lines around it.

```
1.          pr( Debugger::excerpt(ROOT.DS.LIBS.'debugger.php', 321, 2) );
2.
3.          //will output the following.
4.          Array
5.          (
6.              [0] => <code><span style="color: #000000"> * @access public</span></code>
7.              [1] => <code><span style="color: #000000"> */</span></code>
8.              [2] => <code><span style="color: #000000">       function excerpt($file, $line, $context = 2) {</span></code>
9.              [3] => <span class="code-highlight"><code><span style="color: #000000">           $data = $lines = array();</span></code></span>
10.             [4] => <code><span style="color: #000000">                        $data = @explode("\n", file_get_contents($file));</span></code>
11.         )
```

Although this method is used internally, it can be handy if you're creating your own error messages or log entries for custom situations.

exportVar($var, $recursion = 0)

Converts a variable of any type to a string for use in debug output. This method is also used by most of Debugger for internal variable conversions, and can be used in your own Debuggers as well.

invoke($debugger)

Replace the CakePHP Debugger with a new Error Handler.

4.4.3 Debugger Class

The debugger class was introduced with CakePHP 1.2 and offers even more options for obtaining debugging information. It has several functions which are invoked statically, and provide dumping, logging, and error handling functions.

The Debugger Class overrides PHP's default error handling, replacing it with far more useful error reports. The Debugger's error handling is used by default in CakePHP. As with all debugging functions, Configure::debug must be set to a value higher than 0.

When an error is raised, Debugger both outputs information to the page and makes an entry in the error.log file. The error report that is generated has both a stack trace and a code excerpt from where the error was raised. Click on the "Error" link type to reveal the stack trace, and on the "Code" link to reveal the error-causing lines.

4.5 Caching

Caching can be made use of on various levels within a CakePHP application. See how to disable browser caching, full page or element caching, per-request query caching or the Cache class - to cache anything. for more info.

4.6 Logging

While CakePHP core Configure Class settings can really help you see what's happening under the hood, there are certain times that you'll need to log data to the disk in order to find out what's going on. In a world that is becoming more dependent on technologies like SOAP and AJAX, debugging can be rather difficult.

Logging can also be a way to find out what's been going on in your application over time. What search terms are being used? What sorts of errors are my users being shown? How often is a particular query being executed?

Logging data in CakePHP is easy - the log() function is a part of the Object class, which is the common ancestor for almost all CakePHP classes. If the context is a CakePHP class (Model, Controller, Component... almost anything), you can log your data. You can also use CakeLog::write() directly.

4.6.1 Writing to logs

Writing to the log files can be done in 2 different ways. The first is to use the static `CakeLog::write()` method.

```
1.        CakeLog::write('debug', 'Something did not work');
```

The second is to use the log() shortcut function available on any class that extends `Object`. Calling log() will internally call CakeLog::write().

```
1.        //Executing this inside a CakePHP class:
2.        $this->log("Something did not work!", 'debug');
```

All configured log streams are written to sequentially each time `CakeLog::write()` is called. You do not need to configure a stream in order to use logging. If no streams are configured when the log is written to, a `default` stream using the core `FileLog` class will be configured to output into `app/tmp/logs/` just as CakeLog did in CakePHP 1.2

4.6.2 Using the default FileLog class

While CakeLog can be configured to write to a number of user configured logging adapters, it also comes with a default logging configuration. This configuration is identical to how CakeLog behaved in CakePHP 1.2. The default logging configuration will be used any time there are *no other* logging adapters configured. Once a logging adapter has been configured you will need to also configure FileLog if you want file logging to continue.

As its name implies FileLog writes log messages to files. The type of log message being written determines the name of the file the message is stored in. If a type is not supplied, LOG_ERROR is used which writes to the error log. The default log location is `app/tmp/logs/$type.log`

```
1.        //Executing this inside a CakePHP class:
2.         $this->log("Something didn't work!");
3.
4.        //Results in this being appended to app/tmp/logs/error.log
5.        2007-11-02 10:22:02 Error: Something didn't work!
```

You can specify a custom log names, using the second parameter. The default built-in FileLog class will treat this log name as the file you wish to write logs to.

```
1.      //called statically
2.      CakeLog::write('activity', 'A special message for activity logging');
3.
4.      //Results in this being appended to app/tmp/logs/activity.log (rather than error.log)
5.      2007-11-02 10:22:02 Activity: A special message for activity logging
```

The configured directory must be writable by the web server user in order for logging to work correctly.

You can configure additional/alternate FileLog locations using `CakeLog::config()`. FileLog accepts a `path` which allows for custom paths to be used.

```
1.      CakeLog::config('custom_path', array(
2.          'engine' => 'FileLog',
3.          'path' => '/path/to/custom/place/'
4.      ));
```

4.6.3 Creating and configuring log streams

Log stream handlers can be part of your application, or part of plugins. If for example you had a database logger called `DataBaseLogger`. As part of your application it would be placed in `app/libs/log/data_base_logger.php`. As part of a plugin it would be placed in `app/plugins/my_plugin/libs/log/data_base_logger.php`. When configured `CakeLog` will attempt to load Configuring log streams is done by calling `CakeLog::config()`. Configuring our DataBaseLogger would look like

```
1.      //for app/libs
2.      CakeLog::config('otherFile', array(
3.          'engine' => 'DataBaseLogger',
4.          'model' => 'LogEntry',
5.          ...
6.      ));
7.      //for plugin called LoggingPack
```

```
CakeLog::config('otherFile', array(
    'engine' => 'LoggingPack.DataBaseLogger',
    'model' => 'LogEntry',
    ...
));
```

When configuring a log stream the `engine` parameter is used to locate and load the log handler. All of the other configuration properties are passed to the log stream's constructor as an array.

```
class DataBaseLogger {
    function __construct($options = array()) {
        //...
    }
}
```

CakePHP has no requirements for Log streams other than that they must implement a `write` method. This write method must take two parameters `$type, $message` in that order. `$type` is the string type of the logged message, core values are `error`, `warning`, `info` and `debug`. In addition you can define your own types by using them when you call `CakeLog::write`.

It should be noted that you will encounter errors when trying to configure application level loggers from `app/config/core.php`. This is because paths are not yet bootstrapped. Configuring of loggers should be done in `app/config/bootstrap.php` to ensure classes are properly loaded.

4.6.4 Interacting with log streams

You can introspect the configured streams with `CakeLog::configured()`. The return of `configured()` is an array of all the currently configured streams. You can remove streams using `CakeLog::drop($key)`. Once a log stream has been dropped it will no longer receive messages.

4.6.5 Error logging

Errors are now logged when `Configure::write('debug', 0);`. You can use `Configure::write('log', $val)`, to control which errors are logged when debug is off. By default all errors are logged.

```
1.        Configure::write('log', E_WARNING);
```

Would log only warning and fatal errors. Setting `Configure::write('log', false);` will disable error logging when debug = 0.

4.7 Testing

As of CakePHP 1.2 there is support for a comprehensive testing framework built into CakePHP. The framework is an extension of the SimpleTest framework for PHP. This section will discuss how to prepare for testing and how to build and run your tests.

4.7.1 Preparing for testing

Ready to start testing? Good! Lets get going then!

4.7.1.1 Installing SimpleTest

The testing framework provided with CakePHP 1.3 is built upon the SimpleTest testing framework. SimpleTest is not shipped with the default CakePHP installation, so we need to download it first. You can find it here: http://simpletest.sourceforge.net/.

Fetch the latest version, and unzip the code to your `vendors` folder, or your `app/vendors` folder, depending on your preference. You should now have a `vendors/simpletest` directory with all SimpleTest files and folders inside. Remember to have a DEBUG level of at least 1 in your `app/config/core.php` file before running any tests!

There is a new version of SimpleTest 1.1alpha that does not work with CakePHP. Please use 1.0.1.

If you have no test database connection defined in your `app/config/database.php`, test tables will be created with a `test_suite_` prefix. You can create a `$test` database connection to contain any test tables like the one below:

```
1.            var $test = array(
2.                'driver' => 'mysql',
3.                'persistent' => false,
```

```
4.                 'host' => 'dbhost',
5.                 'login' => 'dblogin',
6.                 'password' => 'dbpassword',
7.                 'database' => 'databaseName'
8.             );
```

If the test database is available and CakePHP can connect to it, all tables will be created in this database.

4.7.1.2 Running Core test cases

After installing Simpletest, you can run the core test cases. They are part of every packaged release and can also be found in the git repository.

The tests can then be accessed by browsing to http://your.cake.domain/test.php - depending on how your specific setup looks. Try executing one of the core test groups by clicking on the corresponding link. Executing a test group might take a while, but you should eventually see something like "2/2 test cases complete: 49 passes, 0 fails and 0 exceptions.".

Congratulations, you are now ready to start writing tests!

If you run all of the core tests at once or run core test groups most of them will fail. This is known by the CakePHP developers and is normal so don't panic. Instead, try running each of the core test cases individually.

4.7.2 Testing overview - Unit testing vs. Web testing

The CakePHP test framework supports two types of testing. One is Unit Testing, where you test small parts of your code, such as a method in a component or an action in a controller. The other type of testing supported is Web Testing, where you automate the work of testing your application through navigating pages, filling forms, clicking links and so on.

4.7.3 Preparing test data

4.7.3.1 About fixtures

When testing code that depends on models and data, one can use **fixtures** as a way to generate temporary data tables loaded with sample data that can be used by the test. The benefit of using fixtures is that your test has no chance of disrupting live application data. In addition, you can begin testing your code prior to actually developing live content for an application.

CakePHP attempts to use the connection named `$test` in your `app/config/database.php` configuration file. If this connection is not usable, it will use the `$default` database configuration and create the test tables in the database defined in that configuration. If the default connection is used, the test suite will use "test_suite_" as a prefix to help prevent collision with your existing tables.

CakePHP performs the following during the course of a fixture based test case:

1. Creates tables for each of the fixtures needed
2. Populates tables with data, if data is provided in fixture
3. Runs test methods
4. Empties the fixture tables
5. Removes fixture tables from database

4.7.3.2 Creating fixtures

When creating a fixture you will mainly define two things: how the table is created (which fields are part of the table), and which records will be initially populated to the test table. Let's then create our first fixture, that will be used to test our own Article model. Create a file named **article_fixture.php** in your **app/tests/fixtures** directory, with the following content:

```
1.      <?php
2.       class ArticleFixture extends CakeTestFixture {
3.            var $name = 'Article';
4.
5.            var $fields = array(
6.                'id' => array('type' => 'integer', 'key' => 'primary'),
7.                'title' => array('type' => 'string', 'length' => 255, 'null' => false),
8.                'body' => 'text',
9.                'published' => array('type' => 'integer', 'default' => '0', 'null' => false),
```

```
10.                 'created' => 'datetime',
11.                 'updated' => 'datetime'
12.             );
13.             var $records = array(
14.                 array ('id' => 1, 'title' => 'First Article', 'body' => 'First Article Body', 'published' => '1', 'created' => '2007-03-18 10:39:23', 'updated' => '2007-03-18 10:41:31'),
15.                 array ('id' => 2, 'title' => 'Second Article', 'body' => 'Second Article Body', 'published' => '1', 'created' => '2007-03-18 10:41:23', 'updated' => '2007-03-18 10:43:31'),
16.                 array ('id' => 3, 'title' => 'Third Article', 'body' => 'Third Article Body', 'published' => '1', 'created' => '2007-03-18 10:43:23', 'updated' => '2007-03-18 10:45:31')
17.             );
18.         }
19.         ?>
```

The `$name` variable is extremely significant. If you omit it, cake will use the wrong table names when it sets up your test database, and you'll get strange errors that are difficult to debug. If you use PHP 5.2, you might be used to writing model classes without `$name`, but you must remember to include it in your fixture files.

We use $fields to specify which fields will be part of this table, on how they are defined. The format used to define these fields is the same used in the function **generateColumnSchema()** defined on Cake's database engine classes (for example, on file dbo_mysql.php.) Let's see the available attributes a field can take and their meaning:

type

CakePHP internal data type. Currently supported: string (maps to VARCHAR), text (maps to TEXT), integer (maps to INT), float (maps to FLOAT), datetime (maps to DATETIME), timestamp (maps to TIMESTAMP), time (maps to TIME), date (maps to DATE), and binary (maps to BLOB)

key

set to primary to make the field AUTO_INCREMENT, and a PRIMARY KEY for the table.

length

set to the specific length the field should take.

null

set to either true (to allow NULLs) or false (to disallow NULLs)

default

default value the field takes.

We lastly can set a set of records that will be populated after the test table is created. The format is fairly straight forward and needs little further explanation. Just keep in mind that each record in the $records array must have a key for **every** field specified in the $fields array. If a field for a particular record needs to have a NULL value, just specify the value of that key as NULL.

4.7.3.3 Importing table information and records

Your application may have already working models with real data associated to them, and you might decide to test your model with that data. It would be then a duplicate effort to have to define the table definition and/or records on your fixtures. Fortunately, there's a way for you to define that table definition and/or records for a particular fixture come from an existing model or an existing table.

Let's start with an example. Assuming you have a model named Article available in your application (that maps to a table named articles), change the example fixture given in the previous section (**app/tests/fixtures/article_fixture.php**) to:

```
1.  <?php
2.      class ArticleFixture extends CakeTestFixture {
3.              var $name = 'Article';
4.              var $import = 'Article';
5.      }
6.      ?>
```

This statement tells the test suite to import your table definition from the table linked to the model called Article. You can use any model available in your application. The statement above does not import records, you can do so by changing it to:

```
<?php
class ArticleFixture extends CakeTestFixture {
    var $name = 'Article';
    var $import = array('model' => 'Article', 'records' => true);
}
?>
```

If on the other hand you have a table created but no model available for it, you can specify that your import will take place by reading that table information instead. For example:

```
<?php
   class ArticleFixture extends CakeTestFixture {
          var $name = 'Article';
          var $import = array('table' => 'articles');
   }
 ?>
```

Will import table definition from a table called 'articles' using your CakePHP database connection named 'default'. If you want to change the connection to use just do:

```
<?php
   class ArticleFixture extends CakeTestFixture {
   var $name = 'Article';
   var $import = array('table' => 'articles', 'connection' => 'other');
   }
   ?>
```

Since it uses your CakePHP database connection, if there's any table prefix declared it will be automatically used when fetching table information. The two snippets above do not import records from the table. To force the fixture to also import its records, change it to:

```
<?php
    class ArticleFixture extends CakeTestFixture {
        var $name = 'Article';
        var $import = array('table' => 'articles', 'records' => true);
    }
?>
```

You can naturally import your table definition from an existing model/table, but have your records defined directly on the fixture as it was shown on previous section. For example:

```
<?php
    class ArticleFixture extends CakeTestFixture {
        var $name = 'Article';
        var $import = 'Article';

        var $records = array(
            array ('id' => 1, 'title' => 'First Article', 'body' => 'First Article Body', 'published' => '1', 'created' => '2007-03-18 10:39:23', 'updated' => '2007-03-18 10:41:31'),
            array ('id' => 2, 'title' => 'Second Article', 'body' => 'Second Article Body', 'published' => '1', 'created' => '2007-03-18 10:41:23', 'updated' => '2007-03-18 10:43:31'),
            array ('id' => 3, 'title' => 'Third Article', 'body' => 'Third Article Body', 'published' => '1', 'created' => '2007-03-18 10:43:23', 'updated' => '2007-03-18 10:45:31')
        );
    }
?>
```

4.7.4 Creating tests

First, lets go through a number of rules, or guidelines, concerning tests:

1. PHP files containing tests should be in your **app/tests/cases/[some_folder]**.
2. The filenames of these files should end in **.test.php** instead of just .php.
3. The classes containing tests should extend **CakeTestCase** or **CakeWebTestCase**.
4. The name of any method containing a test (i.e. containing an assertion) should begin with **test**, as in **testPublished()**.

When you have created a test case, you can execute it by browsing to **http://your.cake.domain/cake_folder/test.php** (depending on how your specific setup looks) and clicking App test cases, and then click the link to your specific file.

4.7.4.1 CakeTestCase Callback Methods

If you want to sneak in some logic just before or after an individual CakeTestCase method, and/or before or after your entire CakeTestCase, the following callbacks are available:

start()
First method called in a *test case*.

end()
Last method called in a *test case*.

startCase()
called before a *test case* is started.

endCase()
called after a *test case* has run.

before($method)
Announces the start of a *test method*.

after($method)
Announces the end of a *test method*.

startTest($method)

Called just before a *test method* is executed.

endTest($method)

Called just after a *test method* has completed.

4.7.5 Testing models

4.7.5.1 Creating a test case

Let's say we already have our Article model defined on app/models/article.php, which looks like this:

```
1.      <?php
2.          class Article extends AppModel {
3.                  var $name = 'Article';
4.
5.                  function published($fields = null) {
6.                      $params = array(
7.                              'conditions' => array(
8.                                      $this->name . '.published' => 1
9.                              ),
10.                             'fields' => $fields
11.                     );
12.
13.                     return $this->find('all',$params);
14.                 }
15.
16.         }
17.     ?>
```

We now want to set up a test that will use this model definition, but through fixtures, to test some functionality in the model. CakePHP test suite loads a very minimum set of files (to keep tests isolated), so we have to start by loading our parent model (in this case the Article model which we already

defined), and then inform the test suite that we want to test this model by specifying which DB configuration it should use. CakePHP test suite enables a DB configuration named **test_suite** that is used for all models that rely on fixtures. Setting $useDbConfig to this configuration will let CakePHP know that this model uses the test suite database connection.

CakePHP Models will only use the test_suite DB config if they rely on fixtures in your testcase!

Since we also want to reuse all our existing model code we will create a test model that will extend from Article, set $useDbConfig and $name appropiately. Let's now create a file named **article.test.php** in your **app/tests/cases/models** directory, with the following contents:

```
1.      <?php
2.         App::import('Model','Article');
3.
4.         class ArticleTestCase extends CakeTestCase {
5.                var $fixtures = array( 'app.article' );
6.         }
7.       ?>
```

We have created the ArticleTestCase. In variable **$fixtures** we define the set of fixtures that we'll use.

If your model is associated with other models, you will need to include ALL the fixtures for each associated model even if you don't use them. For example: A hasMany B hasMany C hasMany D. In ATestCase you will have to include fixtures for a, b, c and d.

4.7.5.2 Creating a test method

Let's now add a method to test the function published() in the Article model. the file **app/tests/cases/models/article.test.php** so it now looks like this:

```
1.      <?php
2.          App::import('Model', 'Article');
3.
4.          class ArticleTestCase extends CakeTestCase {
5.              var $fixtures = array( 'app.article' );
```

```
6.
7.              function testPublished() {
8.                  $this->Article =& ClassRegistry::init('Article');
9.
10.                 $result = $this->Article->published(array('id', 'title'));
11.                 $expected = array(
12.                     array('Article' => array( 'id' => 1, 'title' => 'First Article' )),
13.                     array('Article' => array( 'id' => 2, 'title' => 'Second Article' )),
14.                     array('Article' => array( 'id' => 3, 'title' => 'Third Article' ))
15.                 );
16.
17.                 $this->assertEqual($result, $expected);
18.             }
19.         }
20.         ?>
```

You can see we have added a method called **testPublished()**. We start by creating an instance of our fixture based **Article** model, and then run our **published()** method. In **$expected** we set what we expect should be the proper result (that we know since we have defined which records are initally populated to the article table.) We test that the result equals our expectation by using the **assertEqual** method. See the section Creating Tests for information on how to run the test.

4.7.6 Testing controllers

4.7.6.1 Creating a test case

Say you have a typical articles controller, with its corresponding model, and it looks like this:

```
1.      <?php
2.      class ArticlesController extends AppController {
3.          var $name = 'Articles';
```

```
4.          var $helpers = array('Ajax', 'Form', 'Html');
5.
6.          function index($short = null) {
7.            if (!empty($this->data)) {
8.              $this->Article->save($this->data);
9.            }
10.           if (!empty($short)) {
11.             $result = $this->Article->findAll(null, array('id',
12.                'title'));
13.           } else {
14.             $result = $this->Article->findAll();
15.           }
16.
17.           if (isset($this->params['requested'])) {
18.             return $result;
19.           }
20.
21.           $this->set('title', 'Articles');
22.           $this->set('articles', $result);
23.         }
24.       }
25.       ?>
```

Create a file named articles_controller.test.php in your app/tests/cases/controllers directory and put the following inside:

```
1.        <?php
2.        class ArticlesControllerTest extends CakeTestCase {
3.           function startCase() {
4.             echo '<h1>Starting Test Case</h1>';
5.           }
```

```
6.          function endCase() {
7.            echo '<h1>Ending Test Case</h1>';
8.          }
9.          function startTest($method) {
10.           echo '<h3>Starting method ' . $method . '</h3>';
11.         }
12.         function endTest($method) {
13.           echo '<hr />';
14.         }
15.         function testIndex() {
16.           $result = $this->testAction('/articles/index');
17.           debug($result);
18.         }
19.         function testIndexShort() {
20.           $result = $this->testAction('/articles/index/short');
21.           debug($result);
22.         }
23.         function testIndexShortGetRenderedHtml() {
24.           $result = $this->testAction('/articles/index/short',
25.           array('return' => 'render'));
26.           debug(htmlentities($result));
27.         }
28.         function testIndexShortGetViewVars() {
29.           $result = $this->testAction('/articles/index/short',
30.           array('return' => 'vars'));
31.           debug($result);
32.         }
33.         function testIndexFixturized() {
34.           $result = $this->testAction('/articles/index/short',
35.           array('fixturize' => true));
36.           debug($result);
```

```
37.         }
38.         function testIndexPostFixturized() {
39.           $data = array('Article' => array('user_id' => 1, 'published'
40.                => 1, 'slug'=>'new-article', 'title' => 'New Article', 'body' => 'New Body'));
41.           $result = $this->testAction('/articles/index',
42.           array('fixturize' => true, 'data' => $data, 'method' => 'post'));
43.           debug($result);
44.         }
45.       }
46.       ?>
```

4.7.6.2 The testAction method

The new thing here is the **testAction** method. The first argument of that method is the Cake url of the controller action to be tested, as in '/articles/index/short'.

The second argument is an array of parameters, consisting of:

return

Set to what you want returned.

Valid values are:

- 'vars' - You get the view vars available after executing action
- 'view' - You get The rendered view, without the layout
- 'contents' - You get the rendered view's complete html, including the layout
- 'result' - You get the returned value when action uses $this->params['requested'].

The default is 'result'.

fixturize

Set to true if you want your models auto-fixturized (so your application tables get copied, along with their records, to test tables so if you change data it does not affect your real application.) If you set 'fixturize' to an array of models, then only those models will be auto-fixturized while the other will remain with live tables. If you wish to use your fixture files with testAction() do not use fixturize, and instead just use fixtures as you normally would.

method

set to 'post' or 'get' if you want to pass data to the controller

data

the data to be passed. Set it to be an associative array consisting of fields => value. Take a look at `function testIndexPostFixturized()` in above test case to see how we emulate posting form data for a new article submission.

4.7.6.3 Pitfalls

If you use testAction to test a method in a controller that does a redirect, your test will terminate immediately, not yielding any results. See https://trac.cakephp.org/ticket/4154 for a possible fix.

4.7.7 Testing Helpers

Since a decent amount of logic resides in Helper classes, it's important to make sure those classes are covered by test cases.

Helper testing is a bit similar to the same approach for Components. Suppose we have a helper called CurrencyRendererHelper located in `app/views/helpers/currency_renderer.php` with its accompanying test case file located in `app/tests/cases/helpers/currency_renderer.test.php`

4.7.7.1 Creating Helper test, part I

First of all we will define the responsibilities of our CurrencyRendererHelper. Basically, it will have two methods just for demonstration purpose:

function usd($amount)

This function will receive the amount to render. It will take 2 decimal digits filling empty space with zeros and prefix 'USD'.

function euro($amount)

This function will do the same as usd() but prefix the output with 'EUR'. Just to make it a bit more complex, we will also wrap the result in span tags:

```
1.      <span class="euro"></span>
```

Let's create the tests first:

```
1.      <?php
2.      //Import the helper to be tested.
3.      //If the tested helper were using some other helper, like Html,
4.      //it should be impoorted in this line, and instantialized in startTest().
5.      App::import('Helper', 'CurrencyRenderer');
6.      class CurrencyRendererTest extends CakeTestCase {
7.          private $currencyRenderer = null;
8.          //Here we instantiate our helper, and all other helpers we need.
9.          public function startTest() {
10.             $this->currencyRenderer = new CurrencyRendererHelper();
11.         }
12.         //testing usd() function.
13.         public function testUsd() {
14.             $this->assertEqual('USD 5.30', $this->currencyRenderer->usd(5.30));
15.             //We should always have 2 decimal digits.
16.             $this->assertEqual('USD 1.00', $this->currencyRenderer->usd(1));
17.             $this->assertEqual('USD 2.05', $this->currencyRenderer->usd(2.05));
18.             //Testing the thousands separator
```

```
19.             $this->assertEqual('USD 12,000.70', $this->currencyRenderer->usd(12000.70));
20.         }
21.     }
```

Here, we call `usd()` with different parameters and tell the test suite to check if the returned values are equal to what is expected.

Executing the test now will result in errors (because currencyRendererHelper doesn't even exist yet) showing that we have 3 fails.

Once we know what our method should do, we can write the method itself:

```
1.      <?php
2.      class CurrencyRendererHelper extends AppHelper {
3.          public function usd($amount) {
4.              return 'USD ' . number_format($amount, 2, '.', ',');
5.          }
6.      }
```

Here we set the decimal places to 2, decimal separator to dot, thousands separator to comma, and prefix the formatted number with 'USD' string.

Save this in `app/views/helpers/currency_renderer.php` and execute the test. You should see a green bar and messaging indicating 4 passes.

4.7.8 Testing components

Lets assume that we want to test a component called TransporterComponent, which uses a model called Transporter to provide functionality for other controllers. We will use four files:

- A component called Transporters found in **app/controllers/components/transporter.php**
- A model called Transporter found in **app/models/transporter.php**
- A fixture called TransporterTestFixture found in **app/tests/fixtures/transporter_fixture.php**
- The testing code found in **app/tests/cases/transporter.test.php**

4.7.8.1 Initializing the component

Since CakePHP discourages from importing models directly into components we need a controller to access the data in the model.

If the startup() function of the component looks like this:

```
1.        public function startup(&$controller){
2.                  $this->Transporter = $controller->Transporter;
3.         }
```

then we can just design a really simple fake class:

```
1.        class FakeTransporterController {}
```

and assign values into it like this:

```
1.        $this->TransporterComponentTest = new TransporterComponent();
2.        $controller = new FakeTransporterController();
3.        $controller->Transporter = new TransporterTest();
4.        $this->TransporterComponentTest->startup(&$controller);
```

4.7.8.2 Creating a test method

Just create a class that extends CakeTestCase and start writing tests!

```
1.        class TransporterTestCase extends CakeTestCase {
2.            var $fixtures = array('transporter');
3.            function testGetTransporter() {
4.                  $this->TransporterComponentTest = new TransporterComponent();
5.                  $controller = new FakeTransporterController();
6.                  $controller->Transporter = new TransporterTest();
```

```
7.                  $this->TransporterComponentTest->startup(&$controller);
8.
9.                  $result = $this->TransporterComponentTest->getTransporter("12345", "Sweden", "54321", "Sweden");
10.                 $this->assertEqual($result, 1, "SP is best for 1xxxx-5xxxx");
11.
12.                 $result = $this->TransporterComponentTest->getTransporter("41234", "Sweden", "44321", "Sweden");
13.                 $this->assertEqual($result, 2, "WSTS is best for 41xxx-44xxx");
14.
15.                 $result = $this->TransporterComponentTest->getTransporter("41001", "Sweden", "41870", "Sweden");
16.                 $this->assertEqual($result, 3, "GL is best for 410xx-419xx");
17.
18.                 $result = $this->TransporterComponentTest->getTransporter("12345", "Sweden", "54321", "Norway");
19.                 $this->assertEqual($result, 0, "Noone can service Norway");
20.             }
21.         }
22.
```

4.7.9 Web testing - Testing views

Most, if not all, CakePHP projects result in a web application. While unit tests are an excellent way to test small parts of functionality, you might also want to test the functionality on a large scale. The **CakeWebTestCase** class provides a good way of doing this testing from a user point-of-view.

4.7.9.1 About CakeWebTestCase

CakeWebTestCase is a direct extension of the SimpleTest WebTestCase, without any extra functionality. All the functionality found in the SimpleTest documentation for Web testing is also available here. This also means that no functionality other than that of SimpleTest is available. This means that you cannot use fixtures, and **all web test cases involving updating/saving to the database will permanently change your database values**. Test results are often based on what values the database holds, so making sure the database contains the values you expect is part of the testing procedure.

4.7.9.2 Creating a test

In keeping with the other testing conventions, you should create your view tests in tests/cases/views. You can, of course, put those tests anywhere but following the conventions whenever possible is always a good idea. So let's create the file tests/cases/views/complete_web.test.php

First, when you want to write web tests, you must remember to extend **CakeWebTestCase** instead of CakeTestCase:

```
1.        class CompleteWebTestCase extends CakeWebTestCase
```

If you need to do some preparation before you start the test, create a constructor:

```
1.        function CompleteWebTestCase(){
2.          //Do stuff here
3.        }
```

When writing the actual test cases, the first thing you need to do is get some output to look at. This can be done by doing a **get** or **post** request, using **get()** or **post()** respectively. Both these methods take a full url as the first parameter. This can be dynamically fetched if we assume that the test script is located under http://your.domain/cake/folder/webroot/test.php by typing:

```
1.        $this->baseurl = current(split("webroot", $_SERVER['PHP_SELF']));
```

You can then do gets and posts using Cake urls, like this:

```
1.        $this->get($this->baseurl."/products/index/");
2.        $this->post($this->baseurl."/customers/login", $data);
```

The second parameter to the post method, **$data**, is an associative array containing the post data in Cake format:

```
1.        $data = array(
2.          "data[Customer][mail]" => "user@user.com",
3.          "data[Customer][password]" => "userpass");
```

When you have requested the page you can do all sorts of asserts on it, using standard SimpleTest web test methods.

4.7.9.3 Walking through a page

CakeWebTest also gives you an option to navigate through your page by clicking links or images, filling forms and clicking buttons. Please refer to the SimpleTest documentation for more information on that.

4.7.10 Testing plugins

Tests for plugins are created in their own directory inside the plugins folder.

```
/app
    /plugins
        /pizza
            /tests
                /cases
                /fixtures
                /groups
```

They work just like normal tests but you have to remember to use the naming conventions for plugins when importing classes. This is an example of a testcase for the PizzaOrder model from the plugins chapter of this manual. A difference from other tests is in the first line where 'Pizza.PizzaOrder' is imported. You also need to prefix your plugin fixtures with '`plugin.plugin_name.`'.

```
1.  <?php
2.  App::import('Model', 'Pizza.PizzaOrder');
3.  class PizzaOrderCase extends CakeTestCase {
4.      // Plugin fixtures located in /app/plugins/pizza/tests/fixtures/
5.      var $fixtures = array('plugin.pizza.pizza_order');
6.      var $PizzaOrderTest;
7.
8.      function testSomething() {
9.          // ClassRegistry makes the model use the test database connection
10.         $this->PizzaOrderTest =& ClassRegistry::init('PizzaOrder');
```

```
11.                 // do some useful test here
12.                 $this->assertTrue(is_object($this->PizzaOrderTest));
13.             }
14.         }
15.         ?>
```

If you want to use plugin fixtures in the app tests you can reference them using 'plugin.pluginName.fixtureName' syntax in the $fixtures array.

That is all there is to it.

4.7.11 Miscellaneous

4.7.11.1 Customizing the test reporter

The standard test reporter is **very** minimalistic. If you want more shiny output to impress someone, fear not, it is actually very easy to extend. By creating a new reporter and making a request with a matching `output` GET parameter you can get test results with a custom reporter.

Reporters generate the visible output from the test suite. There are two built in reporters: Text and Html. By default all web requests use the Html reporter. You can create your own reporters by creating files in your app/libs. For example you could create the file `app/libs/test_suite/reporters/my_reporter.php` and in it create the following:

```
1.         require_once CAKE_TEST_LIB . 'reporter' . DS . 'cake_base_reporter.php';
2.         class MyReporter extends CakeBaseReporter {
3.             //methods go here.
4.         }
```

Extending `CakeBaseReporter` or one of its subclasses is not required, but strongly suggested as you may get missing errors otherwise. `CakeBaseReporter` encapsulates a few common test suite features such as test case timing and code coverage report generation. You can use your custom reporter by setting the `output` query string parameter to the reporter name minus 'reporter'. For the example above you would set `output=my` to use your custom reporter.

4.7.11.2 Test Reporter methods

Reporters have a number of methods used to generate the various parts of a Test suite response.

paintDocumentStart()

Paints the start of the response from the test suite. Used to paint things like head elements in an html page.

paintTestMenu()

Paints a menu of available test cases.

testCaseList()

Retrieves and paints the list of tests cases.

groupCaseList()

Retrieves and paints the list of group tests.

paintHeader()

Prints before the test case/group test is started.

paintPass()

Prints everytime a test case has passed. Use $this->getTestList() to get an array of information pertaining to the test, and $message to get the test result. Remember to call parent::paintPass($message).

paintFail()

Prints everytime a test case has failed. Remember to call parent::paintFail($message).

paintSkip()

Prints everytime a test case has been skipped. Remember to call parent::paintSkip($message).

paintException()

Prints everytime there is an uncaught exception. Remember to call parent::paintException($message).paintError()

Prints everytime an error is raised. Remember to call parent::paintError($message).

paintFooter()

Prints when the test case/group test is over, i.e. when all test cases has been executed.

paintDocumentEnd()

Paints the end of the response from the test suite. Used to paint things like footer elements in an html page.

4.7.11.3 Grouping tests

If you want several of your test to run at the same time, you can try creating a test group. Create a file in **/app/tests/groups/** and name it something like **your_test_group_name.group.php**. In this file, extend **TestSuite** and import test as follows:

```
1.  <?php
2.  class TryGroupTest extends TestSuite {
3.    var $label = 'try';
4.    function tryGroupTest() {
5.      TestManager::addTestCasesFromDirectory($this, APP_TEST_CASES . DS . 'models');
6.    }
7.  }
```

```
8.      ?>
```

The code above will group all test cases found in the **/app/tests/cases/models/** folder. To add an individual file, use **TestManager::addTestFile**($this, filename).

4.7.12 Running tests in the Command Line

If you have simpletest installed you can run your tests from the command line of your application.

from **app/**

```
cake testsuite help
Usage:
        cake testsuite category test_type file
                - category - "app", "core" or name of a plugin
                - test_type - "case", "group" or "all"
                - test_file - file name with folder prefix and without the (test|group).php suffix

Examples:
                cake testsuite app all
                cake testsuite core all

                cake testsuite app case behaviors/debuggable
                cake testsuite app case models/my_model
                cake testsuite app case controllers/my_controller

                cake testsuite core case file
                cake testsuite core case router
                cake testsuite core case set

                cake testsuite app group mygroup
                cake testsuite core group acl
                cake testsuite core group socket
```

```
                cake testsuite bugs case models/bug
                  // for the plugin 'bugs' and its test case 'models/bug'
                cake testsuite bugs group bug
                  // for the plugin bugs and its test group 'bug'

Code Coverage Analysis:


Append 'cov' to any of the above in order to enable code coverage analysis
```

As the help menu suggests, you'll be able to run all, part, or just a single test case from your app, plugin, or core, right from the command line.

If you have a model test of **test/models/my_model.test.php** you'd run just that test case by running:

```
cake testsuite app models/my_model
```

4.7.13 Test Suite changes in 1.3

The TestSuite harness for 1.3 was heavily refactored and partially rebuilt. The number of constants and global functions have been greatly reduced. Also the number of classes used by the test suite has been reduced and refactored. You **must** update `app/webroot/test.php` to continue using the test suite. We hope that this will be the last time that a change is required to `app/webroot/test.php`.

Removed Constants

- `CAKE_TEST_OUTPUT`
- `RUN_TEST_LINK`
- `BASE`
- `CAKE_TEST_OUTPUT_TEXT`
- `CAKE_TEST_OUTPUT_HTML`

These constants have all been replaced with instance variables on the reporters and the ability to switch reporters.

Removed functions

- `CakePHPTestHeader()`
- `CakePHPTestSuiteHeader()`
- `CakePHPTestSuiteFooter()`
- `CakeTestsGetReporter()`
- `CakePHPTestRunMore()`
- `CakePHPTestAnalyzeCodeCoverage()`
- `CakePHPTestGroupTestList()`
- `CakePHPTestCaseList()`

These methods and the logic they contained have been refactored/rewritten into `CakeTestSuiteDispatcher` and the relevant reporter classes. This made the test suite more modular and easier to extend.

Removed Classes

- HtmlTestManager
- TextTestManager
- CliTestManager

These classes became obsolete as logic was consolidated into the reporter classes.

Modified methods/classes

The following methods have been changed as noted.

- `TestManager::getExtension()` is no longer static.
- `TestManager::runAllTests()` is no longer static.
- `TestManager::runGroupTest()` is no longer static.
- `TestManager::runTestCase()` is no longer static.
- `TestManager::getTestCaseList()` is no longer static.
- `TestManager::getGroupTestList()` is no longer static.

testsuite Console changes

The output of errors, exceptions, and failures from the testsuite console tool have been updated to remove redundant information and increase readability of the messages. If you have other tools built upon the testsuite console, be sure to update those tools with the new formatting.

CodeCoverageManager changes

- `CodeCoverageManager::start()`'s functionality has been moved to `CodeCoverageManager::init()`
- `CodeCoverageManager::start()` now starts coverage generation.
- `CodeCoverageManager::stop()` pauses collection
- `CodeCoverageManager::clear()` stops and clears collected coverage reports.

4.8 Internationalization & Localization

One of the best ways for your applications to reach a larger audience is to cater for multiple languages. This can often prove to be a daunting task, but the internationalization and localization features in CakePHP make it much easier.

First, it's important to understand some terminology. *Internationalization* refers to the ability of an application to be localized. The term *localization* refers to the adaptation of an application to meet specific language (or culture) requirements (i.e., a "locale"). Internationalization and localization are often abbreviated as i18n and l10n respectively; 18 and 10 are the number of characters between the first and last character.

4.8.1 Internationalizing Your Application

There are only a few steps to go from a single-language application to a multi-lingual application, the first of which is to make use of the `__()` function in your code. Below is an example of some code for a single-language application:

```
1.      <h2>Posts</h2>
```

To internationalize your code, all you need to do is to wrap strings in the translate function like so:

```
1.      <h2><?php __('Posts') ?></h2>
```

If you do nothing further, these two code examples are functionally identical - they will both send the same content to the browser. The `__()` function will translate the passed string if a translation is available, or return it unmodified. It works similar to other Gettext implementations (as do the other translate functions, such as `__d()`, `__n()` etc)

With your code ready to be multilingual, the next step is to create your pot file, which is the template for all translatable strings in your application. To generate your pot file(s), all you need to do is run the i18n console task, which will look for where you've used a translate function in your code and generate your pot file(s) for you. You can and should re-run this console task any time you change the translations in your code.

The pot file(s) themselves are not used by CakePHP, they are the templates used to create or update your po files, which contain the translations. Cake will look for your po files in the following location:

```
1.      /app/locale/<locale>/LC_MESSAGES/<domain>.po
```

The default domain is 'default', therefore your locale folder would look something like this:

```
1.      /app/locale/eng/LC_MESSAGES/default.po (English)
2.      /app/locale/fre/LC_MESSAGES/default.po (French)
3.      /app/locale/por/LC_MESSAGES/default.po (Portuguese)
```

To create or edit your po files it's recommended that you do *not* use your favorite editor. To create a po file for the first time it is possible to copy the pot file to the correct location and change the extension *however* unless you're familiar with their format, it's quite easy to create an invalid po file or to save it as the wrong charset (if you're editing manually, use UTF-8 to avoid problems). There are free tools such as Po which make editing and updating your po files an easy task; especially for updating an existing po file with a newly updated pot file.

The three-character locale codes conform to the ISO 639-2 standard, although if you create regional locales (en_US, en_GB, etc.) cake will use them if appropriate.

there is a 1014-character limit for each msgstr value (source needed).

Remember that po files are useful for short messages, if you find you want to translate long paragraphs, or even whole pages - you should consider implementing a different solution. e.g.:

```
1.      // App Controller Code.
2.      function beforeFilter() {
3.          $locale = Configure::read('Config.language');
4.          if ($locale && file_exists(VIEWS . $locale . DS . $this->viewPath)) {
5.              // e.g. use /app/views/fre/pages/tos.ctp instead of /app/views/pages/tos.ctp
6.              $this->viewPath = $locale . DS . $this->viewPath;
7.          }
8.      }
```

or

```
1.      // View code
2.      echo $this->element(Configure::read('Config.language') . '/tos')
```

4.8.2 Localization in CakePHP

To change or set the language for your application, all you need to do is the following:

```
1.      Configure::write('Config.language', 'fre');
```

This tells Cake which locale to use (if you use a regional locale, such as fr_FR, it will use the ISO 639-2 locale as a fallback if it doesn't exist), you can change the language at any time, e.g. in your bootstrap if you're setting the application default language, in your (app) controller beforeFilter if it's specific to the request or user, or in fact anytime at all before you want a message in a different language.

To set the language for the current user, store the setting in the Session object, like this:

```
1.      $this->Session->write('Config.language', 'fre');
```

It's a good idea to serve up public content available in multiple languages from a unique url - this makes it easy for users (and search engines) to find what they're looking for in the language they are expecting. There are several ways to do this, it can be by using language specific subdomains (en.example.com, fra.example.com, etc.), or using a prefix to the url such as is done with this application. You may also wish to glean the information from the browser's user-agent, among other things.

As mentioned in the previous section, displaying localized content is done using the __() convenience function, or one of the other translation functions all of which are globally available, but probably be best utilized in your views. The first parameter of the function is used as the msgid defined in the .po files.

Remember to use the return parameter for the various __* methods if you don't want the string echo'ed directly. For example:

```
1.  <?php
2.  echo $form->error(
3.      'Card.cardNumber',
4.      __("errorCardNumber", true),
5.      array('escape' => false)
6.  );
7.  ?>
```

If you would like to have all of your validation error messages translated by default, a simple solution would be to add the following code in you app_model.php:

```
1.  function invalidate($field, $value = true) {
2.      return parent::invalidate($field, __($value, true));
3.  }
```

The i18n console task will not be able to determine the message id from the above example, which means you'll need to add the entries to your pot file manually (or via your own script). To prevent the need to edit your default.po(t) file every time you run the i18n console task, you can use a different domain such as:

```
1.  function invalidate($field, $value = true) {
```

```
2.            return parent::invalidate($field, __d('validation_errors', $value, true));
3.        }
```

This will look for `$value` in the `validation_errors.po` file.

There's one other aspect of localizing your application which is not covered by the use of the translate functions, and that is date/money formats. Don't forget that CakePHP is PHP :), therefore to set the formats for these things you need to use `setlocale`.

If you pass a locale that doesn't exist on your computer to `setlocale` it will have no effect. You can find the list of available locales by running the command `$locale -a` in a terminal.

4.9 Pagination

One of the main obstacles of creating flexible and user-friendly web applications is designing an intuitive UI. Many applications tend to grow in size and complexity quickly, and designers and programmers alike find they are unable to cope with displaying hundreds or thousands of records. Refactoring takes time, and performance and user satisfaction can suffer.

Displaying a reasonable number of records per page has always been a critical part of every application and used to cause many headaches for developers. CakePHP eases the burden on the developer by providing a quick, easy way to paginate data.

The PaginatorHelper offers a great solution because it's so easy to use. Apart from pagination, it bundles some very easy-to-use sorting features. Last but not least, Ajax sorting and pagination are supported as well.

4.9.1 Controller Setup

In the controller, we start by defining the pagination defaults in the *$paginate* controller variable. It is important to note here that the order key must be defined in the array structure given.

```
1.    class RecipesController extends AppController {
2.        var $paginate = array(
```

```
3.                 'limit' => 25,
4.                 'order' => array(
5.                     'Post.title' => 'asc'
6.                 )
7.             );
8.         }
```

You can also include other find() options, such as *fields*:

```
1.         class RecipesController extends AppController {
2.             var $paginate = array(
3.                 'fields' => array('Post.id', 'Post.created'),
4.                 'limit' => 25,
5.                 'order' => array(
6.                     'Post.title' => 'asc'
7.                 )
8.             );
9.         }
```

Other keys that can be included in the *$paginate* array are similar to the parameters of the *Model->find('all')* method, that is: *conditions*, *fields*, *order*, *limit*, *page*, *contain*, *joins*, and *recursive*. In fact, you can define more than one set of pagination defaults in the controller, you just name the pieces of the array after the model you wish to configure:

```
1.         class RecipesController extends AppController {
2.             var $paginate = array(
3.                 'Recipe' => array (...),
4.                 'Author' => array (...)
5.             );
6.         }
```

Example of syntax using Containable Behavior:

```
1.      class RecipesController extends AppController {
2.          var $paginate = array(
3.              'limit' => 25,
4.              'contain' => array('Article')
5.          );
6.      }
```

Once the *$paginate* variable has been defined, we can call the *paginate()* method in controller actions. This method returns paged *find()* results from the model, and grabs some additional paging statistics, which are passed to the View behind the scenes. This method also adds PaginatorHelper to the list of helpers in your controller, if it has not been added already.

```
1.      function list_recipes() {
2.          // similar to findAll(), but fetches paged results
3.          $data = $this->paginate('Recipe');
4.          $this->set('data', $data);
5.      }
```

You can filter the records by passing conditions as second parameter to the `paginate()` function.

```
1.      $data = $this->paginate('Recipe', array('Recipe.title LIKE' => 'a%'));
```

Or you can also set *conditions* and other keys in the `$paginate` array inside your action.

```
1.      function list_recipes() {
2.          $this->paginate = array(
3.              'conditions' => array('Recipe.title LIKE' => 'a%'),
4.              'limit' => 10
5.          );
6.          $data = $this->paginate('Recipe');
7.          $this->set(compact('data'));
```

```
8.      );
```

4.9.2 Pagination in Views

It's up to you to decide how to show records to the user, but most often this will be done inside HTML tables. The examples below assume a tabular layout, but the PaginatorHelper available in views doesn't always need to be restricted as such.

See the details on PaginatorHelper in the API.

As mentioned, the PaginatorHelper also offers sorting features which can be easily integrated into your table column headers:

```
1.      // app/views/recipes/list_recipes.ctp
2.      <table>
3.          <tr>
4.              <th><?php echo $this->Paginator->sort('ID', 'id'); ?></th>
5.              <th><?php echo $this->Paginator->sort('Title', 'title'); ?></th>
6.          </tr>
7.             <?php foreach($data as $recipe): ?>
8.          <tr>
9.              <td><?php echo $recipe['Recipe']['id']; ?> </td>
10.             <td><?php echo $recipe['Recipe']['title']; ?> </td>
11.         </tr>
12.         <?php endforeach; ?>
13.     </table>
```

The links output from the sort() method of the PaginatorHelper allow users to click on table headers to toggle the sorting of the data by a given field.

It is also possible to sort a column based on associations:

```
1.      <table>
2.          <tr>
```

```
            <th><?php echo $this->Paginator->sort('Title', 'title'); ?></th>
            <th><?php echo $this->Paginator->sort('Author', 'Author.name'); ?></th>
        </tr>
        <?php foreach($data as $recipe): ?>
        <tr>
            <td><?php echo $recipe['Recipe']['title']; ?> </td>
            <td><?php echo $recipe['Author']['name']; ?> </td>
        </tr>
        <?php endforeach; ?>
    </table>
```

The final ingredient to pagination display in views is the addition of page navigation, also supplied by the PaginationHelper.

```
    <!-- Shows the page numbers -->
    <?php echo $this->Paginator->numbers(); ?>
    <!-- Shows the next and previous links -->
    <?php echo $this->Paginator->prev('« Previous', null, null, array('class' => 'disabled')); ?>
    <?php echo $this->Paginator->next('Next »', null, null, array('class' => 'disabled')); ?>
    <!-- prints X of Y, where X is current page and Y is number of pages -->
    <?php echo $this->Paginator->counter(); ?>
```

The wording output by the counter() method can also be customized using special markers:

```
    <?php
    echo $this->Paginator->counter(array(
        'format' => 'Page %page% of %pages%, showing %current% records out of
                %count% total, starting on record %start%, ending on %end%'
    ));
    ?>
```

To pass all URL arguments to paginator functions, add the following to your view:

```
1.      $this->Paginator->options(array('url' => $this->passedArgs));
```

Route elements that are not named arguments should manually be merged with `$this->passedArgs`:

```
1.      //for urls like http://www.example.com/en/controller/action
2.      //that are routed as Router::connect('/:lang/:controller/:action/*', array(), array('lang' => 'ta|en'));
3.      $this->Paginator->options(array('url' => array_merge(array('lang' => $lang), $this->passedArgs)));
```

Or you can specify which params to pass manually:

```
1.      $this->Paginator->options(array('url' => array("0", "1")));
```

4.9.3 AJAX Pagination

It's very easy to incorporate Ajax functionality into pagination. Using the JsHelper and RequestHandlerComponent you can easily add Ajax pagination to your application. See here for more information on Ajax pagination

Configuring the PaginatorHelper to use a custom helper

By default in 1.3 the `PaginatorHelper` uses JsHelper to do ajax features. However, if you don't want that and want to use the `AjaxHelper` or a custom helper for ajax links, you can do so by changing the `$helpers` array in your controller. After running `paginate()` do the following.

```
1.      $this->set('posts', $this->paginate());
2.      $this->helpers['Paginator'] = array('ajax' => 'Ajax');
```

Will change the `PaginatorHelper` to use the `AjaxHelper` for ajax operations. You could also set the 'ajax' key to be any helper, as long as that class implements a `link()` method that behaves like `HtmlHelper::link()`

4.9.4 Custom Query Pagination

Fix me: Please add an example where overriding paginate is justified

A good example of when you would need this is if the underlying DB does not support the SQL LIMIT syntax. This is true of IBM's DB2. You can still use the CakePHP pagination by adding the custom query to the model.

Should you need to create custom queries to generate the data you want to paginate, you can override the `paginate()` and `paginateCount()` model methods used by the pagination controller logic.

Before continuing check you can't achieve your goal with the core model methods.

The `paginate()` method uses the same parameters as `Model::find()`. To use your own method/logic override it in the model you wish to get the data from.

```
1.      /**
2.       * Overridden paginate method - group by week, away_team_id and home_team_id
3.       */
4.      function paginate($conditions, $fields, $order, $limit, $page = 1, $recursive = null, $extra = array()) {
5.          $recursive = -1;
6.          $group = $fields = array('week', 'away_team_id', 'home_team_id');
7.           return  $this->find('all',  compact('conditions',  'fields',  'order',  'limit',  'page',  'recursive',
'group'));
8.      }
```

You also need to override the core `paginateCount()`, this method expects the same arguments as `Model::find('count')`. The example below uses some Postgres-specifc features, so please adjust accordingly depending on what database you are using.

```
1.      /**
```

```
2.          * Overridden paginateCount method
3.          */
4.         function paginateCount($conditions = null, $recursive = 0, $extra = array()) {
5.             $sql = "SELECT DISTINCT ON(week, home_team_id, away_team_id) week, home_team_id, away_team_id FROM games";
6.             $this->recursive = $recursive;
7.             $results = $this->query($sql);
8.             return count($results);
9.         }
```

The observant reader will have noticed that the paginate method we've defined wasn't actually necessary - All you have to do is add the keyword in controller's `$paginate` class variable.

```
1.         /**
2.         * Add GROUP BY clause
3.         */
4.         var $paginate = array(
5.             'MyModel' => array('limit' => 20,
6.                                'order' => array('week' => 'desc'),
7.                                'group' => array('week', 'home_team_id', 'away_team_id'))
8.                               );
9.         /**
10.        * Or on-the-fly from within the action
11.        */
12.        function index() {
13.            $this->paginate = array(
14.            'MyModel' => array('limit' => 20,
15.                               'order' => array('week' => 'desc'),
16.                               'group' => array('week', 'home_team_id', 'away_team_id'))
17.                              );
```

However, it will still be necessary to override the `paginateCount()` method to get an accurate value.

4.10 REST

Many newer application programmers are realizing the need to open their core functionality to a greater audience. Providing easy, unfettered access to your core API can help get your platform accepted, and allows for mashups and easy integration with other systems.

While other solutions exist, REST is a great way to provide easy access to the logic you've created in your application. It's simple, usually XML-based (we're talking simple XML, nothing like a SOAP envelope), and depends on HTTP headers for direction. Exposing an API via REST in CakePHP is simple.

4.10.1 The Simple Setup

The fastest way to get up and running with REST is to add a few lines to your routes.php file, found in app/config. The Router object features a method called mapResources(), that is used to set up a number of default routes for REST access to your controllers. If we wanted to allow REST access to a recipe database, we'd do something like this:

```
1.      //In app/config/routes.php...
2.
3.      Router::mapResources('recipes');
4.      Router::parseExtensions();
```

The first line sets up a number of default routes for easy REST access where method specifies the desired result format (e.g. xml, json, rss). These routes are HTTP Request Method sensitive.

HTTP Method	URL.*method*	Controller action invoked
GET	/recipes.*method*	RecipesController::index()
GET	/recipes/*123.method*	RecipesController::view(123)
POST	/recipes.*method*	RecipesController::add()

PUT	/recipes/123.*method*	RecipesController::edit(123)
DELETE	/recipes/*123.method*	RecipesController::delete(123)
POST	/recipes/123.*method*	RecipesController::edit(123)

CakePHP's Router class uses a number of different indicators to detect the HTTP method being used. Here they are in order of preference:

1. The *_method* POST variable
2. The X_HTTP_METHOD_OVERRIDE
3. The REQUEST_METHOD header

The *_method* POST variable is helpful in using a browser as a REST client (or anything else that can do POST easily). Just set the value of _method to the name of the HTTP request method you wish to emulate.

Once the router has been set up to map REST requests to certain controller actions, we can move on to creating the logic in our controller actions. A basic controller might look something like this:

```
1.      // controllers/recipes_controller.php
2.      class RecipesController extends AppController {
3.          var $components = array('RequestHandler');
4.          function index() {
5.              $recipes = $this->Recipe->find('all');
6.              $this->set(compact('recipes'));
7.          }
8.          function view($id) {
9.              $recipe = $this->Recipe->findById($id);
10.             $this->set(compact('recipe'));
11.         }
12.         function edit($id) {
13.             $this->Recipe->id = $id;
```

```
            if ($this->Recipe->save($this->data)) {
                $message = 'Saved';
            } else {
                $message = 'Error';
            }
            $this->set(compact("message"));
        }
    function delete($id) {
            if($this->Recipe->delete($id)) {
                $message = 'Deleted';
            } else {
                $message = 'Error';
            }
            $this->set(compact("message"));
    }
}
```

Since we've added a call to Router::parseExtensions(), the CakePHP router is already primed to serve up different views based on different kinds of requests. Since we're dealing with REST requests, the view type is XML. We place the REST views for our RecipesController inside app/views/recipes/xml. We can also use the XmlHelper for quick-and-easy XML output in those views. Here's what our index view might look like:

```
// app/views/recipes/xml/index.ctp
<recipes>
    <?php echo $xml->serialize($recipes); ?>
</recipes>
```

Experienced CakePHP users might notice that we haven't included the XmlHelper in our RecipesController $helpers array. This is on purpose - when serving up a specific content type using parseExtensions(), CakePHP automatically looks for a view helper that matches the type. Since we're using XML as the content type, the XmlHelper is automatically loaded up for our use in those views.

The rendered XML will end up looking something like this:

```
1.      <posts>
2.          <post id="234" created="2008-06-13" modified="2008-06-14">
3.              <author id="23423" first_name="Billy" last_name="Bob"></author>
4.              <comment id="245" body="This is a comment for this post."></comment>
5.          </post>
6.          <post id="3247" created="2008-06-15" modified="2008-06-15">
7.              <author id="625" first_name="Nate" last_name="Johnson"></author>
8.              <comment id="654" body="This is a comment for this post."></comment>
9.          </post>
10.     </posts>
```

Creating the logic for the edit action is a bit trickier, but not by much. Since you're providing an API that outputs XML, it's a natural choice to receive XML as input. Not to worry, however: the RequestHandler and Router classes make things much easier. If a POST or PUT request has an XML content-type, then the input is taken and passed to an instance of Cake's Xml object, which is assigned to the $data property of the controller. Because of this feature, handling XML and POST data in parallel is seamless: no changes are required to the controller or model code. Everything you need should end up in $this->data.

A commonly-required serialization format is JSON, which would be requested by using the ".json" extension in paths. Cake will automatically attempt to find /views/layouts/json/default.ctp and /views/[object]/json/[action].ctp which are not provided by default. You will need to create these to accomodate your API's specific needs. Additionally, you will need to parse any JSON sent to the controller into the $this->data property. While this is not built in to Cake, the Cake developer community has quite a bit of sample code out there that should get you started.

4.10.2 Custom REST Routing

If the default routes created by mapResources() don't work for you, use the Router::connect() method to define a custom set of REST routes. The connect() method allows you to define a number of different options for a given URL. The first parameter is the URL itself, and the second parameter allows you to supply those options. The third parameter allows you to specify regex patterns to help CakePHP identify certain markers in the specified URL.

We'll provide a simple example here, and allow you to tailor this route for your other RESTful purposes. Here's what our edit REST route would look like, without using mapResources():

```
1.      Router::connect(
2.          "/:controller/:id",
3.          array("action" => "edit", "[method]" => "PUT"),
4.          array("id" => "[0-9]+")
5.      )
```

Advanced routing techniques are covered elsewhere, so we'll focus on the most important point for our purposes here: the [method] key of the options array in the second parameter. Once that key has been set, the specified route works only for that HTTP request method (which could also be GET, DELETE, etc.)

5 Core Components

CakePHP has a number of built-in components. They provide out of the box functionality for several commonly used tasks.

Acl	The Acl component provides an easy to use interface for database and ini based access control lists.
Auth	The auth component provides an easy to use authentication system using a variety of authentication processes, such as controller callbacks, Acl, or Object callbacks.
Cookie	The cookie component behaves in a similar fashion to the SessionComponent in that it provides a wrapper for PHP's native cookie support.
Email	An interface that can be used to send emails using one of several mail transfer agents including php's mail() and smtp.
RequestHandler	The request handler allows you to introspect further into the requests your visitors and inform your application about the content types and requested information.
Security	The security component allows you to set tighter security and use and manage HTTP authentication.

Session	The session component provides a storage independent wrapper to PHP's sessions.

To learn more about each component see the menu on the left, or learn more about creating your own components.

All core components now can be configured in the `$components` array of a controller.

```
<?php
class AppController extends Controller {
    var $components = array(
        'Auth' => array(
            'loginAction' => array('controller' => 'users', 'action' => 'signOn'),
            'fields' => array('username' => 'email', 'password' => 'password'),
        ),
        'Security',
        'Email' => array(
            'from' => 'webmaster@domain.com',
            'sendAs' => 'html',
        ),
    );
}
```

You can override the settings in the controller's `beforeFilter()`

```
<?php
class MembersController extends AppController {
    function beforeFilter() {
        $this->Email->from = 'support@domain.com';
    }
}
```

5.1 Access Control Lists

CakePHP's access control list functionality is one of the most oft-discussed, most likely because it is the most sought after, but also because it can be the most confusing. If you're looking for a good way to get started with ACLs in general, read on.

Be brave and stick with it, even if the going gets rough. Once you get the hang of it, it's an extremely powerful tool to have on hand when developing your application.

5.1.1 Understanding How ACL Works

Powerful things require access control. Access control lists are a way to manage application permissions in a fine-grained, yet easily maintainable and manageable way.

Access control lists, or ACL, handle two main things: things that want stuff, and things that are wanted. In ACL lingo, things (most often users) that want to use stuff are called access request objects, or AROs. Things in the system that are wanted (most often actions or data) are called access control objects, or ACOs. The entities are called 'objects' because sometimes the requesting object isn't a person - sometimes you might want to limit the access certain Cake controllers have to initiate logic in other parts of your application. ACOs could be anything you want to control, from a controller action, to a web service, to a line on your grandma's online diary.

To review:

- ACO - Access Control Object - Something that is wanted
- ARO - Access Request Object - Something that wants something

Essentially, ACL is what is used to decide when an ARO can have access to an ACO.

In order to help you understand how everything works together, let's use a semi-practical example. Imagine, for a moment, a computer system used by a familiar group of fantasy novel adventurers from the *Lord of the Rings*. The leader of the group, Gandalf, wants to manage the party's assets while maintaining a healthy amount of privacy and security for the other members of the party. The first thing he needs to do is create a list of the AROs involved:

- Gandalf

- Aragorn
- Bilbo
- Frodo
- Gollum
- Legolas
- Gimli
- Pippin
- Merry

Realize that ACL is *not* the same as authentication. ACL is what happens *after* a user has been authenticated. Although the two are usually used in concert, it's important to realize the difference between knowing who someone is (authentication) and knowing what they can do (ACL).

The next thing Gandalf needs to do is make an initial list of things, or ACOs, the system will handle. His list might look something like:

- Weapons
- The One Ring
- Salted Pork
- Diplomacy
- Ale

Traditionally, systems were managed using a sort of matrix, that showed a basic set of users and permissions relating to objects. If this information were stored in a table, it might look like the following table:

	Weapons	**The Ring**	**Salted Pork**	**Diplomacy**	**Ale**
Gandalf			Allow	Allow	Allow
Aragorn	Allow		Allow	Allow	Allow

Bilbo					Allow
Frodo		Allow			Allow
Gollum			Allow		
Legolas	Allow		Allow	Allow	Allow
Gimli	Allow		Allow		
Pippin				Allow	Allow
Merry					Allow

At first glance, it seems that this sort of system could work rather well. Assignments can be made to protect security (only Frodo can access the ring) and protect against accidents (keeping the hobbits out of the salted pork and weapons). It seems fine grained enough, and easy enough to read, right?

For a small system like this, maybe a matrix setup would work. But for a growing system, or a system with a large amount of resources (ACOs) and users (AROs), a table can become unwieldy rather quickly. Imagine trying to control access to the hundreds of war encampments and trying to manage them by unit. Another drawback to matrices is that you can't really logically group sections of users or make cascading permissions changes to groups of users based on those logical groupings. For example, it would sure be nice to automatically allow the hobbits access to the ale and pork once the battle is over: Doing it on an individual user basis would be tedious and error prone. Making a cascading permissions change to all 'hobbits' would be easy.

ACL is most usually implemented in a tree structure. There is usually a tree of AROs and a tree of ACOs. By organizing your objects in trees, permissions can still be dealt out in a granular fashion, while still maintaining a good grip on the big picture. Being the wise leader he is, Gandalf elects to use ACL in his new system, and organizes his objects along the following lines:

- Fellowship of the Ring™
    - Warriors
        - Aragorn
        - Legolas

        - Gimli
    - Wizards
        - Gandalf
    - Hobbits
        - Frodo
        - Bilbo
        - Merry
        - Pippin
    - Visitors
        - Gollum

Using a tree structure for AROs allows Gandalf to define permissions that apply to entire groups of users at once. So, using our ARO tree, Gandalf can tack on a few group-based permissions:

- Fellowship of the Ring
  (**Deny**: all)
    - Warriors
      (**Allow**: Weapons, Ale, Elven Rations, Salted Pork)
        - Aragorn
        - Legolas
        - Gimli
    - Wizards
      (**Allow**: Salted Pork, Diplomacy, Ale)
        - Gandalf
    - Hobbits
      (**Allow**: Ale)
        - Frodo
        - Bilbo
        - Merry

        - Pippin
    - Visitors

      (**Allow**: Salted Pork)
        - Gollum

If we wanted to use ACL to see if the Pippin was allowed to access the ale, we'd first get his path in the tree, which is Fellowship->Hobbits->Pippin. Then we see the different permissions that reside at each of those points, and use the most specific permission relating to Pippin and the Ale.

ARO Node	Permission Info	Result
Fellowship of the Ring	Deny all	Denying access to ale.
Hobbits	Allow 'ale'	Allowing access to ale!
Pippin	--	Still allowing ale!

Since the 'Pippin' node in the ACL tree doesn't specifically deny access to the ale ACO, the final result is that we allow access to that ACO.

The tree also allows us to make finer adjustments for more granular control - while still keeping the ability to make sweeping changes to groups of AROs:

- Fellowship of the Ring

  (**Deny**: all)
    - Warriors

      (**Allow**: Weapons, Ale, Elven Rations, Salted Pork)
        - Aragorn

          (Allow: Diplomacy)
        - Legolas
        - Gimli
    - Wizards

      (**Allow**: Salted Pork, Diplomacy, Ale)

    - Gandalf
- Hobbits

  (**Allow**: Ale)
    - Frodo

      (Allow: Ring)
    - Bilbo
    - Merry

      (Deny: Ale)
    - Pippin

      (Allow: Diplomacy)
- Visitors

  (**Allow**: Salted Pork)
    - Gollum

This approach allows us both the ability to make wide-reaching permissions changes, but also fine-grained adjustments. This allows us to say that all hobbits can have access to ale, with one exception—Merry. To see if Merry can access the Ale, we'd find his path in the tree: Fellowship->Hobbits->Merry and work our way down, keeping track of ale-related permissions:

ARO Node	Permission Info	Result
Fellowship of the Ring	Deny all	Denying access to ale.
Hobbits	Allow 'ale'	Allowing access to ale!
Merry	Deny 'ale'	Denying ale.

5.1.2 Defining Permissions: Cake's INI-based ACL

Cake's first ACL implementation was based on INI files stored in the Cake installation. While it's useful and stable, we recommend that you use the database backed ACL solution, mostly because of its ability to create new ACOs and AROs on the fly. We meant it for usage in simple applications - and especially for those folks who might not be using a database for some reason.

By default, CakePHP's ACL is database-driven. To enable INI-based ACL, you'll need to tell CakePHP what system you're using by updating the following lines in app/config/core.php

```
1.      //Change these lines:
2.      Configure::write('Acl.classname', 'DbAcl');
3.      Configure::write('Acl.database', 'default');
4.      //To look like this:
5.      Configure::write('Acl.classname', 'IniAcl');
6.      //Configure::write('Acl.database', 'default');
```

ARO/ACO permissions are specified in **/app/config/acl.ini.php**. The basic idea is that AROs are specified in an INI section that has three properties: groups, allow, and deny.

- groups: names of ARO groups this ARO is a member of.
- allow: names of ACOs this ARO has access to
- deny: names of ACOs this ARO should be denied access to

ACOs are specified in INI sections that only include the allow and deny properties.

As an example, let's see how the Fellowship ARO structure we've been crafting would look like in INI syntax:

```
;-------------------------------------
; AROs
;-------------------------------------
[aragorn]
groups = warriors
allow = diplomacy

[legolas]
groups = warriors

[gimli]
groups = warriors
```

```
[gandalf]
groups = wizards

[frodo]
groups = hobbits
allow = ring

[bilbo]
groups = hobbits

[merry]
groups = hobbits
deny = ale

[pippin]
groups = hobbits

[gollum]
groups = visitors

;-------------------------------------
; ARO Groups
;-------------------------------------
[warriors]
allow = weapons, ale, salted_pork

[wizards]
allow = salted_pork, diplomacy, ale

[hobbits]
allow = ale

[visitors]
allow = salted_pork
```

Now that you've got your permissions defined, you can skip along to the section on checking permissions using the ACL component.

5.1.3 Defining Permissions: Cake's Database ACL

Now that we've covered INI-based ACL permissions, let's move on to the (more commonly used) database ACL.

5.1.3.1 Getting Started

The default ACL permissions implementation is database powered. Cake's database ACL consists of a set of core models, and a console application that comes with your Cake installation. The models are used by Cake to interact with your database in order to store and retrieve nodes in tree format. The console application is used to initialize your database and interact with your ACO and ARO trees.

To get started, first you'll need to make sure your `/app/config/database.php` is present and correctly configured. See section 4.1 for more information on database configuration.

Once you've done that, use the CakePHP console to create your ACL database tables:

```
$ cake schema create DbAcl
```

Running this command will drop and re-create the tables necessary to store ACO and ARO information in tree format. The output of the console application should look something like the following:

```
---------------------------------------------------------------
Cake Schema Shell
---------------------------------------------------------------

The following tables will be dropped.
acos
aros
aros_acos

Are you sure you want to drop the tables? (y/n)
[n] > y
Dropping tables.
acos updated.
aros updated.
aros_acos updated.
```

```
The following tables will be created.
acos
aros
aros_acos

Are you sure you want to create the tables? (y/n)
[y] > y
Creating tables.
acos updated.
aros updated.
aros_acos updated.
End create.
```

This replaces an older deprecated command, "initdb".

You can also use the SQL file found in `app/config/sql/db_acl.sql`, but that's nowhere near as fun.

When finished, you should have three new database tables in your system: acos, aros, and aros_acos (the join table to create permissions information between the two trees).

If you're curious about how Cake stores tree information in these tables, read up on modified database tree traversal. The ACL component uses CakePHP's Tree Behavior to manage the trees' inheritances. The model class files for ACL are all compiled in a single file db_acl.php.

Now that we're all set up, let's work on creating some ARO and ACO trees.

5.1.3.2 Creating Access Request Objects (AROs) and Access Control Objects (ACOs)

In creating new ACL objects (ACOs and AROs), realize that there are two main ways to name and access nodes. The *first* method is to link an ACL object directly to a record in your database by specifying a model name and foreign key value. The *second* method can be used when an object has no direct relation to a record in your database - you can provide a textual alias for the object.

In general, when you're creating a group or higher level object, use an alias. If you're managing access to a specific item or record in the database, use the model/foreign key method.

You create new ACL objects using the core CakePHP ACL models. In doing so, there are a number of fields you'll want to use when saving data: `model`, `foreign_key`, `alias`, and `parent_id`.

The `model` and `foreign_key` fields for an ACL object allows you to link up the object to its corresponding model record (if there is one). For example, many AROs will have corresponding User records in the database. Setting an ARO's `foreign_key` to the User's ID will allow you to link up ARO and User information with a single User model find() call if you've set up the correct model associations. Conversely, if you want to manage edit operation on a specific blog post or recipe listing, you may choose to link an ACO to that specific model record.

The `alias` for an ACL object is just a human-readable label you can use to identify an ACL object that has no direct model record correlation. Aliases are usually useful in naming user groups or ACO collections.

The `parent_id` for an ACL object allows you to fill out the tree structure. Supply the ID of the parent node in the tree to create a new child.

Before we can create new ACL objects, we'll need to load up their respective classes. The easiest way to do this is to include Cake's ACL Component in your controller's $components array:

```
1.      var $components = array('Acl');
```

Once we've got that done, let's see what some examples of creating these objects might look like. The following code could be placed in a controller action somewhere:

While the examples here focus on ARO creation, the same techniques can be used to create an ACO tree.

Keeping with our Fellowship setup, let's first create our ARO groups. Because our groups won't really have specific records tied to them, we'll use aliases to create these ACL objects. What we're doing here is from the perspective of a controller action, but could be done elsewhere. What we'll cover here is a bit of an artificial approach, but you should feel comfortable using these techniques to build AROs and ACOs on the fly.

This shouldn't be anything drastically new - we're just using models to save data like we always do:

```
1.      function anyAction()
2.      {
3.          $aro =& $this->Acl->Aro;
4.
5.          //Here's all of our group info in an array we can iterate through
6.          $groups = array(
7.              0 => array(
8.                  'alias' => 'warriors'
9.              ),
10.             1 => array(
11.                 'alias' => 'wizards'
12.             ),
13.             2 => array(
14.                 'alias' => 'hobbits'
15.             ),
16.             3 => array(
17.                 'alias' => 'visitors'
18.             ),
19.         );
20.
21.         //Iterate and create ARO groups
22.         foreach($groups as $data)
23.         {
24.             //Remember to call create() when saving in loops...
25.             $aro->create();
26.
27.             //Save data
28.             $aro->save($data);
29.         }
```

```
30.             //Other action logic goes here...
31.         }
```

Once we've got them in there, we can use the ACL console application to verify the tree structure.

```
$ cake acl view aro

Aro tree:
---------------------------------------------------------------
  [1]warriors

  [2]wizards

  [3]hobbits

  [4]visitors

---------------------------------------------------------------
```

I suppose it's not much of a tree at this point, but at least we've got some verification that we've got four top-level nodes. Let's add some children to those ARO nodes by adding our specific user AROs under these groups. Every good citizen of Middle Earth has an account in our new system, so we'll tie these ARO records to specific model records in our database.

When adding child nodes to a tree, make sure to use the ACL node ID, rather than a foreign_key value.

```
1.          function anyAction()
2.          {
3.              $aro = new Aro();
4.
5.              //Here are our user records, ready to be linked up to new ARO records
6.              //This data could come from a model and modified, but we're using static
```

```
7.          //arrays here for demonstration purposes.
8.
9.          $users = array(
10.             0 => array(
11.                 'alias' => 'Aragorn',
12.                 'parent_id' => 1,
13.                 'model' => 'User',
14.                 'foreign_key' => 2356,
15.             ),
16.             1 => array(
17.                 'alias' => 'Legolas',
18.                 'parent_id' => 1,
19.                 'model' => 'User',
20.                 'foreign_key' => 6342,
21.             ),
22.             2 => array(
23.                 'alias' => 'Gimli',
24.                 'parent_id' => 1,
25.                 'model' => 'User',
26.                 'foreign_key' => 1564,
27.             ),
28.             3 => array(
29.                 'alias' => 'Gandalf',
30.                 'parent_id' => 2,
31.                 'model' => 'User',
32.                 'foreign_key' => 7419,
33.             ),
34.             4 => array(
35.                 'alias' => 'Frodo',
36.                 'parent_id' => 3,
37.                 'model' => 'User',
```

```
38.                 'foreign_key' => 7451,
39.             ),
40.             5 => array(
41.                 'alias' => 'Bilbo',
42.                 'parent_id' => 3,
43.                 'model' => 'User',
44.                 'foreign_key' => 5126,
45.             ),
46.             6 => array(
47.                 'alias' => 'Merry',
48.                 'parent_id' => 3,
49.                 'model' => 'User',
50.                 'foreign_key' => 5144,
51.             ),
52.             7 => array(
53.                 'alias' => 'Pippin',
54.                 'parent_id' => 3,
55.                 'model' => 'User',
56.                 'foreign_key' => 1211,
57.             ),
58.             8 => array(
59.                 'alias' => 'Gollum',
60.                 'parent_id' => 4,
61.                 'model' => 'User',
62.                 'foreign_key' => 1337,
63.             ),
64.         );
65.
66.         //Iterate and create AROs (as children)
67.         foreach($users as $data)
68.         {
```

```
69.                 //Remember to call create() when saving in loops...
70.                 $aro->create();
71.                 //Save data
72.                 $aro->save($data);
73.             }
74.
75.             //Other action logic goes here...
76.         }
```

Typically you won't supply both an alias and a model/foreign_key, but we're using both here to make the structure of the tree easier to read for demonstration purposes.

The output of that console application command should now be a little more interesting. Let's give it a try:

```
$ cake acl view aro

Aro tree:
---------------------------------------------------------------
  [1]warriors

    [5]Aragorn

    [6]Legolas

    [7]Gimli

  [2]wizards

    [8]Gandalf

  [3]hobbits
```

```
    [9]Frodo

    [10]Bilbo

    [11]Merry

    [12]Pippin

  [4]visitors

    [13]Gollum

---------------------------------------------------------------
```

Now that we've got our ARO tree setup properly, let's discuss a possible approach for structuring an ACO tree. While we can structure more of an abstract representation of our ACO's, it's often more practical to model an ACO tree after Cake's Controller/Action setup. We've got five main objects we're handling in this Fellowship scenario, and the natural setup for that in a Cake application is a group of models, and ultimately the controllers that manipulate them. Past the controllers themselves, we'll want to control access to specific actions in those controllers.

Based on that idea, let's set up an ACO tree that will mimic a Cake app setup. Since we have five ACOs, we'll create an ACO tree that should end up looking something like the following:

- Weapons
- Rings
- PorkChops
- DiplomaticEfforts
- Ales

One nice thing about a Cake ACL setup is that each ACO automatically contains four properties related to CRUD (create, read, update, and delete) actions. You can create children nodes under each of these five main ACOs, but using Cake's built in action management covers basic CRUD operations on a given object. Keeping this in mind will make your ACO trees smaller and easier to maintain. We'll see how these are used later on when we discuss how to assign permissions.

Since you're now a pro at adding AROs, use those same techniques to create this ACO tree. Create these upper level groups using the core Aco model.

5.1.3.3 Assigning Permissions

After creating our ACOs and AROs, we can finally assign permissions between the two groups. This is done using Cake's core Acl component. Let's continue on with our example.

Here we'll work in the context of a controller action. We do that because permissions are managed by the Acl Component.

```
1.      class SomethingsController extends AppController
2.      {
3.          // You might want to place this in the AppController
4.          // instead, but here works great too.
5.          var $components = array('Acl');
6.      }
```

Let's set up some basic permissions using the AclComponent in an action inside this controller.

```
1.      function index()
2.      {
3.          //Allow warriors complete access to weapons
4.          //Both these examples use the alias syntax
5.          $this->Acl->allow('warriors', 'Weapons');
6.
7.          //Though the King may not want to let everyone
8.          //have unfettered access
9.          $this->Acl->deny('warriors/Legolas', 'Weapons', 'delete');
10.         $this->Acl->deny('warriors/Gimli',   'Weapons', 'delete');
11.
12.         die(print_r('done', 1));
13.     }
```

The first call we make to the AclComponent allows any user under the 'warriors' ARO group full access to anything under the 'Weapons' ACO group. Here we're just addressing ACOs and AROs by their aliases.

Notice the usage of the third parameter? That's where we use those handy actions that are in-built for all Cake ACOs. The default options for that parameter are `create`, `read`, `update`, and `delete` but you can add a column in the `aros_acos` database table (prefixed with _ - for example `_admin`) and use it alongside the defaults.

The second set of calls is an attempt to make a more fine-grained permission decision. We want Aragorn to keep his full-access privileges, but deny other warriors in the group the ability to delete Weapons records. We're using the alias syntax to address the AROs above, but you might want to use the model/foriegn key syntax yourself. What we have above is equivalent to this:

```
1.      // 6342 = Legolas
2.      // 1564 = Gimli
3.      $this->Acl->deny(array('model' => 'User', 'foreign_key' => 6342), 'Weapons', 'delete');
4.      $this->Acl->deny(array('model' => 'User', 'foreign_key' => 1564), 'Weapons', 'delete');
```

Addressing a node using the alias syntax uses a slash-delimited string ('/users/employees/developers'). Addressing a node using model/foreign key syntax uses an array with two parameters: `array('model' => 'User', 'foreign_key' => 8282)`.

The next section will help us validate our setup by using the AclComponent to check the permissions we've just set up.

5.1.3.4 Checking Permissions: The ACL Component

Let's use the AclComponent to make sure dwarves and elves can't remove things from the armory. At this point, we should be able to use the AclComponent to make a check between the ACOs and AROs we've created. The basic syntax for making a permissions check is:

```
1.      $this->Acl->check( $aro, $aco, $action = '*');
```

Let's give it a try inside a controller action:

```
1.      function index()
2.      {
3.          //These all return true:
4.          $this->Acl->check('warriors/Aragorn', 'Weapons');
5.          $this->Acl->check('warriors/Aragorn', 'Weapons', 'create');
6.          $this->Acl->check('warriors/Aragorn', 'Weapons', 'read');
7.          $this->Acl->check('warriors/Aragorn', 'Weapons', 'update');
8.          $this->Acl->check('warriors/Aragorn', 'Weapons', 'delete');
9.
10.         //Remember, we can use the model/foreign key syntax
11.         //for our user AROs
12.         $this->Acl->check(array('model' => 'User', 'foreign_key' => 2356), 'Weapons');
13.
14.         //These also return true:
15.         $result = $this->Acl->check('warriors/Legolas', 'Weapons', 'create');
16.         $result = $this->Acl->check('warriors/Gimli', 'Weapons', 'read');
17.
18.         //But these return false:
19.         $result = $this->Acl->check('warriors/Legolas', 'Weapons', 'delete');
20.         $result = $this->Acl->check('warriors/Gimli', 'Weapons', 'delete');
21.     }
```

The usage here is demonstrational, but hopefully you can see how checking like this can be used to decide whether or not to allow something to happen, show an error message, or redirect the user to a login.

5.2 Authentication

User authentication systems are a common part of many web applications. In CakePHP there are several systems for authenticating users, each of which provides different options. At its core the authentication component will check to see if a user has an account with a site. If they do, the component will give access to that user to the complete site.

This component can be combined with the ACL (access control lists) component to create more complex levels of access within a site. The ACL Component, for example, could allow you to grant one user access to public site areas, while granting another user access to protected administrative portions of the site.

CakePHP's AuthComponent can be used to create such a system easily and quickly. Let's take a look at how you would build a very simple authentication system.

Like all components, you use it by adding 'Auth' to the list of components in your controller:

```
1.      class FooController extends AppController {
2.          var $components = array('Auth');
```

Or add it to your AppController so all of your controllers will use it:

```
1.      class AppController extends Controller {
2.          // AppController's components are NOT merged with defaults,
3.          // so session component is lost if it's not included here!
4.          var $components = array('Auth', 'Session');
```

Now, there are a few conventions to think about when using AuthComponent. By default, the AuthComponent expects you to have a table called 'users' with fields called 'username' and 'password' to be used.

In some situations, databases don't let you use 'password' as a column name. See Setting Auth Component Variables for an example how to change the default field names to work with your own environment.

Let's set up our users table using the following SQL:

```
1.      CREATE TABLE users (
2.          id integer auto_increment,
3.          username char(50),
```

```
4.            password char(40),
5.            PRIMARY KEY (id)
6.        );
```

Something to keep in mind when creating a table to store all your user authentication data is that the AuthComponent expects the password value stored in the database to be hashed instead of being stored in plaintext. Make sure that the field you will be using to store passwords is long enough to store the hash (40 characters for SHA1, for example).

If you want to add a user manually to the database, the simplest method to get the right data is to attempt to log in and look at the SQL log.

For the most basic setup, you'll only need to create two actions in your controller:

```
1.        class UsersController extends AppController {
2.            var $name = 'Users';
3.            var $components = array('Auth'); // Not necessary if declared in your app controller
4.
5.            /**
6.             *  The AuthComponent provides the needed functionality
7.             *  for login, so you can leave this function blank.
8.             */
9.            function login() {
10.           }
11.           function logout() {
12.               $this->redirect($this->Auth->logout());
13.           }
14.       }
```

While you can leave the login() function blank, you do need to create the login view template (saved in app/views/users/login.ctp). This is the only UsersController view template you need to create, however. The example below assumes you are already using the Form helper:

```
<?php
    echo $session->flash('auth');
    echo $this->Form->create('User', array('action' => 'login'));
    echo $this->Form->input('username');
    echo $this->Form->input('password');
    echo $this->Form->end('Login');
?>
```

This view creates a simple login form where you enter a username and password. Once you submit this form, the AuthComponent takes care of the rest for you. The session flash message will display any notices generated by the AuthComponent. Upon successful login the database record of the current logged in user is saved to session.

Believe it or not, we're done! That's how to implement an incredibly simple, database-driven authentication system using the Auth component. However, there is a lot more we can do. Let's take a look at some more advanced usage of the component.

5.2.1 Setting Auth Component Variables

Whenever you want to alter a default option for AuthComponent, you do that by creating a beforeFilter() method for your controller, and then calling various built-in methods or setting component variables.

For example, to change the field name used for passwords from 'password' to 'secretword', you would do the following:

```
class UsersController extends AppController {
    var $components = array('Auth');
    function beforeFilter() {
        $this->Auth->fields = array(
            'username' => 'username',
            'password' => 'secretword'
            );
    }
}
```

In this particular situation, you would also need to remember to change the field name in the view template!

Alternately, you can specify settings for Auth by placing them inside the controller's $components property.

```
1.      class AppController extends Controller {
2.          var $components = array(
3.              'Auth' => array(
4.                  'authorize' => 'actions',
5.                  'actionPath' => 'controllers/',
6.                  'loginAction' => array(
7.                      'controller' => 'users',
8.                      'action' => 'login',
9.                      'plugin' => false,
10.                    'admin' => false,
11.                     ),
12.                 ),
13.             'Acl',
14.             'Session',
15.             );
16.     }
```

Another common use of Auth component variables is to allow access to certain methods without the user being logged in (by default Auth restricts access to every action except the login and logout methods).

For example if we want to allow all users access to the index and view methods (but not any other), we would do the following:

```
1.      function beforeFilter() {
2.              $this->Auth->allow('index','view');
3.      }
```

5.2.2 Displaying Auth Error Messages

In order to display the error messages that Auth spits out you need to add the following code to your view. In this case, the message will appear below the regular flash messages:

In order to show all normal flash messages and auth flash messages for all views add the following two lines to the views/layouts/default.ctp file in the body section preferable before the content_for_layout line.

```
1.      <?php
2.          echo $this->Session->flash();
3.          echo $this->Session->flash('auth');
4.      ?>
```

To customize the Auth error messages, place the following code in the AppController or wherever you have placed Auth's settings:

<?php $this->Auth->loginError = "This message shows up when the wrong credentials are used"; $this->Auth->authError = "This error shows up with the user tries to access a part of the website that is protected."; ?>

```
1.      <?php
2.          $this->Auth->loginError = "This message shows up when the wrong credentials are used";
3.          $this->Auth->authError = "This error shows up with the user tries to access a part of the website that is protected.";
4.      ?>
```

5.2.3 Troubleshooting Auth Problems

It can sometimes be quite difficult to diagnose problems when it's not behaving as expected, so here are a few pointers to remember.

5.2.3.1 Password Hashing

The automatic hashing of your password input field happens **only** if posted data also contains username and password fields

When posting information to an action via a form, the Auth component automatically hashes the contents of your password input field if posted data also contains username field. So, if you are trying to create some sort of registration page, make sure to have the user fill out a 'confirm password' field so that you can compare the two. Here's some sample code:

```
<?php
function register() {
    if ($this->data) {
        if ($this->data['User']['password'] == $this->Auth->password($this->data['User']['password_confirm'])) {
            $this->User->create();
            $this->User->save($this->data);
        }
    }
}
?>
```

5.2.4 Change Hash Function

The AuthComponent uses the Security class to hash a password. The Security class uses the SHA1 scheme by default. To change another hash function used by the Auth component, use the `setHash` method passing it `md5`, `sha1` or `sha256` as its first and only parameter.

```
Security::setHash('md5'); // or sha1 or sha256.
```

The Security class uses a salt value (set in `/app/config/core.php`) to hash the password.

If you want to use different password hashing logic beyond md5/sha1 with the application salt, you will need to override the standard hashPassword mechanism - You may need to do this if for example you have an existing database that previously used a hashing scheme without a salt. To do this, create the method `hashPasswords` in the class you want to be responsible for hashing your passwords (usually the User model) and set `authenticate` to the object you're authenticating against (usually this is User) like so:

```
1.      function beforeFilter() {
2.         $this->Auth->authenticate = ClassRegistry::init('User');
3.         ...
4.         parent::beforeFilter();
5.      }
```

With the above code, the User model hashPasswords() method will be called each time Cake calls AuthComponent::hashPasswords(). Here's an example hashPassword function, appropriate if you've already got a users table full of plain md5-hashed passwords:

```
1.      class User extends AppModel {
2.          function hashPasswords($data) {
3.              if (isset($data['User']['password'])) {
4.                  $data['User']['password'] = md5($data['User']['password']);
5.                  return $data;
6.              }
7.              return $data;
8.          }
9.      }
```

5.2.5 AuthComponent Methods

5.2.5.1 action

```
action (string $action = ':controller/:action')
```

If you are using ACO's as part of your ACL structure, you can get the path to the ACO node bound to a particular controller/action pair:

```
1.          $acoNode = $this->Auth->action('users/delete');
```

If you don't pass in any values, it uses the current controller / action pair

5.2.5.2 allow

If you have some actions in your controller that you don't have to authenticate against (such as a user registration action), you can add methods that the AuthComponent should ignore. The following example shows how to allow an action named 'register'.

```
1.  function beforeFilter() {
2.      ...
3.      $this->Auth->allow('register');
4.  }
```

If you wish to allow multiple actions to skip authentication, you supply them as parameters to the allow() method:

```
1.  function beforeFilter() {
2.      ...
3.      $this->Auth->allow('foo', 'bar', 'baz');
4.  }
```

Shortcut: you may also allow all the actions in a controller by using '*'.

```
1.  function beforeFilter() {
2.      ...
3.      $this->Auth->allow('*');
4.  }
```

If you are using requestAction in your layout or elements you should allow those actions in order to be able to open login page properly.

The auth component assumes that your actions names follow conventions and are underscored.

5.2.5.3 deny

There may be times where you will want to remove actions from the list of allowed actions (set using $this->Auth->allow()). Here's an example:

```
function beforeFilter() {
    $this->Auth->authorize = 'controller';
    $this->Auth->allow('delete');
}
function isAuthorized() {
    if ($this->Auth->user('role') != 'admin') {
        $this->Auth->deny('delete');
    }
    ...
}
```

5.2.5.4 hashPasswords

hashPasswords ($data)

This method checks if the `$data` contains the username and password fields as specified by the variable `$fields` indexed by the model name as specified by `$userModel`. If the `$data` array contains both the username and password, it hashes the password field in the array and returns the `data` array in the same format. This function should be used prior to insert or update calls of the user when the password field is affected.

```
$data['User']['username'] = 'me@me.com';
$data['User']['password'] = 'changeme';
$hashedPasswords = $this->Auth->hashPasswords($data);
pr($hashedPasswords);
/* returns:
Array
(
    [User] => Array
    (
        [username] => me@me.com
        [password] => 8ed3b7e8ced419a679a7df93eff22fae
    )
```

```
13.             )
14.             */
```

The *$hashedPasswords['User']['password']* field would now be hashed using the `password` function of the component.

If your controller uses the Auth component and posted data contains the fields as explained above, it will automatically hash the password field using this function.

5.2.5.5 mapActions

If you are using Acl in CRUD mode, you may want to assign certain non-default actions to each part of CRUD.

```
1.         $this->Auth->mapActions(
2.             array(
3.                 'create' => array('someAction'),
4.                 'read' => array('someAction', 'someAction2'),
5.                 'update' => array('someAction'),
6.                 'delete' => array('someAction')
7.             )
8.         );
```

5.2.5.6 login

```
login($data = null)
```

If you are doing some sort of Ajax-based login, you can use this method to manually log someone into the system. If you don't pass any value for `$data`, it will automatically use POST data passed into the controller.

for example, in an application you may wish to assign a user a password and auto log them in after registration. In an over simplified example:

View:

```
1.      echo $this->Form->create('User',array('action'=>'register'));
2.      echo $this->Form->input('username');
3.      echo $this->Form->end('Register');
```

Controller:

```
1.      function register() {
2.          if(!empty($this->data)) {
3.              $this->User->create();
4.              $assigned_password = 'password';
5.              $this->data['User']['password'] = $assigned_password;
6.              if($this->User->save($this->data)) {
7.                  // send signup email containing password to the user
8.                  $this->Auth->login($this->data);
9.                  $this->redirect('home');
10.         }
11.     }
```

One thing to note is that you must manually redirect the user after login as loginRedirect is not called.

`$this->Auth->login($data)` returns 1 on successful login, 0 on a failure

5.2.5.7 logout

Provides a quick way to de-authenticate someone, and redirect them to where they need to go. This method is also useful if you want to provide a 'Log me out' link inside a members' area of your application.

Example:

```
1.      $this->redirect($this->Auth->logout());
```

5.2.5.8 password

password (string $password)

Pass in a string, and you can get what the hashed password would look like. This is an essential functionality if you are creating a user registration screen where you have users enter their password a second time to confirm it.

```
1.      if ($this->data['User']['password'] ==
2.          $this->Auth->password($this->data['User']['password2'])) {
3.          // Passwords match, continue processing
4.          ...
5.      } else {
6.          $this->flash('Typed passwords did not match', 'users/register');
7.      }
```

The auth component will automatically hash the password field if the username field is also present in the submitted data

Cake appends your password string to a salt value and then hashes it. The hashing function used depends on the one set by the core utility class `Security` (sha1 by default). You can use the `Security::setHash` function to change the hashing method. The salt value is used from your application's configuration defined in your `core.php`

5.2.5.9 user

user(string $key = null)

This method provides information about the currently authenticated user. The information is taken from the session. For example:

```
1.      if ($this->Auth->user('role') == 'admin') {
2.          $this->flash('You have admin access');
3.      }
```

It can also be used to return the whole user session data like so:

```
1.        $data['User'] = $this->Auth->user();
```

If this method returns null, the user is not logged in.

In the view you can use the Session helper to retrieve the currently authenticated user's information:

```
1.        $session->read('Auth.User'); // returns complete user record
2.        $session->read('Auth.User.first_name') //returns particular field value
```

The session key can be different depending on which model Auth is configured to use. Eg. If you use model `Account` instead of `User`, then the session key would be `Auth.Account`

5.2.6 AuthComponent Variables

Now, there are several Auth-related variables that you can use as well. Usually you add these settings in your Controller's beforeFilter() method. Or, if you need to apply such settings site-wide, you would add them to App Controller's beforeFilter()

5.2.6.1 userModel

Don't want to use a User model to authenticate against? No problem, just change it by setting this value to the name of the model you want to use.

```
1.        <?php
2.            $this->Auth->userModel = 'Member';
3.        ?>
```

5.2.6.2 fields

Overrides the default username and password fields used for authentication.

```
<?php
    $this->Auth->fields = array('username' => 'email', 'password' => 'passwd');
?>
```

5.2.6.3 userScope

Use this to provide additional requirements for authentication to succeed.

```
<?php
    $this->Auth->userScope = array('User.active' => true);
?>
```

5.2.6.4 loginAction

You can change the default login from */users/login* to be any action of your choice.

```
<?php
    $this->Auth->loginAction = array('admin' => false, 'controller' => 'members', 'action' => 'login');
?>
```

5.2.6.5 loginRedirect

The AuthComponent remembers what controller/action pair you were trying to get to before you were asked to authenticate yourself by storing this value in the Session, under the `Auth.redirect` key. However, if this session value is not set (if you're coming to the login page from an external link, for example), then the user will be redirected to the URL specified in loginRedirect.

Example:

```
<?php
    $this->Auth->loginRedirect = array('controller' => 'members', 'action' => 'home');
?>
```

5.2.6.6 logoutRedirect

You can also specify where you want the user to go after they are logged out, with the default being the login action.

```
1.      <?php
2.          $this->Auth->logoutRedirect   =   array(Configure::read('Routing.admin')   =>   false,   'controller'   =>
    'members', 'action' => 'logout');
3.      ?>
```

5.2.6.7 loginError

Change the default error message displayed when someone does not successfully log in.

```
1.      <?php
2.          $this->Auth->loginError = "No, you fool!  That's not the right password!";
3.      ?>
```

5.2.6.8 authError

Change the default error message displayed when someone attempts to access an object or action to which they do not have access.

```
1.      <?php
2.          $this->Auth->authError = "Sorry, you are lacking access.";
3.      ?>
```

5.2.6.9 autoRedirect

Normally, the AuthComponent will automatically redirect you as soon as it authenticates. Sometimes you want to do some more checking before you redirect users:

```
1.      <?php
2.          function beforeFilter() {
3.              ...
4.              $this->Auth->autoRedirect = false;
```

```
5.          }
6.          ...
7.          function login() {
8.          //-- code inside this function will execute only when autoRedirect was set to false (i.e. in a
   beforeFilter).
9.              if ($this->Auth->user()) {
10.                 if (!empty($this->data['User']['remember_me'])) {
11.                     $cookie = array();
12.                     $cookie['username'] = $this->data['User']['username'];
13.                     $cookie['password'] = $this->data['User']['password'];
14.                     $this->Cookie->write('Auth.User', $cookie, true, '+2 weeks');
15.                     unset($this->data['User']['remember_me']);
16.                 }
17.                 $this->redirect($this->Auth->redirect());
18.             }
19.             if (empty($this->data)) {
20.                 $cookie = $this->Cookie->read('Auth.User');
21.                 if (!is_null($cookie)) {
22.                     if ($this->Auth->login($cookie)) {
23.                         //  Clear auth message, just in case we use it.
24.                         $this->Session->delete('Message.auth');
25.                         $this->redirect($this->Auth->redirect());
26.                     }
27.                 }
28.             }
29.         }
30.     ?>
```

The code in the login function will not execute *unless* you set $autoRedirect to false in a beforeFilter. The code present in the login function will only execute *after* authentication was attempted. This is the best place to determine whether or not a successful login occurred by the AuthComponent (should you desire to log the last successful login timestamp, etc.).

With autoRedirect set to false, you can also inject additional code such as keeping track of the last successful login timestamp

```
1.      <?php
2.          function login() {
3.              if( !(empty($this->data)) && $this->Auth->user() ){
4.                  $this->User->id = $this->Auth->user('id');
5.                  $this->User->saveField('last_login', date('Y-m-d H:i:s') );
6.                  $this->redirect($this->Auth->redirect());
7.              }
8.          }
9.      ?>
```

5.2.6.10 authorize

Normally, the AuthComponent will attempt to verify that the login credentials you've entered are accurate by comparing them to what's been stored in your user model. However, there are times where you might want to do some additional work in determining proper credentials. By setting this variable to one of several different values, you can do different things. Here are some of the more common ones you might want to use.

```
1.      <?php
2.          $this->Auth->authorize = 'controller';
3.      ?>
```

When authorize is set to 'controller', you'll need to add a method called isAuthorized() to your controller. This method allows you to do some more authentication checks and then return either true or false.

```
1.      <?php
```

```
2.            function isAuthorized() {
3.                if ($this->action == 'delete') {
4.                    if ($this->Auth->user('role') == 'admin') {
5.                        return true;
6.                    } else {
7.                        return false;
8.                    }
9.                }
10.               return true;
11.           }
12.       ?>
```

Remember that this method will be checked after you have already passed the basic authentication check against the user model.

```
1.        <?php
2.            $this->Auth->authorize = array('model'=>'User');
3.        ?>
```

Don't want to add anything to your controller and might be using ACO's? You can get the AuthComponent to call a method in your user model called isAuthorized() to do the same sort of thing:

```
1.        <?php
2.            class User extends AppModel {
3.                ...
4.                function isAuthorized($user, $controller, $action) {
5.                    switch ($action) {
6.                        case 'default':
7.                            return false;
8.                            break;
9.                        case 'delete':
```

```
                        if ($user['User']['role'] == 'admin') {
                            return true;
                        }
                        break;
                    }
                }
            }
        ?>
```

Lastly, you can use authorize with actions such as below

```
<?php
    $this->Auth->authorize = 'actions';
?>
```

By using actions, Auth will make use of ACL and check with AclComponent::check(). An isAuthorized function is not needed.

```
<?php
    $this->Auth->authorize = 'crud';
?>
```

By using crud, Auth will make use of ACL and check with AclComponent::check(). Actions should be mapped to CRUD (see mapActions).

5.2.6.11 sessionKey

Name of the session array key where the record of the current authed user is stored.

Defaults to "Auth", so if unspecified, the record is stored in "Auth.{$userModel name}".

```
<?php
```

```
2.          $this->Auth->sessionKey = 'Authorized';
3.      ?>
```

5.2.6.12 ajaxLogin

If you are doing Ajax or Javascript based requests that require authenticated sessions, set this variable to the name of a view element you would like to be rendered and returned when you have an invalid or expired session.

As with any part of CakePHP, be sure to take a look at AuthComponent class for a more in-depth look at the AuthComponent.

5.2.6.13 authenticate

This variable holds a reference to the object responsible for hashing passwords if it is necessary to change/override the default password hashing mechanism. See Changing the Encryption Type for more info.

5.2.6.14 actionPath

If using action-based access control, this defines how the paths to action ACO nodes is computed. If, for example, all controller nodes are nested under an ACO node named 'Controllers', $actionPath should be set to 'Controllers/'.

5.2.6.15 flashElement

In case that you want to have another layout for your Authentication error message you can define with the flashElement variable that another element will be used for display.

```
1.      <?php
2.          $this->Auth->flashElement    = "message_error";
3.      ?>
```

5.2.7 allowedActions

Set the default allowed actions to allow if setting the component to 'authorize' => 'controller'

```
1.        var $components = array(
2.          'Auth' => array(
3.            'authorize' => 'controller',
4.            'allowedActions' => array('index','view','display');
5.          )
6.        );
```

index, view, and display actions are now allowed by default.

5.3 Cookies

The CookieComponent is a wrapper around the native PHP setcookie method. It also includes a host of delicious icing to make coding cookies in your controllers very convenient. Before attempting to use the CookieComponent, you must make sure that 'Cookie' is listed in your controllers' $components array.

5.3.1 Controller Setup

There are a number of controller variables that allow you to configure the way cookies are created and managed. Defining these special variables in the beforeFilter() method of your controller allows you to define how the CookieComponent works.

Cookie variable	default	description
string $name	'CakeCookie'	The name of the cookie.
string $key	null	This string is used to encrypt the value written to the cookie. This string should be random and difficult to guess.
string $domain	''	The domain name allowed to access the cookie. e.g. Use '.yourdomain.com' to allow access from all your subdomains.
int or string $time	'5 Days'	The time when your cookie will expire. Integers are interpreted as seconds and a value of 0 is equivalent to a 'session cookie': i.e. the cookie expires when the browser is closed. If a string is set, this will be interpreted with PHP function

		strtotime(). You can set this directly within the write() method.
string $path	'/'	The server path on which the cookie will be applied. If $cookiePath is set to '/foo/', the cookie will only be available within the /foo/ directory and all sub-directories such as /foo/bar/ of your domain. The default value is the entire domain. You can set this directly within the write() method.
boolean $secure	false	Indicates that the cookie should only be transmitted over a secure HTTPS connection. When set to true, the cookie will only be set if a secure connection exists. You can set this directly within the write() method.

The following snippet of controller code shows how to include the CookieComponent and set up the controller variables needed to write a cookie named 'baker_id' for the domain 'example.com' which needs a secure connection, is available on the path '/bakers/preferences/', and expires in one hour.

```
1.      var $components    = array('Cookie');
2.      function beforeFilter() {
3.        $this->Cookie->name = 'baker_id';
4.        $this->Cookie->time =  3600;  // or '1 hour'
5.        $this->Cookie->path = '/bakers/preferences/';
6.        $this->Cookie->domain = 'example.com';
7.        $this->Cookie->secure = true;  //i.e. only sent if using secure HTTPS
8.        $this->Cookie->key = 'qSI232qs*&sXOw!';
9.      }
```

Next, let's look at how to use the different methods of the Cookie Component.

5.3.2 Using the Component

The CookieComponent offers a number of methods for working with Cookies.

write(mixed $key, mixed $value, boolean $encrypt, mixed $expires)

The write() method is the heart of cookie component, $key is the cookie variable name you want, and the $value is the information to be stored.

```
1.        $this->Cookie->write('name','Larry');
```

You can also group your variables by supplying dot notation in the key parameter.

```
1.        $this->Cookie->write('User.name', 'Larry');
2.          $this->Cookie->write('User.role','Lead');
```

If you want to write more than one value to the cookie at a time, you can pass an array:

```
1.        $this->Cookie->write('User',
2.            array('name'=>'Larry','role'=>'Lead')
3.        );
```

All values in the cookie are encrypted by default. If you want to store the values as plain-text, set the third parameter of the write() method to false. The encryption performed on cookie values is fairly uncomplicated encryption system. It uses `Security.salt` and a predefined Configure class var `Security.cipherSeed` to encrypt values. To make your cookies more secure you should change `Security.cipherSeed` in app/config/core.php to ensure a better encryption.

```
1.        $this->Cookie->write('name','Larry',false);
```

The last parameter to write is $expires – the number of seconds before your cookie will expire. For convenience, this parameter can also be passed as a string that the php strtotime() function understands:

```
1.        //Both cookies expire in one hour.
2.          $this->Cookie->write('first_name','Larry',false, 3600);
3.          $this->Cookie->write('last_name','Masters',false, '1 hour');
```

read(mixed $key)

This method is used to read the value of a cookie variable with the name specified by $key.

```
1.        // Outputs "Larry"
2.          echo $this->Cookie->read('name');
3.
4.          //You can also use the dot notation for read
5.          echo $this->Cookie->read('User.name');
6.
7.          //To get the variables which you had grouped
8.          //using the dot notation as an array use something like
9.          $this->Cookie->read('User');
10.
11.         // this outputs something like array('name' => 'Larry', 'role'=>'Lead')
```

delete(mixed $key)

Deletes a cookie variable of the name in $key. Works with dot notation.

```
1.          //Delete a variable
2.          $this->Cookie->delete('bar')
3.
4.          //Delete the cookie variable bar, but not all under foo
5.          $this->Cookie->delete('foo.bar')
6.
```

destroy()

Destroys the current cookie.

5.4 Email

The emailComponent is a way for you to add simple email sending functionality to your CakePHP application. Using the same concepts of layouts and view ctp files to send formated messages as text, html or both. It supports sending via the built in mail functions of PHP, via smtp server or a debug mode

where it writes the message out to a session flash message. It supports file attachments and does some basic header injection checking/ filtering for you. There is a lot that it doesn't do for you but it will get you started.

5.4.1 Class Attributes and Variables

These are the values that you can set before you call `EmailComponent::send()`

to	Address the message is going to (string). Separate the addresses with a comma if you want to send the email to more than one recipient.
cc	array of addresses to cc the message to
bcc	array of addresses to bcc (blind carbon copy) the message to
replyTo	reply to address (string)
return	Return mail address that will be used in case of any errors(string) (for mail-daemon/errors)
from	from address (string)
subject	subject for the message (string)
template	The email element to use for the message (located in `app/views/elements/email/html/` and `app/views/elements/email/text/`)
layout	The layout used for the email (located in `app/views/layouts/email/html/` and `app/views/layouts/email/text/`)
lineLength	Length at which lines should be wrapped. Defaults to 70. (integer)
sendAs	how do you want message sent string values of `text`, `html` or `both`
attachments	array of files to send (absolute and relative paths)
delivery	how to send the message (`mail`, `smtp` [would require smtpOptions set below] and `debug`)

smtpOptions	associative array of options for smtp mailer (`port`, `host`, `timeout`, `username`, `password`, `client`)

There are some other things that can be set but you should refer to the api documentation for more information

5.4.1.1 Sending Multiple Emails in a loop

If you wish to send multiple emails using a loop, you'll need to reset the email fields using the reset method of the Email component. You'll need to reset before setting the email properties again.

```
1.      $this->Email->reset()
```

5.4.1.2 Debugging Emails

If you do **not** want to actually send an email and instead want to test out the functionality, you can use the following delivery option:

```
1.      $this->Email->delivery = 'debug';
```

In order to view those debugging information you need to create an extra line in your view or layout file (e.g. underneath your normal flash message in /layouts/default.ctp):

```
1.      <?php echo $this->Session->flash(); ?>
2.      <?php echo $this->Session->flash('email'); ?>
```

5.4.2 Sending a basic message

To send a message without using a template, simply pass the body of the message as a string (or an array of lines) to the send() method. For example:

```
1.      $this->Email->from    = 'Somebody <somebody@example.com>';
2.      $this->Email->to      = 'Somebody Else <somebody.else@example.com>';
3.      $this->Email->subject = 'Test';
4.      $this->Email->send('Hello message body!');
```

5.4.2.1 Setting up the Layouts

To use both text and html mailing message you need to create layout files for them, just like in setting up your default layouts for the display of your views in a browser, you need to set up default layouts for your email messages. In the `app/views/layouts/` directory you need to set up (at a minimum) the following structure

```
1.          email/
2.              html/
3.                  default.ctp
4.              text/
5.                  default.ctp
```

These are the files that hold the layout templates for your default messages. Some example content is below

email/text/default.ctp

```
1.      <?php echo $content_for_layout; ?>
```

email/html/default.ctp

```
1.      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2.      <html>
3.          <body>
4.              <?php echo $content_for_layout; ?>
5.          </body>
6.      </html>
```

5.4.2.2 Setup an email element for the message body

In the `app/views/elements/email/` directory you need to set up folders for `text` and `html` unless you plan to just send one or the other. In each of these folders you need to create templates for both types of messages referring to the content that you send to the view either by using $this->set() or using the $contents parameter of the send() method. Some simple examples are shown below. For this example we will call the templates simple_message.ctp

text

```
1.      Dear <?php echo $User['User']['first']. ' ' . $User['User']['last'] ?>,
```

```
2.          Thank you for your interest.
```

html

```
1.      <p>Dear <?php echo $User['User']['first']. ' ' . $User['User']['last'] ?>,<br />
2.          Thank you for your interest.</p>
```

The `$content` parameter for the send() method is sent to any templates as `$content`.

5.4.2.3 Controller code for using Email component

In your controller you need to add the component to your `$components` array or add a $components array to your controller like:

```
1.      <?php
2.      var $components = array('Email');
3.      ?>
```

In this example we will set up a private method to handle sending the email messages to a user identified by an `$id`. In our controller (let's use the User controller in this example)

```
1.      <?php
2.      function _sendNewUserMail($id) {
3.          $User = $this->User->read(null,$id);
4.          $this->Email->to = $User['User']['email'];
5.          $this->Email->bcc = array('secret@example.com');
6.          $this->Email->subject = 'Welcome to our really cool thing';
7.          $this->Email->replyTo = 'support@example.com';
8.          $this->Email->from = 'Cool Web App <app@example.com>';
9.          $this->Email->template = 'simple_message'; // note no '.ctp'
10.         //Send as 'html', 'text' or 'both' (default is 'text')
11.         $this->Email->sendAs = 'both'; // because we like to send pretty mail
12.         //Set view variables as normal
```

```
            $this->set('User', $User);
            //Do not pass any args to send()
            $this->Email->send();
        }
    ?>
```

You have sent a message, you could call this from another method like

```

    $this->_sendNewUserMail( $this->User->id );
```

5.4.2.4 Attachments

Here's how you can send file attachments along with your message. You set an array containing the paths to the files to attach to the `attachments` property of the component.

```
    $this->Email->attachments = array(
        TMP . 'foo.doc',
        'bar.doc' => TMP . 'some-temp-name'
    );
```

The first file `foo.doc` will be attached with the same filename. For the second file we specify an alias `bar.doc` will be be used for attaching instead of its actual filename `some-temp-name`

5.4.3 Sending A Message Using SMTP

To send an email using an SMTP server, the steps are similar to sending a basic message. Set the delivery method to `smtp` and assign any options to the Email object's `smtpOptions` property. You may also retrieve SMTP errors generated during the session by reading the `smtpError` property of the component.

```
        /* SMTP Options */
        $this->Email->smtpOptions = array(
```

```
3.                'port'=>'25',
4.                'timeout'=>'30',
5.                'host' => 'your.smtp.server',
6.                'username'=>'your_smtp_username',
7.                'password'=>'your_smtp_password',
8.                'client' => 'smtp_helo_hostname'
9.           );
10.           /* Set delivery method */
11.           $this->Email->delivery = 'smtp';
12.           /* Do not pass any args to send() */
13.           $this->Email->send();
14.           /* Check for SMTP errors. */
15.           $this->set('smtp_errors', $this->Email->smtpError);
```

If your SMTP server requires authentication, be sure to specify the username and password parameters for `smtpOptions` as shown in the example.

If you don't know what an SMTP HELO is, then you most likely will not need to set the `client` parameter for the `smtpOptions`. This is only needed for compatibility with SMTP servers which do not fully respect RFC 821 (SMTP HELO).

Here are example options for using Gmail's SMTP server.

```
1.           /* SMTP Options */
2.           $this->Email->smtpOptions = array(
3.                'port'=>'465',
4.                'timeout'=>'30',
5.                'host' => 'ssl://smtp.gmail.com',
6.                'username'=>'your_username@gmail.com',
7.                'password'=>'your_gmail_password',
8.           );
```

5.5 Request Handling

The Request Handler component is used in CakePHP to obtain additional information about the HTTP requests that are made to your applications. You can use it to inform your controllers about Ajax as well as gain additional insight into content types that the client accepts and automatically changes to the appropriate layout when file extensions are enabled.

By default RequestHandler will automatically detect Ajax requests based on the HTTP-X-Requested-With header that many javascript libraries use. When used in conjunction with Router::parseExtensions() RequestHandler will automatically switch the layout and view files to those that match the requested type. Furthermore, if a helper with the same name as the requested extension exists, it will be added to the Controllers Helper array. Lastly, if XML data is POST'ed to your Controllers, it will be parsed into an XML object which is assigned to Controller::data, and can then be saved as model data. In order to make use of Request Handler it must be included in your $components array.

```
1.      <?php
2.      class WidgetController extends AppController {
3.
4.          var $components = array('RequestHandler');
5.
6.          //rest of controller
7.      }
8.      ?>
```

5.5.1 Obtaining Request Information

Request Handler has several methods that provide information about the client and its request.

accepts ($type = null)

$type can be a string, or an array, or null. If a string, accepts will return true if the client accepts the content type. If an array is specified, accepts return true if any one of the content types is accepted by the client. If null returns an array of the content-types that the client accepts. For example:

```
1.      class PostsController extends AppController {
```

```
2.
3.          var $components = array('RequestHandler');
4.          function beforeFilter () {
5.              if ($this->RequestHandler->accepts('html')) {
6.                  // Execute code only if client accepts an HTML (text/html) response
7.              } elseif ($this->RequestHandler->accepts('xml')) {
8.                  // Execute XML-only code
9.              }
10.             if ($this->RequestHandler->accepts(array('xml', 'rss', 'atom'))) {
11.                 // Executes if the client accepts any of the above: XML, RSS or Atom
12.             }
13.         }
14.     }
```

Other request 'type' detection methods include:

isAjax()

Returns true if the request contains the X-Requested-Header equal to XMLHttpRequest.

isSSL()

Returns true if the current request was made over an SSL connection.

isXml()

Returns true if the current request accepts XML as a response.

isRss()

Returns true if the current request accepts RSS as a response.

isAtom()

Returns true if the current call accepts an Atom response, false otherwise.

isMobile()

Returns true if user agent string matches a mobile web browser, or if the client accepts WAP content. The supported Mobile User Agent strings are:

- iPhone
- MIDP
- AvantGo
- BlackBerry
- J2ME
- Opera Mini
- DoCoMo
- NetFront
- Nokia
- PalmOS
- PalmSource
- portalmmm
- Plucker
- ReqwirelessWeb
- SonyEricsson
- Symbian
- UP.Browser
- Windows CE
- Xiino

isWap()

Returns true if the client accepts WAP content.

All of the above request detection methods can be used in a similar fashion to filter functionality intended for specific content types. For example when responding to Ajax requests, you often will want to disable browser caching, and change the debug level. However, you want to allow caching for non-ajax requests. The following would accomplish that:

```
1.          if ($this->RequestHandler->isAjax()) {
2.              Configure::write('debug', 0);
3.              $this->header('Pragma: no-cache');
4.              $this->header('Cache-control: no-cache');
5.              $this->header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
6.          }
7.          //Continue Controller action
```

You could also disable caching with the functionally analogous `Controller::disableCache`

```
1.          if ($this->RequestHandler->isAjax()) {
2.              $this->disableCache();
3.          }
4.          //Continue Controller action
```

5.5.2 Request Type Detection

RequestHandler also provides information about what type of HTTP request has been made and allowing you to respond to each Request Type.

isPost()

Returns true if the request is a POST request.

isPut()

Returns true if the request is a PUT request.

isGet()

Returns true if the request is a GET request.

isDelete()

Returns true if the request is a DELETE request.

5.5.3 Obtaining Additional Client Information

getClientIP()

Get the remote client IP address

getReferer()

Returns the domain name from which the request originated

getAjaxVersion()

Gets Prototype version if call is Ajax, otherwise empty string. The Prototype library sets a special "Prototype version" HTTP header.

5.5.4 Responding To Requests

In addition to request detection RequestHandler also provides easy access to altering the output and content type mappings for your application.

setContent($name, $type = null)

- $name string - The name or file extension of the Content-type ie. html, css, json, xml.
- $type mixed - The mime-type(s) that the Content-type maps to.

setContent adds/sets the Content-types for the given name. Allows content-types to be mapped to friendly aliases and or extensions. This allows RequestHandler to automatically respond to requests of each type in its startup method. If you are using Router::parseExtension, you should use the file extension as the name of the Content-type. Furthermore, these content types are used by prefers() and accepts().

setContent is best used in the beforeFilter() of your controllers, as this will best leverage the automagicness of content-type aliases.

The default mappings are:

- **javascript** text/javascript
- **js** text/javascript
- **json** application/json
- **css** text/css
- **html** text/html, */*
- **text** text/plain
- **txt** text/plain
- **csv** application/vnd.ms-excel, text/plain
- **form** application/x-www-form-urlencoded
- **file** multipart/form-data
- **xhtml** application/xhtml+xml, application/xhtml, text/xhtml
- **xhtml-mobile** application/vnd.wap.xhtml+xml
- **xml** application/xml, text/xml
- **rss** application/rss+xml
- **atom** application/atom+xml
- **amf** application/x-amf
- **wap** text/vnd.wap.wml, text/vnd.wap.wmlscript, image/vnd.wap.wbmp
- **wml** text/vnd.wap.wml
- **wmlscript** text/vnd.wap.wmlscript
- **wbmp** image/vnd.wap.wbmp
- **pdf** application/pdf
- **zip** application/x-zip
- **tar** application/x-tar

prefers($type = null)

Determines which content-types the client prefers. If no parameter is given the most likely content type is returned. If $type is an array the first type the client accepts will be returned. Preference is determined primarily by the file extension parsed by Router if one has been provided, and secondly by the list of content-types in HTTP_ACCEPT.

renderAs($controller, $type)

- $controller - Controller Reference
- $type - friendly content type name to render content for ex. xml, rss.

Change the render mode of a controller to the specified type. Will also append the appropriate helper to the controller's helper array if available and not already in the array.

respondAs($type, $options)

- $type - Friendly content type name ex. xml, rss or a full content type like application/x-shockwave
- $options - If $type is a friendly type name that has more than one content association, $index is used to select the content type.

Sets the response header based on content-type map names. If DEBUG is greater than 1, the header is not set.

responseType()

Returns the current response type Content-type header or null if one has yet to be set.

mapType($ctype)

Maps a content-type back to an alias

5.6 Security Component

The Security Component creates an easy way to integrate tighter security in your application. An interface for managing HTTP-authenticated requests can be created with Security Component. It is configured in the beforeFilter() of your controllers. It has several configurable parameters. All of these properties can be set directly or through setter methods of the same name.

If an action is restricted using the Security Component it is black-holed as an invalid request which will result in a 404 error by default. You can configure this behavior by setting the $this->Security->blackHoleCallback property to a callback function in the controller. Keep in mind that black holes from all of the Security Component's methods will be run through this callback method.

By using the Security Component you automatically get CSRF and form tampering protection. Hidden token fields will automatically be inserted into forms and checked by the Security component. Among other things, a form submission will not be accepted after a certain period of inactivity, which depends on the setting of `Security.level`. On 'high', this timeout is as short as 10 minutes. Other token fields include a randomly generated nonce (one-time id) and a hash of fields which (and only which) must be present in the submitted POST data.

If you are using Security component's form protection features and other components that process form data in their `startup()` callbacks, be sure to place Security Component before those components in your `$components` array.

When using the Security Component you **must** use the FormHelper to create your forms. The Security Component looks for certain indicators that are created and managed by the FormHelper (especially those created in create() and end()). Dynamically altering the fields that are submitted in a POST request (e.g. disabling, deleting or creating new fields via JavaScript) is likely to trigger a black-holing of the request. See the `$validatePost` or `$disabledFields` configuration parameters.

5.6.1 Configuration

$blackHoleCallback

A Controller callback that will handle and requests that are blackholed.

$requirePost

A List of controller actions that require a POST request to occur. An array of controller actions or '*' to force all actions to require a POST.

$requireSecure

List of actions that require an SSL connection to occur. An array of controller actions or '*' to force all actions to require a SSL connection.

$requireAuth

List of actions that requires a valid authentication key. This validation key is set by Security Component.

$requireLogin

List of actions that require HTTP-Authenticated logins (basic or digest). Also accepts '*' indicating that all actions of this controller require HTTP-authentication.

$loginOptions

Options for HTTP-Authenticate login requests. Allows you to set the type of authentication and the controller callback for the authentication process.

$loginUsers

An associative array of usernames => passwords that are used for HTTP-authenticated logins. If you are using digest authentication, your passwords should be MD5-hashed.

$allowedControllers

A List of Controller from which the actions of the current controller are allowed to receive requests from. This can be used to control cross controller requests.

$allowedActions

Actions from which actions of the current controller are allowed to receive requests. This can be used to control cross controller requests.

$disabledFields

List of form fields that shall be ignored when validating POST - The value, presence or absence of these form fields will not be taken into account when evaluating whether a form submission is valid. Specify fields as you do for the Form Helper (`Model.fieldname`).

$validatePost

Set to `false` to completely skip the validation of POST requests, essentially turning CSRF protection off.

5.6.2 Methods

5.6.2.1 requirePost()

Sets the actions that require a POST request. Takes any number of arguments. Can be called with no arguments to force all actions to require a POST.

5.6.2.2 requireSecure()

Sets the actions that require a SSL-secured request. Takes any number of arguments. Can be called with no arguments to force all actions to require a SSL-secured.

5.6.2.3 requireAuth()

Sets the actions that require a valid Security Component generated token. Takes any number of arguments. Can be called with no arguments to force all actions to require a valid authentication.

5.6.2.4 requireLogin()

Sets the actions that require a valid HTTP-Authenticated request. Takes any number of arguments. Can be called with no arguments to force all actions to require valid HTTP-authentication.

5.6.2.5 loginCredentials(string $type)

Attempt to validate login credentials for a HTTP-authenticated request. $type is the type of HTTP-Authentication you want to check. Either 'basic', or 'digest'. If left null/empty both will be tried. Returns an array with login name and password if successful.

5.6.2.6 loginRequest(array $options)

Generates the text for an HTTP-Authenticate request header from an array of $options.

$options generally contains a 'type', 'realm' . Type indicate which HTTP-Authenticate method to use. Realm defaults to the current HTTP server environment.

5.6.2.7 parseDigestAuthData(string $digest)

Parse an HTTP digest authentication request. Returns and array of digest data as an associative array if succesful, and null on failure.

5.6.2.8 generateDigestResponseHash(array $data)

Creates a hash that to be compared with an HTTP digest-authenticated response. $data should be an array created by SecurityComponent::parseDigestAuthData().

5.6.2.9 blackHole(object $controller, string $error)

Black-hole an invalid request with a 404 error or a custom callback. With no callback, the request will be exited. If a controller callback is set to SecurityComponent::blackHoleCallback, it will be called and passed any error information.

5.6.3 Usage

Using the security component is generally done in the controller beforeFilter(). You would specify the security restrictions you want and the Security Component will enforce them on its startup.

```
1.  <?php
2.  class WidgetController extends AppController {
3.      var $components = array('Security');
4.      function beforeFilter() {
5.          $this->Security->requirePost('delete');
6.      }
7.  }
8.  ?>
```

In this example the delete action can only be successfully triggered if it recieves a POST request.

```
<?php
class WidgetController extends AppController {
    var $components = array('Security');
    function beforeFilter() {
        if(isset($this->params[Configure::read('Routing.admin')])){
            $this->Security->requireSecure();
        }
    }
}
?>
```

This example would force all actions that had admin routing to require secure SSL requests.

```
<?php
class WidgetController extends AppController {
    var $components = array('Security');
    function beforeFilter() {
        if(isset($this->params[Configure::read('Routing.admin')])){
            $this->Security->blackHoleCallback = 'forceSSL';
            $this->Security->requireSecure();
        }
    }
    function forceSSL() {
        $this->redirect('https://' . env('SERVER_NAME') . $this->here);
    }
}
?>
```

This example would force all actions that had admin routing to require secure SSL requests. When the request is black holed, it will call the nominated forceSSL() callback which will redirect non-secure requests to secure requests automatically.

5.6.4 Basic HTTP Authentication

The SecurityComponent has some very powerful authentication features. Sometimes you may need to protect some functionality inside your application using HTTP Basic Authentication. One common usage for HTTP Auth is protecting a REST or SOAP API.

This type of authentication is called basic for a reason. Unless you're transferring information over SSL, credentials will be transferred in plain text.

Using the SecurityComponent for HTTP authentication is easy. The code example below includes the SecurityComponent and adds a few lines of code inside the controller's beforeFilter method.

```
1.      class ApiController extends AppController {
2.          var $name = 'Api';
3.          var $uses = array();
4.          var $components = array('Security');
5.          function beforeFilter() {
6.              $this->Security->loginOptions = array(
7.                  'type'=>'basic',
8.                  'realm'=>'MyRealm'
9.              );
10.             $this->Security->loginUsers = array(
11.                 'john'=>'johnspassword',
12.                 'jane'=>'janespassword'
13.             );
14.             $this->Security->requireLogin();
15.         }
16.
17.         function index() {
18.             //protected application logic goes here...
19.         }
20.     }
```

The loginOptions property of the SecurityComponent is an associative array specifying how logins should be handled. You only need to specify the **type** as **basic** to get going. Specify the **realm** if you want display a nice message to anyone trying to login or if you have several authenticated sections (= realms) of your application you want to keep separate.

The loginUsers property of the SecurityComponent is an associative array containing users and passwords that should have access to this realm. The examples here use hard-coded user information, but you'll probably want to use a model to make your authentication credentials more manageable.

Finally, requireLogin() tells SecurityComponent that this Controller requires login. As with requirePost(), above, providing method names will protect those methods while keeping others open.

5.7 Sessions

The CakePHP session component provides a way to persist client data between page requests. It acts as a wrapper for the $_SESSION as well as providing convenience methods for several $_SESSION related functions.

Sessions can be persisted in a few different ways. The default is to use the settings provided by PHP; however, other options exist.

cake

Saves the session files in your app's tmp/sessions directory.

database

Uses CakePHP's database sessions.

cache

Use the caching engine configured by Cache::config(). Very useful in conjunction with Memcache (in setups with multiple application servers) to store both cached data and sessions.

php

The default setting. Saves session files as indicated by php.ini

To change the default Session handling method alter the Session.save Configuration to reflect the option you desire. If you choose 'database' you should also uncomment the Session.database settings and run the database session SQL file located in app/config

To provide a custom configuration, set Session.save Configuration to a filename. CakePHP will use your file in the CONFIGS directory for the settings.

```
1.  // app/config/core.php
2.  Configure::write('Session.save','my_session');
```

This will allow you to customize the session handling.

```
1.  // app/config/my_session.php
2.  //
3.  // Revert value and get rid of the referrer check even when,
4.  // Security.level is medium
5.  ini_restore('session.referer_check');
6.  ini_set('session.use_trans_sid', 0);
7.  ini_set('session.name', Configure::read('Session.cookie'));
8.  // Cookie is now destroyed when browser is closed, doesn't
9.  // persist for days as it does by default for security
10. // low and medium
11. ini_set('session.cookie_lifetime', 0);
12. // Cookie path is now '/' even if you app is within a sub
13. // directory on the domain
14. $this->path = '/';
15. ini_set('session.cookie_path', $this->path);
16. // Session cookie now persists across all subdomains
17. ini_set('session.cookie_domain', env('HTTP_BASE'));
```

5.7.1 Methods

The Session component is used to interact with session information. It includes basic CRUD functions as well as features for creating feedback messages to users.

It should be noted that Array structures can be created in the Session by using dot notation. So User.username would reference the following:

```
1.          array('User' =>
2.                  array('username' => 'clarkKent@dailyplanet.com')
3.          );
```

Dots are used to indicate nested arrays. This notation is used for all Session component methods wherever a $name is used.

5.7.1.1 write

write($name, $value)

Write to the Session puts $value into $name. $name can be a dot separated array. For example:

```
1.      $this->Session->write('Person.eyeColor', 'Green');
```

This writes the value 'Green' to the session under Person => eyeColor.

5.7.1.2 setFlash

setFlash($message, $element = 'default', $params = array(), $key = 'flash')

Used to set a session variable that can be used for output in the View. $element allows you to control which element (located in `/app/views/elements`) should be used to render the message in. In the element the message is available as `$message`. If you leave the `$element` set to 'default', the message will be wrapped with the following:

```
1.      <div id="flashMessage" class="message"> [message] </div>
```

$params allows you to pass additional view variables to the rendered layout. $key sets the $messages index in the Message array. Default is 'flash'.

Parameters can be passed affecting the rendered div, for example adding "class" in the $params array will apply a class to the `div` output using `$session->flash()` in your layout or view.

```
1.      $this->Session->setFlash('Example message text', 'default', array('class' => 'example_class'))
```

The output from using `$session->flash()` with the above example would be:

```
1.      <div id="flashMessage" class="example_class">Example message text</div>
```

5.7.1.3 read

`read($name)`

Returns the value at $name in the Session. If $name is null the entire session will be returned. E.g.

```
1.      $green = $this->Session->read('Person.eyeColor');
```

Retrieve the value Green from the session.

5.7.1.4 check

`check($name)`

Used to check if a Session variable has been set. Returns true on existence and false on non-existence.

5.7.1.5 delete

`delete($name)`

Clear the session data at $name. E.g.

```
1.      $this->Session->delete('Person.eyeColor');
```

Our session data no longer has the value 'Green', or the index eyeColor set. However, Person is still in the Session. To delete the entire Person information from the session use.

```
1.      $this->Session->delete('Person');
```

5.7.1.6 destroy

The `destroy` method will delete the session cookie and all session data stored in the temporary file system. It will then destroy the PHP session and then create a fresh session.

```
1.      $this->Session->destroy()
```

5.7.1.7 error

`error()`

Used to determine the last error in a session.

6 Core Behaviors

Behaviors add extra functionality to your models. CakePHP comes with a number of built-in behaviors such as Translate, Tree and the mighty Containable.

6.1 ACL

The Acl behavior provides a way to seamlessly integrate a model with your ACL system. It can create both AROs or ACOs transparently.

To use the new behavior, you can add it to the $actsAs property of your model. When adding it to the actsAs array you choose to make the related Acl entry an ARO or an ACO. The default is to create AROs.

```
1.      class User extends AppModel {
2.          var $actsAs = array('Acl' => array('type' =>  'requester'));
3.      }
```

This would attach the Acl behavior in ARO mode. To join the ACL behavior in ACO mode use:

```
1.      class Post extends AppModel {
2.          var $actsAs = array('Acl' => array('type' => 'controlled'));
3.      }
```

You can also attach the behavior on the fly like so:

```
1.          $this->Post->Behaviors->attach('Acl', array('type' => 'controlled'));
```

6.1.1 Using the AclBehavior

Most of the AclBehavior works transparently on your Model's afterSave(). However, using it requires that your Model has a parentNode() method defined. This is used by the AclBehavior to determine parent->child relationships. A model's parentNode() method must return null or return a parent Model reference.

```
1.      function parentNode() {
2.          return null;
3.      }
```

If you want to set an ACO or ARO node as the parent for your Model, parentNode() must return the alias of the ACO or ARO node.

```
1.      function parentNode() {
2.              return 'root_node';
3.      }
```

A more complete example. Using an example User Model, where User belongsTo Group.

```
1.      function parentNode() {
2.          if (!$this->id && empty($this->data)) {
3.              return null;
4.          }
5.          $data = $this->data;
6.          if (empty($this->data)) {
7.              $data = $this->read();
8.          }
9.          if (!$data['User']['group_id']) {
10.             return null;
11.         } else {
12.             $this->Group->id = $data['User']['group_id'];
13.             $groupNode = $this->Group->node();
14.             return array('Group' => array('id' => $groupNode[0]['Aro']['foreign_key']));
15.         }
16.     }
```

In the above example the return is an array that looks similar to the results of a model find. It is important to have the id value set or the parentNode relation will fail. The AclBehavior uses this data to construct its tree structure.

6.1.2 node()

The AclBehavior also allows you to retrieve the Acl node associated with a model record. After setting $model->id. You can use $model->node() to retrieve the associated Acl node.

You can also retrieve the Acl Node for any row, by passing in a data array.

```
1.          $this->User->id = 1;
2.          $node = $this->User->node();
```

```
3.
4.          $user = array('User' => array(
5.              'id' => 1
6.          ));
7.          $node = $this->User->node($user);
```

Will both return the same Acl Node information.

6.2 Containable

A new addition to the CakePHP 1.2 core is the `ContainableBehavior`. This model behavior allows you to filter and limit model find operations. Using Containable will help you cut down on needless wear and tear on your database, increasing the speed and overall performance of your application. The class will also help you search and filter your data for your users in a clean and consistent way.

Containable allows you to streamline and simplify operations on your model bindings. It works by temporarily or permanently altering the associations of your models. It does this by using the supplied containments to generate a series of `bindModel` and `unbindModel` calls.

To use the new behavior, you can add it to the $actsAs property of your model:

```
1.      class Post extends AppModel {
2.          var $actsAs = array('Containable');
3.      }
```

You can also attach the behavior on the fly:

```
1.      $this->Post->Behaviors->attach('Containable');
```

Using Containable

To see how Containable works, let's look at a few examples. First, we'll start off with a find() call on a model named Post. Let's say that Post hasMany Comment, and Post hasAndBelongsToMany Tag. The amount of data fetched in a normal find() call is rather extensive:

```
1.      debug($this->Post->find('all'));
```

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Comment] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [post_id] => 1
                            [author] => Daniel
                            [email] => dan@example.com
                            [website] => http://example.com
                            [comment] => First comment
                            [created] => 2008-05-18 00:00:00
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [post_id] => 1
                            [author] => Sam
                            [email] => sam@example.net
                            [website] => http://example.net
                            [comment] => Second comment
                            [created] => 2008-05-18 00:00:00
                        )
                )
            [Tag] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
```

```
                            [name] => Awesome
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [name] => Baking
                        )
                )
        )
[1] => Array
        (
            [Post] => Array
                (...
```

For some interfaces in your application, you may not need that much information from the Post model. One thing the `ContainableBehavior` does is help you cut down on what find() returns.

For example, to get only the post-related information, you can do the following:

```
1.      $this->Post->contain();
2.      $this->Post->find('all');
```

You can also invoke Containable's magic from inside the find() call:

```
1.      $this->Post->find('all', array('contain' => false));
```

Having done that, you end up with something a lot more concise:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
```

```
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
        )
[1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => Second article
                    [content] => bbb
                    [created] => 2008-05-19 00:00:00
                )
        )
```

This sort of help isn't new: in fact, you can do that without the `ContainableBehavior` doing something like this:

```
1.      $this->Post->recursive = -1;
2.      $this->Post->find('all');
```

Containable really shines when you have complex associations, and you want to pare down things that sit at the same level. The model's `$recursive` property is helpful if you want to hack off an entire level of recursion, but not when you want to pick and choose what to keep at each level. Let's see how it works by using the `contain()` method.

The contain method's first argument accepts the name, or an array of names, of the models to keep in the find operation. If we wanted to fetch all posts and their related tags (without any comment information), we'd try something like this:

```
1.      $this->Post->contain('Tag');
2.      $this->Post->find('all');
```

Again, we can use the contain key inside a find() call:

```
1.      $this->Post->find('all', array('contain' => 'Tag'));
```

Without Containable, you'd end up needing to use the `unbindModel()` method of the model, multiple times if you're paring off multiple models. Containable creates a cleaner way to accomplish this same task.

Containing deeper associations

Containable also goes a step deeper: you can filter the data of the *associated* models. If you look at the results of the original find() call, notice the author field in the Comment model. If you are interested in the posts and the names of the comment authors — and nothing else — you could do something like the following:

```
1.  $this->Post->contain('Comment.author');
2.  $this->Post->find('all');
3.  //or..
4.  $this->Post->find('all', array('contain' => 'Comment.author'));
```

Here, we've told Containable to give us our post information, and just the author field of the associated Comment model. The output of the find call might look something like this:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Comment] => Array
                (
                    [0] => Array
                        (
                            [author] => Daniel
                            [post_id] => 1
                        )
                    [1] => Array
                        (
```

```
                            [author] => Sam
                            [post_id] => 1
                        )
                )
        )
[1] => Array
        (...
```

As you can see, the Comment arrays only contain the author field (plus the post_id which is needed by CakePHP to map the results).

You can also filter the associated Comment data by specifying a condition:

```
1.      $this->Post->contain('Comment.author = "Daniel"');
2.      $this->Post->find('all');
3.      //or...
4.      $this->Post->find('all', array('contain' => 'Comment.author = "Daniel"'));
```

This gives us a result that gives us posts with comments authored by Daniel:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Comment] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [post_id] => 1
                            [author] => Daniel
                            [email] => dan@example.com
```

```
                            [website] => http://example.com
                            [comment] => First comment
                            [created] => 2008-05-18 00:00:00
                        )
                )
        )
```

Additional filtering can be performed by supplying the standard `Model->find()` options:

```
1.      $this->Post->find('all', array('contain' => array(
2.          'Comment' => array(
3.              'conditions' => array('Comment.author =' => "Daniel"),
4.              'order' => 'Comment.created DESC'
5.          )
6.      )));
```

Here's an example of using the `ContainableBehavior` when you've got deep and complex model relationships.

Let's consider the following model associations:

```
User->Profile
User->Account->AccountSummary
User->Post->PostAttachment->PostAttachmentHistory->HistoryNotes
User->Post->Tag
```

This is how we retrieve the above associations with Containable:

```
1.      $this->User->find('all', array(
2.          'contain'=>array(
3.              'Profile',
4.              'Account' => array(
5.                  'AccountSummary'
6.              ),
```

```
                'Post' => array(
                    'PostAttachment' => array(
                        'fields' => array('id', 'name'),
                        'PostAttachmentHistory' => array(
                            'HistoryNotes' => array(
                                'fields' => array('id', 'note')
                            )
                        )
                    ),
                    'Tag' => array(
                        'conditions' => array('Tag.name LIKE' => '%happy%')
                    )
                )
            )
        ));
```

Keep in mind that `contain` key is only used once in the main model, you don't need to use 'contain' again for related models

When using 'fields' and 'contain' options - be careful to include all foreign keys that your query directly or indirectly requires. Please also note that because Containable must to be attached to all models used in containment, you may consider attaching it to your AppModel.

ContainableBehavior options

The `ContainableBehavior` has a number of options that can be set when the Behavior is attached to a model. The settings allow you to fine tune the behavior of Containable and work with other behaviors more easily.

- **recursive** (boolean, optional) set to true to allow containable to automatically determine the recursiveness level needed to fetch specified models, and set the model recursiveness to this level. setting it to false disables this feature. The default value is `true`.
- **notices** (boolean, optional) issues E_NOTICES for bindings referenced in a containable call that are not valid. The default value is `true`.
- **autoFields**: (boolean, optional) auto-add needed fields to fetch requested bindings. The default value is `true`.

You can change ContainableBehavior settings at run time by reattaching the behavior as seen in Using behaviors

ContainableBehavior can sometimes cause issues with other behaviors or queries that use aggregate functions and/or GROUP BY statements. If you get invalid SQL errors due to mixing of aggregate and non-aggregate fields, try disabling the `autoFields` setting.

```
1.      $this->Post->Behaviors->attach('Containable', array('autoFields' => false));
```

6.2.1 Using Containable with pagination

By including the 'contain' parameter in the `$paginate` property it will apply to both the find('count') and the find('all') done on the model

See the section Using Containable for further details.

Here's an example of how to contain associations when paginating.

```
1.      $this->paginate['User'] = array(
2.          'contain' => array('Profile', 'Account'),
3.          'order' => 'User.username'
4.      );
5.      $users = $this->paginate('User');
```

6.3 Translate

TranslateBehavior is actually quite easy to setup and works out of the box with very little configuration. In this section, you will learn how to add and setup the behavior to use in any model.

If you are using TranslateBehavior in alongside containable issue, be sure to set the 'fields' key for your queries. Otherwise you could end up with invalid SQL generated.

6.3.1 Initializing the i18n Database Tables

You can either use the CakePHP console or you can manually create it. It is advised to use the console for this, because it might happen that the layout changes in future versions of CakePHP. Sticking to the console will make sure that you have the correct layout.

```
./cake i18n
```

Select `[I]` which will run the i18n database intialization script. You will be asked if you want to drop any existing and if you want to create it. Answer with yes if you are sure there is no i18n table already, and answer with yes again to create the table.

6.3.2 Attaching the Translate Behavior to your Models

Add it to your model by using the `$actsAs` property like in the following example.

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate'
    );
}
?>
```

This will do nothing yet, because it expects a couple of options before it begins to work. You need to define which fields of the current model should be tracked in the translation table we've created in the first step.

6.3.3 Defining the Fields

You can set the fields by simply extending the `'Translate'` value with another array, like so:

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

```
4.          var $actsAs = array(
5.              'Translate' => array(
6.                  'fieldOne', 'fieldTwo', 'and_so_on'
7.              )
8.          );
9.      }
10.     ?>
```

After you have done that (for example putting "name" as one of the fields) you already finished the basic setup. Great! According to our current example the model should now look something like this:

```
1.      <?php
2.      class Post extends AppModel {
3.          var $name = 'Post';
4.          var $actsAs = array(
5.              'Translate' => array(
6.                  'name'
7.              )
8.          );
9.      }
10.     ?>
```

When defining fields for TranslateBehavior to translate, be sure to omit those fields from the translated model's schema. If you leave the fields in, there can be issues when retrieving data with fallback locales.

6.3.4 Conclusion

From now on each record update/creation will cause TranslateBehavior to copy the value of "name" to the translation table (default: i18n) along with the current locale. A locale is the identifier of the language, so to speak.

The *current locale* is the current value of `Configure::read('Config.language')`. The value of *Config.language* is assigned in the L10n Class - unless it is already set. However, the TranslateBehavior allows you to override this on-the-fly, which allows the user of your page to create multiple versions without the need to change his preferences. More about this in the next section.

6.3.5 Retrieve all translation records for a field

If you want to have all translation records attached to the current model record you simply extend the *field array* in your behavior setup as shown below. The naming is completely up to you.

```
<?php
class Post extends AppModel {
var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name' => 'nameTranslation'
        )
    );
}
?>
```

With this setup the result of $this->Post->find() should look something like this:

```
Array
(
     [Post] => Array
         (
             [id] => 1
             [name] => Beispiel Eintrag
             [body] => lorem ipsum...
             [locale] => de_de
         )
```

```
            [nameTranslation] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [locale] => en_us
                            [model] => Post
                            [foreign_key] => 1
                            [field] => name
                            [content] => Example entry
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [locale] => de_de
                            [model] => Post
                            [foreign_key] => 1
                            [field] => name
                            [content] => Beispiel Eintrag
                        )
                )
        )
```

Note: The model record contains a *virtual* field called "locale". It indicates which locale is used in this result.

Note that only fields of the model you are directly doing `find` on will be translated. Models attached via associations won't be translated because triggering callbacks on associated models is currently not supported.

6.3.5.1 Using the bindTranslation method

You can also retrieve all translations, only when you need them, using the bindTranslation method

bindTranslation($fields, $reset)

`$fields` is a named-key array of field and association name, where the key is the translatable field and the value is the fake association name.

```
1.  $this->Post->bindTranslation(array ('name' => 'nameTranslation'));
2.  $this->Post->find('all', array ('recursive'=>1)); // need at least recursive 1 for this to work.
```

With this setup the result of your find() should look something like this:

```
1.  Array
2.  (
3.      [Post] => Array
4.          (
5.              [id] => 1
6.              [name] => Beispiel Eintrag
7.              [body] => lorem ipsum...
8.              [locale] => de_de
9.          )
10.     [nameTranslation] => Array
11.         (
12.             [0] => Array
13.                 (
14.                     [id] => 1
15.                     [locale] => en_us
16.                     [model] => Post
17.                     [foreign_key] => 1
18.                     [field] => name
19.                     [content] => Example entry
20.                 )
21.             [1] => Array
22.                 (
```

```
                [id] => 2
                [locale] => de_de
                [model] => Post
                [foreign_key] => 1
                [field] => name
                [content] => Beispiel Eintrag
            )
        )
    )
```

6.3.6 Saving in another language

You can force the model which is using the TranslateBehavior to save in a language other than the on detected.

To tell a model in what language the content is going to be you simply change the value of the `$locale` property on the model before you save the data to the database. You can do that either in your controller or you can define it directly in the model.

Example A: In your controller

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';

    function add() {
        if ($this->data) {
            $this->Post->locale = 'de_de'; // we are going to save the german version
            $this->Post->create();
            if ($this->Post->save($this->data)) {
                $this->redirect(array('action' => 'index'));
            }
        }
    }
```

```
    }
?>
```

Example B: In your model

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name'
        )
    );

    // Option 1) just define the property directly
    var $locale = 'en_us';

    // Option 2) create a simple method
    function setLanguage($locale) {
        $this->locale = $locale;
    }
}
?>
```

6.3.7 Multiple Translation Tables

If you expect a lot entries you probably wonder how to deal with a rapidly growing database table. There are two properties introduced by TranslateBehavior that allow to specify which "Model" to bind as the model containing the translations.

These are **$translateModel** and **$translateTable**.

Lets say we want to save our translations for all posts in the table "post_i18ns" instead of the default "i18n" table. To do so you need to setup your model like this:

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name'
        )
    );

    // Use a different model (and table)
    var $translateModel = 'PostI18n';
}
?>
```

Important is that you have to pluralize the table. It is now a usual model and can be treated as such and thus comes with the conventions involved. The table schema itself must be identical with the one generated by the CakePHP console script. To make sure it fits one could just initialize a empty i18n table using the console and rename the table afterwards.

6.3.7.1 Create the TranslateModel

For this to work you need to create the actual model file in your models folder. Reason is that there is no property to set the displayField directly in the model using this behavior yet.

Make sure that you change the `$displayField` to `'field'`.

```
<?php
class PostI18n extends AppModel {
    var $displayField = 'field'; // important
}
// filename: post_i18n.php
?>
```

That's all it takes. You can also add all other model stuff here like $useTable. But for better consistency we could do that in the model which actually uses this translation model. This is where the optional `$translateTable` comes into play.

6.3.7.2 Changing the Table

If you want to change the name of the table you simply define $translateTable in your model, like so:

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name'
        )
    );

    // Use a different model
    var $translateModel = 'PostI18n';

    // Use a different table for translateModel
    var $translateTable = 'post_translations';
}
?>
```

Please note that **you can't use $translateTable alone**. If you don't intend to use a custom `$translateModel` then leave this property untouched. Reason is that it would break your setup and show you a "Missing Table" message for the default I18n model which is created in runtime.

6.4 Tree

It's fairly common to want to store hierarchical data in a database table. Examples of such data might be categories with unlimited subcategories, data related to a multilevel menu system or a literal representation of hierarchy such as is used to store access control objects with ACL logic.

For small trees of data, or where the data is only a few levels deep it is simple to add a parent_id field to your database table and use this to keep track of which item is the parent of what. Bundled with cake however, is a powerful behavior which allows you to use the benefits of MPTT logic without worrying about any of the intricacies of the technique - unless you want to ;).

6.4.1 Requirements

To use the tree behavior, your database table needs 3 fields as listed below (all are ints):

- parent - default fieldname is parent_id, to store the id of the parent object
- left - default fieldname is lft, to store the lft value of the current row.
- right - default fieldname is rght, to store the rght value of the current row.

If you are familiar with MPTT logic you may wonder why a parent field exists - quite simply it's easier to do certain tasks if a direct parent link is stored on the database - such as finding direct children.

The `parent` field must be able to have a NULL value! It might seem to work, if you just give the top elements a parent value of zero, but reordering the tree (and possible other operations) will fail.

6.4.2 Basic Usage

The tree behavior has a lot packed into it, but let's start with a simple example - create the following database table and put some data in it:

```
CREATE TABLE categories (
        id INTEGER(10) UNSIGNED NOT NULL AUTO_INCREMENT,
        parent_id INTEGER(10) DEFAULT NULL,
        lft INTEGER(10) DEFAULT NULL,
        rght INTEGER(10) DEFAULT NULL,
        name VARCHAR(255) DEFAULT '',
        PRIMARY KEY  (id)
);

INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(1, 'My Categories', NULL, 1, 30);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(2, 'Fun', 1, 2, 15);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(3, 'Sport', 2, 3, 8);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(4, 'Surfing', 3, 4, 5);
```

```
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(5, 'Extreme knitting', 3, 6, 7);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(6, 'Friends', 2, 9, 14);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(7, 'Gerald', 6, 10, 11);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(8, 'Gwendolyn', 6, 12, 13);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(9, 'Work', 1, 16, 29);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(10, 'Reports', 9, 17, 22);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(11, 'Annual', 10, 18, 19);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(12, 'Status', 10, 20, 21);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(13, 'Trips', 9, 23, 28);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(14, 'National', 13, 24, 25);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(15, 'International', 13, 26, 27);
```

For the purpose of checking that everything is setup correctly, we can create a test method and output the contents of our category tree to see what it looks like. With a simple controller:

```
1.  <?php
2.  class CategoriesController extends AppController {
3.      var $name = 'Categories';
4.      function index() {
5.          $this->data = $this->Category->generatetreelist(null, null, null, '   ');
6.          debug ($this->data); die;
7.      }
8.  }
9.  ?>
```

and an even simpler model definition:

```
1.  <?php
2.  // app/models/category.php
3.  class Category extends AppModel {
4.      var $name = 'Category';
5.      var $actsAs = array('Tree');
6.  }
7.  ?>
```

We can check what our category tree data looks like by visiting `/categories` You should see something like this:

- My Categories
    - Fun
        - Sport
            - Surfing
            - Extreme knitting
        - Friends
            - Gerald
            - Gwendolyn
    - Work
        - Reports
            - Annual
            - Status
        - Trips
            - National
            - International

6.4.2.1 Adding data

In the previous section, we used existing data and checked that it looked hierarchal via the method `generatetreelist`. However, usually you would add your data in exactly the same way as you would for any model. For example:

```
1.      // pseudo controller code
2.      $data['Category']['parent_id'] =  3;
3.      $data['Category']['name'] =  'Skating';
4.      $this->Category->save($data);
```

When using the tree behavior its not necessary to do any more than set the parent_id, and the tree behavior will take care of the rest. If you don't set the parent_id, the tree behavior will add to the tree making your new addition a new top level entry:

```
// pseudo controller code
$data = array();
$data['Category']['name'] =  'Other People\'s Categories';
$this->Category->save($data);
```

Running the above two code snippets would alter your tree as follows:

- My Categories
    - Fun
        - Sport
            - Surfing
            - Extreme knitting
            - Skating **New**
        - Friends
            - Gerald
            - Gwendolyn
    - Work
        - Reports
            - Annual
            - Status
        - Trips
            - National
            - International
- Other People's Categories **New**

6.4.2.2 Modifying data

Modifying data is as transparent as adding new data. If you modify something, but do not change the parent_id field - the structure of your data will remain unchanged. For example:

```
1.      // pseudo controller code
2.      $this->Category->id = 5; // id of Extreme knitting
3.      $this->Category->save(array('name' =>'Extreme fishing'));
```

The above code did not affect the parent_id field - even if the parent_id is included in the data that is passed to save if the value doesn't change, neither does the data structure. Therefore the tree of data would now look like:

- My Categories
    - Fun
        - Sport
            - Surfing
            - Extreme fishing **Updated**
            - Skating
        - Friends
            - Gerald
            - Gwendolyn
    - Work
        - Reports
            - Annual
            - Status
        - Trips
            - National
            - International
- Other People's Categories

Moving data around in your tree is also a simple affair. Let's say that Extreme fishing does not belong under Sport, but instead should be located under Other People's Categories. With the following code:

```
1.      // pseudo controller code
```

```
2.      $this->Category->id = 5; // id of Extreme fishing
3.      $newParentId = $this->Category->field('id', array('name' => 'Other People\'s Categories'));
4.      $this->Category->save(array('parent_id' => $newParentId));
```

As would be expected the structure would be modified to:

- My Categories
  - Fun
    - Sport
      - Surfing
      - Skating
    - Friends
      - Gerald
      - Gwendolyn
  - Work
    - Reports
      - Annual
      - Status
    - Trips
      - National
      - International
- Other People's Categories
  - Extreme fishing **Moved**

6.4.2.3 Deleting data

The tree behavior provides a number of ways to manage deleting data. To start with the simplest example; let's say that the reports category is no longer useful. To remove it *and any children it may have* just call delete as you would for any model. For example with the following code:

```
1.      // pseudo controller code
```

```
2.          $this->Category->id = 10;
3.          $this->Category->delete();
```

The category tree would be modified as follows:

- My Categories
    - Fun
        - Sport
            - Surfing
            - Skating
        - Friends
            - Gerald
            - Gwendolyn
    - Work
        - Trips
            - National
            - International
- Other People's Categories
    - Extreme fishing

6.4.2.4 Querying and using your data

Using and manipulating hierarchical data can be a tricky business. In addition to the core find methods, with the tree behavior there are a few more tree-orientated permutations at your disposal.

Most tree behavior methods return and rely on data being sorted by the `lft` field. If you call `find()` and do not order by `lft`, or call a tree behavior method and pass a sort order, you may get undesirable results.

6.4.2.4.1 Children

The `children` method takes the primary key value (the id) of a row and returns the children, by default in the order they appear in the tree. The second optional parameter defines whether or not only direct children should be returned. Using the example data from the previous section:

```
1.      $allChildren = $this->Category->children(1); // a flat array with 11 items
2.      // -- or --
3.      $this->Category->id = 1;
4.      $allChildren = $this->Category->children(); // a flat array with 11 items
5.      // Only return direct children
6.      $directChildren = $this->Category->children(1, true); // a flat array with 2 items
```

If you want a recursive array use `find('threaded')`

Parameters for this function include:

- **$id**: The ID of the record to look up
- **$direct**: Set to true to return only the direct descendants
- **$fields**: Single string field name or array of fields to include in the return
- **$order**: SQL string of ORDER BY conditions
- **$limit**: SQL LIMIT statement
- **$page**: for accessing paged results
- **$recursive**: Number of levels deep for recursive associated Models

6.4.2.4.2 Counting children

As with the method `children`, `childCount` takes the primary key value (the id) of a row and returns how many children it has. The second optional parameter defines whether or not only direct children are counted. Using the example data from the previous section:

```
1.      $totalChildren = $this->Category->childCount(1); // will output 11
2.      // -- or --
3.      $this->Category->id = 1;
```

```
4.        $directChildren = $this->Category->childCount(); // will output 11
5.        // Only counts the direct descendants of this category
6.        $numChildren = $this->Category->childCount(1, true); // will output 2
```

6.4.2.4.3 generatetreelist

generatetreelist ($conditions=null, $keyPath=null, $valuePath=null, $spacer= '_', $recursive=null)

This method will return data similar to `find('list')`, with an indented prefix to show the structure of your data. Below is an example of what you can expect this method to return.

- `$conditions` - Uses the same conditional options as find().
- `$keyPath` - Path to the field to use for the key.
- `$valuePath` - Path to the field to use for the label.
- `$spacer` - The string to use in front of each item to indicate depth.
- `$recursive` - The number of levels deep to fetch associated records

All the parameters are optional, with the following defaults:

- `$conditions` = `null`
- `$keyPath` = Model's primary key
- `$valuePath` = Model's displayField
- `$spacer` = `'_'`
- `$recursive` = Model's recursive setting

```
1.        $treelist = $this->Category->generatetreelist();
```

Output:

```
array(
        [1] =>  "My Categories",
```

```
        [2] =>  "_Fun",
        [3] =>  "__Sport",
        [4] =>  "___Surfing",
        [16] => "___Skating",
        [6] =>  "__Friends",
        [7] =>  "___Gerald",
        [8] =>  "___Gwendolyn",
        [9] =>  "_Work",
        [13] => "__Trips",
        [14] => "___National",
        [15] => "___International",
        [17] => "Other People's Categories",
        [5] =>  "_Extreme fishing"
)
```

6.4.2.4.4 getparentnode

This convenience function will, as the name suggests, return the parent node for any node, or *false* if the node has no parent (its the root node). For example:

```
1.      $parent = $this->Category->getparentnode(2); //<- id for fun
2.      // $parent contains All categories
```

6.4.2.4.5 getpath

`getpath( $id = null, $fields = null, $recursive = null )`

The 'path' when refering to hierachial data is how you get from where you are to the top. So for example the path from the category "International" is:

- My Categories
    - ...
    - Work
        - Trips
            - ...
            - International

Using the id of "International" getpath will return each of the parents in turn (starting from the top).

```
1.      $parents = $this->Category->getpath(15);
```

```
// contents of $parents
array(
        [0] =>  array('Category' => array('id' => 1, 'name' => 'My Categories', ..)),
        [1] =>  array('Category' => array('id' => 9, 'name' => 'Work', ..)),
        [2] =>  array('Category' => array('id' => 13, 'name' => 'Trips', ..)),
        [3] =>  array('Category' => array('id' => 15, 'name' => 'International', ..)),
)
```

6.4.3 Advanced Usage

The tree behavior doesn't only work in the background, there are a number of specific methods defined in the behavior to cater for all your hierarchical data needs, and any unexpected problems that might arise in the process.

6.4.3.1 moveDown

Used to move a single node down the tree. You need to provide the ID of the element to be moved and a positive number of how many positions the node should be moved down. All child nodes for the specified node will also be moved.

Here is an example of a controller action (in a controller named Categories) that moves a specified node down the tree:

```
1.      function movedown($name = null, $delta = null) {
2.              $cat = $this->Category->findByName($name);
3.              if (empty($cat)) {
4.                  $this->Session->setFlash('There is no category named ' . $name);
5.                  $this->redirect(array('action' => 'index'), null, true);
6.              }
7.
8.              $this->Category->id = $cat['Category']['id'];
9.
```

```
10.                 if ($delta > 0) {
11.                     $this->Category->moveDown($this->Category->id, abs($delta));
12.                 } else {
13.                     $this->Session->setFlash('Please provide the number of positions the field should be moved down.');
14.                 }
15.
16.                 $this->redirect(array('action' => 'index'), null, true);
17.             }
```

For example, if you'd like to move the "Sport" category one position down, you would request: /categories/movedown/Sport/1.

6.4.3.2 moveUp

Used to move a single node up the tree. You need to provide the ID of the element to be moved and a positive number of how many positions the node should be moved up. All child nodes will also be moved.

Here's an example of a controller action (in a controller named Categories) that moves a node up the tree:

```
1.         function moveup($name = null, $delta = null){
2.                 $cat = $this->Category->findByName($name);
3.                 if (empty($cat)) {
4.                     $this->Session->setFlash('There is no category named ' . $name);
5.                     $this->redirect(array('action' => 'index'), null, true);
6.                 }
7.
8.                 $this->Category->id = $cat['Category']['id'];
9.
10.                 if ($delta > 0) {
11.                     $this->Category->moveUp($this->Category->id, abs($delta));
12.                 } else {
```

```
13.                 $this->Session->setFlash('Please provide a number of positions the category should be moved up.');
14.             }
15.
16.             $this->redirect(array('action' => 'index'), null, true);
17.
18.         }
```

For example, if you would like to move the category "Gwendolyn" up one position you would request /categories/moveup/Gwendolyn/1. Now the order of Friends will be Gwendolyn, Gerald.

6.4.3.3 removeFromTree

```
removeFromTree($id=null, $delete=false)
```

Using this method wil either delete or move a node but retain its sub-tree, which will be reparented one level higher. It offers more control than `delete()`, which for a model using the tree behavior will remove the specified node and all of its children.

Taking the following tree as a starting point:

- My Categories
    - Fun
        - Sport
            - Surfing
            - Extreme knitting
            - Skating

Running the following code with the id for 'Sport'

```
1.      $this->Node->removeFromTree($id);
```

The Sport node will be become a top level node:

- My Categories
    - Fun
        - Surfing
        - Extreme knitting
        - Skating
- Sport **Moved**

This demonstrates the default behavior of `removeFromTree` of moving the node to have no parent, and re-parenting all children.

If however the following code snippet was used with the id for 'Sport'

```
1.      $this->Node->removeFromTree($id,true);
```

The tree would become

- My Categories
    - Fun
        - Surfing
        - Extreme knitting
        - Skating

This demonstrates the alternate use for `removeFromTree`, the children have been reparented and 'Sport' has been deleted.

6.4.3.4 reorder

```
reorder ( array('id' => null, 'field' => $Model->displayField, 'order' => 'ASC', 'verify' => true) )
```

Reorders the nodes (and child nodes) of the tree according to the field and direction specified in the parameters. This method does not change the parent of any node.

```
1.      $model->reorder(array(
2.          'id' => ,    //id of record to use as top node for reordering, default: $Model->id
3.          'field' => , //which field to use in reordering, default: $Model->displayField
4.          'order' => , //direction to order, default: 'ASC'
5.          'verify' =>  //whether or not to verify the tree before reorder, default: true
6.      ));
```

If you have saved your data or made other operations on the model, you might want to set `$model->id = null` before calling `reorder`. Otherwise only the current node and it's children will be reordered.

6.4.4 Data Integrity

Due to the nature of complex self referential data structures such as trees and linked lists, they can occasionally become broken by a careless call. Take heart, for all is not lost! The Tree Behavior contains several previously undocumented features designed to recover from such situations.

6.4.4.1 Recover

`recover(&$model, $mode = 'parent', $missingParentAction = null)`

The `mode` parameter is used to specify the source of info that is valid/correct. The opposite source of data will be populated based upon that source of info. E.g. if the MPTT fields are corrupt or empty, with the `$mode 'parent'` the values of the `parent_id` field will be used to populate the left and right fields. The `missingParentAction` parameter only applies to "parent" mode and determines what to do if the parent field contains an id that is not present.

Available `$mode` options:

- `'parent'` - use the existing `parent_id`'s to update the `lft` and `rght` fields
- `'tree'` - use the existing `lft` and `rght` fields to update `parent_id`

Available `missingParentActions` options when using `mode='parent'`:

- `null` - do nothing and carry on
- `'return'` - do nothing and return
- `'delete'` - delete the node
- `int` - set the parent_id to this id

```
1.      // Rebuild all the left and right fields based on the parent_id
2.      $this->Category->recover();
3.      // or
4.      $this->Category->recover('parent');
5.
6.      // Rebuild all the parent_id's based on the lft and rght fields
7.      $this->Category->recover('tree');
```

6.4.4.2 Reorder

`reorder(&$model, $options = array())`

Reorders the nodes (and child nodes) of the tree according to the field and direction specified in the parameters. This method does not change the parent of any node.

Reordering affects all nodes in the tree by default, however the following options can affect the process:

- `'id'` - only reorder nodes below this node.
- `'field'` - field to use for sorting, default is the `displayField` for the model.
- `'order'` - `'ASC'` for ascending, `'DESC'` for descending sort.
- `'verify'` - whether or not to verify the tree prior to resorting.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
1.      array(
2.          'id' => null,
```

```
3.            'field' => $model->displayField,
4.            'order' => 'ASC',
5.            'verify' => true
6.        )
```

6.4.4.3 Verify

verify(&$model)

Returns `true` if the tree is valid otherwise an array of errors, with fields for type, incorrect index and message.

Each record in the output array is an array of the form (type, id, message)

- `type` is either `'index'` or `'node'`
- `'id'` is the id of the erroneous node.
- `'message'` depends on the error

```
1.            $this->Categories->verify();
```

Example output:

```
Array
(
    [0] => Array
        (
            [0] => node
            [1] => 3
            [2] => left and right values identical
        )
    [1] => Array
        (
            [0] => node
            [1] => 2
            [2] => The parent node 999 doesn't exist
```

```
        )
    [10] => Array
        (
            [0] => index
            [1] => 123
            [2] => missing
        )
    [99] => Array
        (
            [0] => node
            [1] => 163
            [2] => left greater than right
        )

)
```

7 Core Helpers

Helpers are the component-like classes for the presentation layer of your application. They contain presentational logic that is shared between many views, elements, or layouts.

This section describes each of the helpers that come with CakePHP such as Form, Html, JavaScript and RSS.

Read Helpers to learn more about helpers and how you can build your own helpers.

7.1 AJAX

Both the JavascriptHelper and the AjaxHelper are deprecated, and the JsHelper + HtmlHelper should be used in their place. See The Migration Guide

The AjaxHelper utilizes the ever-popular Prototype and script.aculo.us libraries for Ajax operations and client side effects. To use the AjaxHelper, you must have a current version of the JavaScript libraries from www.prototypejs.org and http://script.aculo.us placed in /app/webroot/js/. In addition, you must include the Prototype and script.aculo.us JavaScript libraries in any layouts or views that require AjaxHelper functionality.

You'll need to include the Ajax and Javascript helpers in your controller:

```
1.      class WidgetsController extends AppController {
2.          var $name = 'Widgets';
3.          var $helpers = array('Html','Ajax','Javascript');
4.      }
```

Once you have the javascript helper included in your controller, you can use the javascript helper link() method to include Prototype and Scriptaculous:

```
1.      echo $html->script('prototype');
2.      echo $html->script('scriptaculous');
```

Now you can use the Ajax helper in your view:

```
1.      $ajax->whatever();
```

If the RequestHandler Component is included in the controller then CakePHP will automatically apply the Ajax layout when an action is requested via AJAX

```
1.      class WidgetsController extends AppController {
2.          var $name = 'Widgets';
3.          var $helpers = array('Html','Ajax','Javascript');
4.          var $components = array( 'RequestHandler' );
5.      }
```

7.1.1 AjaxHelper Options

Most of the methods of the AjaxHelper allow you to supply an $options array. You can use this array to configure how the AjaxHelper behaves. Before we cover the specific methods in the helper, let's look at the different options available through this special array. You'll want to refer to this section as you start using the methods in the AjaxHelper later on.

7.1.1.1 General Options

$option keys	Description
$options['evalScripts']	Determines if script tags in the returned content are evaluated. Set to *true* by default.
$options['frequency']	The number of seconds between interval based checks.
$options['indicator']	The DOM id of an element to show while a request is loading and to hide when a request is completed.
$options['position']	To insert rather than replace, use this option to specify an insertion position of *top*, *bottom*, *after*, or *before*.
$options['update']	The id of the DOM element to be updated with returned content.
$options['url']	The url of the controller/action that you want to call.
$options['type']	Indicate whether the request should be 'synchronous' or 'asynchronous' (default).
$options['with']	A URL-encoded string which will be added to the URL for get methods or in to the post body for any other method. Example: x=1&foo=bar&y=2. The parameters will be available in $this->params['form'] or available in $this->data depending on formatting. For more information see the Prototype Serialize method.

7.1.1.2 Callback Options

Callback options allow you to call JavaScript functions at specific points in the request process. If you're looking for a way to inject a bit of logic before, after, or during your AjaxHelper operations, use these callbacks to set things up.

$options keys	Description
$options['condition']	JavaScript code snippet that needs to evaluate to *true* before request is initiated.
$options['before']	Executed before request is made. A common use for this callback is to enable the visibility of a progress indicator.
$options['confirm']	Text to display in a JavaScript confirmation alert before proceeding.
$options['loading']	Callback code to be executed while data is being fetched from server.
$options['after']	JavaScript called immediately after request has run; fires before the $options['loading'] callback runs.

$options['loaded']	Callback code to be executed when the remote document has been received by client.
$options['interactive']	Called when the user can interact with the remote document, even though it has not finished loading.
$options['complete']	JavaScript callback to be run when XMLHttpRequest is complete.

7.1.2 Methods

7.1.2.1 link

link(string $title, mixed $href, array $options, string $confirm, boolean $escapeTitle)

Returns a link to a remote action defined by `$options['url']` or `$href` that's called in the background using XMLHttpRequest when the link is clicked. The result of that request can then be inserted into a DOM object whose id can be specified with `$options['update']`.

If `$options['url']` is blank the href is used instead

Example:

```
1.      <div id="post">
2.      </div>
3.      <?php echo $ajax->link(
4.          'View Post',
5.          array( 'controller' => 'posts', 'action' => 'view', 1 ),
6.          array( 'update' => 'post' )
7.      );
8.      ?>
```

By default, these remote requests are processed asynchronously during which various callbacks can be triggered

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'View Post',
    array( 'controller' => 'posts', 'action' => 'post', 1 ),
    array( 'update' => 'post', 'complete' => 'alert( "Hello World" )'  )
);
?>
```

To use synchronous processing specify `$options['type'] = 'synchronous'`.

To automatically set the ajax layout include the *RequestHandler* component in your controller

By default the contents of the target element are replaced. To change this behaviour set the `$options['position']`

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'View Post',
    array( 'controller' => 'posts', 'action' => 'view', 1),
    array( 'update' => 'post', 'position' => 'top'  )
);
?>
```

`$confirm` can be used to call up a JavaScript confirm() message before the request is run. Allowing the user to prevent execution.

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'Delete Post',
    array( 'controller' => 'posts', 'action' => 'delete', 1 ),
    array( 'update' => 'post' ),
    'Do you want to delete this post?'
);
?>
```

7.1.2.2 remoteFunction

```
remoteFunction(array $options);
```

This function creates the JavaScript needed to make a remote call. It is primarily used as a helper for link(). This is not used very often unless you need to generate some custom scripting.

The `$options` for this function are the same as for the `link` method

Example:

```
<div id="post">
</div>
<script type="text/javascript">
<?php echo $ajax->remoteFunction(
    array(
        'url' => array( 'controller' => 'posts', 'action' => 'view', 1 ),
        'update' => 'post'
    )
); ?>
</script>
```

It can also be assigned to HTML Event Attributes:

```
1.      <?php
2.          $remoteFunction = $ajax->remoteFunction(
3.              array(
4.              'url' => array( 'controller' => 'posts', 'action' => 'view', 1 ),
5.              'update' => 'post' )
6.          );
7.      ?>
8.      <div id="post" onmouseover="<?php echo $remoteFunction; ?>" >
9.      Mouse Over This
10.     </div>
```

If `$options['update']` is not passed, the browser will ignore the server response.

7.1.2.3 remoteTimer

`remoteTimer(array $options)`

Periodically calls the action at `$options['url']`, every `$options['frequency']` seconds. Usually used to update a specific div (specified by `$options['update']`) with the result of the remote call. Callbacks can be used.

`remoteTimer` is the same as the `remoteFunction` except for the extra `$options['frequency']`

Example:

```
1.      <div id="post">
2.      </div>
3.      <?php
4.      echo $ajax->remoteTimer(
```

```
            array(
            'url' => array( 'controller' => 'posts', 'action' => 'view', 1 ),
            'update' => 'post', 'complete' => 'alert( "request completed" )',
            'position' => 'bottom', 'frequency' => 5
            )
        );
        ?>
```

The default `$options['frequency']` is 10 seconds

7.1.2.4 form

`form(string $action, string $type, array $options)`

Returns a form tag that submits to $action using XMLHttpRequest instead of a normal HTTP request via $type ('post' or 'get'). Otherwise, form submission will behave exactly like normal: data submitted is available at $this->data inside your controllers. If $options['update'] is specified, it will be updated with the resulting document. Callbacks can be used.

The options array should include the model name e.g.

```
        $ajax->form('edit','post',array('model'=>'User','update'=>'UserInfoDiv'));
```

Alternatively, if you need to cross post to another controller from your form:

```
        $ajax->form(array('type' => 'post',
            'options' => array(
                'model'=>'User',
                'update'=>'UserInfoDiv',
                'url' => array(
                    'controller' => 'comments',
                    'action' => 'edit'
                )
            )
```

```
10.         ));
```

You should not use the `$ajax->form()` and `$ajax->submit()` in the same form. If you want the form validation to work properly use the `$ajax->submit()` method as shown below.

7.1.2.5 submit

`submit(string $title, array $options)`

Returns a submit button that submits the form to `$options['url']` and updates the div specified in `$options['update']`

```
1.          <div id='testdiv'>
2.          <?php
3.          echo $form->create('User');
4.          echo $form->input('email');
5.          echo $form->input('name');
6.          echo  $ajax->submit('Submit',  array('url'=>  array('controller'=>'users',  'action'=>'add'),  'update'  =>
'testdiv'));
7.          echo $form->end();
8.          ?>
9.          </div>
```

Use the `$ajax->submit()` method if you want form validation to work properly. i.e. You want the messages you specify in your validation rules to show up correctly.

7.1.2.6 observeField

`observeField(string $fieldId, array $options)`

Observes the field with the DOM id specified by $field_id (every $options['frequency'] seconds) and makes an XMLHttpRequest when its contents have changed.

```
1.      <?php echo $form->create( 'Post' ); ?>
2.      <?php $titles = array( 1 => 'Tom', 2 => 'Dick', 3 => 'Harry' ); ?>
3.      <?php echo $form->input( 'title', array( 'options' => $titles ) ) ?>
4.      </form>
5.      <?php
6.      echo $ajax->observeField( 'PostTitle',
7.          array(
8.              'url' => array( 'action' => 'edit' ),
9.              'frequency' => 0.2,
10.         )
11.     );
12.     ?>
```

`observeField` uses the same options as `link`

The field to send up can be set using `$options['with']`. This defaults to `Form.Element.serialize('$fieldId')`. Data submitted is available at `$this->data` inside your controllers. Callbacks can be used with this function.

To send up the entire form when the field changes use `$options['with'] = Form.serialize( $('Form ID') )`

7.1.2.7 observeForm

observeForm(string $form_id, array $options)

Similar to observeField(), but operates on an entire form identified by the DOM id $form_id. The supplied $options are the same as observeField(), except the default value of the $options['with'] option evaluates to the serialized (request string) value of the form.

7.1.2.8 autoComplete

autoComplete(string $fieldId, string $url, array $options)

Renders a text field with $fieldId with autocomplete. The remote action at $url should return a suitable list of autocomplete terms. Often an unordered list is used for this. First, you need to set up a controller action that fetches and organizes the data you'll need for your list, based on user input:

```
1.      function autoComplete() {
2.          //Partial strings will come from the autocomplete field as
3.          //$this->data['Post']['subject']
4.          $this->set('posts', $this->Post->find('all', array(
5.                      'conditions' => array(
6.                          'Post.subject LIKE' => $this->data['Post']['subject'].'%'
7.                      ),
8.                      'fields' => array('subject')
9.          )));
10.         $this->layout = 'ajax';
11.     }
```

Next, create `app/views/posts/auto_complete.ctp` that uses that data and creates an unordered list in (X)HTML:

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

Finally, utilize autoComplete() in a view to create your auto-completing form field:

```
1.      <?php echo $form->create('User', array('url' => '/users/index')); ?>
2.          <?php echo $ajax->autoComplete('Post.subject', '/posts/autoComplete')?>
```

```
3.        <?php echo $form->end('View Post')?>
```

Once you've got the autoComplete() call working correctly, use CSS to style the auto-complete suggestion box. You might end up using something similar to the following:

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

7.1.2.9 isAjax

isAjax()

Allows you to check if the current request is a Prototype Ajax request inside a view. Returns a boolean. Can be used for presentational logic to show/hide blocks of content.

7.1.2.10 drag & drop

drag(string $id, array $options)

Makes a Draggable element out of the DOM element specified by $id. For more information on the parameters accepted in $options see http://github.com/madrobby/scriptaculous/wikis/draggable.

Common options might include:

$options keys	Description
$options['handle']	Sets whether the element should only be draggable by an embedded handle. The value must be an element reference or element id or a string referencing a CSS class value. The first child/grandchild/etc. element found within the element that has this CSS class

	value will be used as the handle.
$options['revert']	If set to true, the element returns to its original position when the drags ends. Revert can also be an arbitrary function reference, called when the drag ends.
$options['constraint']	Constrains the drag to either 'horizontal' or 'vertical', leave blank for no constraints.

```
drop(string $id, array $options)
```

Makes the DOM element specified by $id able to accept dropped elements. Additional parameters can be specified with $options. For more information see http://github.com/madrobby/scriptaculous/wikis/droppables.

Common options might include:

$options keys	Description
$options['accept']	Set to a string or javascript array of strings describing CSS classes that the droppable element will accept. The drop element will only accept elements of the specified CSS classes.
$options['containment']	The droppable element will only accept the dragged element if it is contained in the given elements (element ids). Can be a string or a javascript array of id references.
$options['overlap']	If set to 'horizontal' or 'vertical', the droppable element will only react to a draggable element if it is overlapping the droparea by more than 50% in the given axis.
$options['onDrop']	A javascript call back that is called when the dragged element is dropped on the droppable element.

```
dropRemote(string $id, array $options)
```

Makes a drop target that creates an XMLHttpRequest when a draggable element is dropped on it. The $options array for this function are the same as those specified for drop() and link().

7.1.2.11 slider

```
slider(string $id, string $track_id, array $options)
```

Creates a directional slider control. For more information see http://wiki.github.com/madrobby/scriptaculous/slider.

Common options might include:

$options keys	Description
$options['axis']	Sets the direction the slider will move in. 'horizontal' or 'vertical'. Defaults to horizontal
$options['handleImage']	The id of the image that represents the handle. This is used to swap out the image src with disabled image src when the slider is enabled. Used in conjunction with handleDisabled.
$options['increment']	Sets the relationship of pixels to values. Setting to 1 will make each pixel adjust the slider value by one.
$options['handleDisabled']	The id of the image that represents the disabled handle. This is used to change the image src when the slider is disabled. Used in conjunction handleImage.
$options['change'] $options['onChange']	JavaScript callback fired when the slider has finished moving, or has its value changed. The callback function receives the slider's current value as a parameter.
$options['slide'] $options['onSlide']	JavaScript callback that is called whenever the slider is moved by dragging. It receives the slider's current value as a parameter.

7.1.2.12 editor

```
editor(string $id, string $url, array $options)
```

Creates an in-place editor at DOM id. The supplied `$url` should be an action that is responsible for saving element data. For more information and demos see http://github.com/madrobby/scriptaculous/wikis/ajax-inplaceeditor.

Common options might include:

$options keys	Description
`$options['collection']`	Activate the 'collection' mode of in-place editing. $options['collection'] takes an array which is turned into options for the select. To learn more about collection see http://github.com/madrobby/scriptaculous/wikis/ajax-inplacecollectioneditor.
`$options['callback']`	A function to execute before the request is sent to the server. This can be used to format the information sent to the server. The signature is `function(form, value)`
`$options['okText']`	Text of the submit button in edit mode
`$options['cancelText']`	The text of the link that cancels editing
`$options['savingText']`	The text shown while the text is sent to the server
`$options['formId']`	
`$options['externalControl']`	
`$options['rows']`	The row height of the input field
`$options['cols']`	The number of columns the text area should span
`$options['size']`	Synonym for ‘cols’ when using single-line
`$options['highlightcolor']`	The highlight color
`$options['highlightendcolor']`	The color which the highlight fades to
`$options['savingClassName']`	
`$options['formClassName']`	
`$options['loadingText']`	
`$options['loadTextURL']`	

Example

```
1.      <div id="in_place_editor_id">Text To </div>
2.      <?php
3.      echo $ajax->editor(
4.          "in_place_editor_id",
5.          array(
6.              'controller' => 'Posts',
7.              'action' => 'update_title',
8.              $id
9.          ),
10.         array()
11.     );
12.     ?>
```

7.1.2.13 sortable

```
sortable(string $id, array $options)
```

Makes a list or group of floated objects contained by $id sortable. The options array supports a number of parameters. To find out more about sortable see http://wiki.github.com/madrobby/scriptaculous/sortable.

Common options might include:

$options keys	Description
$options['tag']	Indicates what kind of child elements of the container will be made sortable. Defaults to 'li'.
$options['only']	Allows for further filtering of child elements. Accepts a CSS class.
$options['overlap']	Either 'vertical' or 'horizontal'. Defaults to vertical.
$options['constraint']	Restrict the movement of the draggable elements. accepts 'horizontal' or 'vertical'. Defaults to vertical.

$options['handle']	Makes the created Draggables use handles, see the handle option on Draggables.
$options['onUpdate']	Called when the drag ends and the Sortable's order is changed in any way. When dragging from one Sortable to another, the callback is called once on each Sortable.
$options['hoverclass']	Give the created droppable a hoverclass.
$options['ghosting']	If set to true, dragged elements of the sortable will be cloned and appear as a ghost, instead of directly manipulating the original element.

7.2 Cache

The Cache helper assists in caching entire layouts and views, saving time repetitively retrieving data. View Caching in Cake temporarily stores parsed layouts and views with the storage engine of choice. It should be noted that the Cache helper works quite differently than other helpers. It does not have methods that are directly called. Instead a view is marked with cache tags indicating which blocks of content should not be cached.

When a URL is requested, Cake checks to see if that request string has already been cached. If it has, the rest of the url dispatching process is skipped. Any nocache blocks are processed normally and the view is served. This creates a big savings in processing time for each request to a cached URL as minimal code is executed. If Cake doesn't find a cached view, or the cache has expired for the requested URL it continues to process the request normally.

7.2.1 General Caching

Caching is intended to be a means of temporary storage to help reduce load on the server. For example you could store the results of a time-expensive database query so that it is not required to run on every page load.

With this in mind caching is not permanent storage and should never be used to permanently store anything. And only cache things that can be regenerated when needed.

7.2.2 Cache Engines in Cake

New since 1.2 are several cache engines or cache backends. These interface transparently with the cache helper, allowing you to store view caches in a multitude of media without worrying about the specifics of that media. The choice of cache engine is controlled through the app/config/core.php config file.

Most options for each caching engine are listed in the core.php config file and more detailed information on each caching engine can be found in the Caching Section.

File	The File Engine is the default caching engine used by cake. It writes flat files to the filesystem and it has several optional parameters but works well with the defaults.		
APC	The APC engine implements the Alternative PHP Cache opcode Cacher. Like XCache, this engine caches the compiled PHP opcode.		
XCache	The XCache caching engine is functionally similar to APC other than it implements the XCache opcode caching engine. It requires the entry of a user and password to work properly.	**Memcache**	The Memcache engine works with a memcaching server allowing you to create a cache object in system memory. More information on memcaching can be found on php.net and memcached

7.2.3 Cache Helper Configuration

View Caching and the Cache Helper have several important configuration elements. They are detailed below.

To use the cache helper in any view or controller, you must first uncomment and set Configure::Cache.check to true in `core.php` of your app/config folder. If this is not set to true, then the cache will not be checked or created.

7.2.4 Caching in the Controller

Any controllers that utilize caching functionality need to include the CacheHelper in their $helpers array.

```
1.      var $helpers = array('Cache');
```

You also need to indicate which actions need caching, and how long each action will be cached. This is done through the $cacheAction variable in your controllers. $cacheAction should be set to an array which contains the actions you want cached, and the duration in seconds you want those views cached. The time value can be expressed in a strtotime() format. (ie. "1 hour", or "3 minutes").

Using the example of an ArticlesController, that receives a lot of traffic that needs to be cached.

Cache frequently visited Articles for varying lengths of time

```
1.      var $cacheAction = array(
2.          'view/23' => 21600,
3.          'view/48' => 36000,
4.          'view/52'  => 48000
5.      );
```

Remember to use your routes in the $cacheAction if you have any.

Cache an entire action in this case a large listing of articles

```
1.      var $cacheAction = array(
2.          'archives/' => '60000'
3.      );
```

Cache every action in the controller using a strtotime() friendly time to indicate Controller wide caching time.

```
1.      var $cacheAction = "1 hour";
```

You can also enable controller/component callbacks for cached views created with `CacheHelper`. To do so you must use the array format for `$cacheAction` and create an array like the following:

```
1.      var $cacheAction = array(
2.          'view' => array('callbacks' => true, 'duration' => 21600),
3.          'add' => array('callbacks' => true, 'duration' => 36000),
4.          'index'  => array('callbacks' => true, 'duration' => 48000)
5.      );
```

By setting `callbacks => true` you tell CacheHelper that you want the generated files to create the components and models for the controller. As well as, fire the component initialize, controller beforeFilter, and component startup callbacks.

`callbacks => true` partly defeats the purpose of caching. This is also the reason it is disabled by default.

7.2.5 Marking Non-Cached Content in Views

There will be times when you don't want an *entire* view cached. For example, certain parts of the page may look different whether a user is currently logged in or browsing your site as a guest.

To indicate blocks of content that are *not* to be cached, wrap them in `<cake:nocache> </cake:nocache>` like so:

```
1.      <cake:nocache>
2.      <?php if ($session->check('User.name')) : ?>
3.          Welcome, <?php echo $session->read('User.name')?>.
4.      <?php else: ?>
5.          <?php echo $html->link('Login', 'users/login')?>
6.      <?php endif; ?>
7.      </cake:nocache>
```

It should be noted that once an action is cached, the controller method for the action will not be called - otherwise what would be the point of caching the page. Therefore, it is not possible to wrap `<cake:nocache> </cake:nocache>` around variables which are set from the controller as they will be *null*.

7.2.6 Clearing the Cache

It is important to remember that the Cake will clear a cached view if a model used in the cached view is modified. For example, if a cached view uses data from the Post model, and there has been an INSERT, UPDATE, or DELETE query made to a Post, the cache for that view is cleared, and new content is generated on the next request.

If you need to manually clear the cache, you can do so by calling Cache::clear(). This will clear **all** cached data, excluding cached view files. If you need to clear the cached view files, use `clearCache()`.

7.3 Form

The FormHelper is a new addition to CakePHP. Most of the heavy lifting in form creation is now done using this new class, rather than (now deprecated) methods in the HtmlHelper. The FormHelper focuses on creating forms quickly, in a way that will streamline validation, re-population and layout. The FormHelper is also flexible - it will do almost everything for you automagically, or you can use specific methods to get only what you need.

7.3.1 Creating Forms

The first method you'll need to use in order to take advantage of the FormHelper is `create()`. This special method outputs an opening form tag.

```
create(string $model = null, array $options = array())
```

All parameters are optional. If `create()` is called with no parameters supplied, it assumes you are building a form that submits to the current controller, via either the `add()` or `edit()` action. The default method for form submission is POST. The form element is also returned with a DOM ID. The ID is generated using the name of the model, and the name of the controller action, CamelCased. If I were to call `create()` inside a UsersController view, I'd see something like the following output in the rendered view:

```
1.      <form id="UserAddForm" method="post" action="/users/add">
```

You can also pass `false` for `$model`. This will place your form data into the array: `$this->data` (instead of in the sub-array: `$this->data['Model']`). This can be handy for short forms that may not represent anything in your database.

The `create()` method allows us to customize much more using the parameters, however. First, you can specify a model name. By specifying a model for a form, you are creating that form's *context*. All fields are assumed to belong to this model (unless otherwise specified), and all models referenced are assumed to be associated with it. If you do not specify a model, then it assumes you are using the default model for the current controller.

```
1.      <?php echo $this->Form->create('Recipe'); ?>
2.
3.      //Output:
4.      <form id="RecipeAddForm" method="post" action="/recipes/add">
```

This will POST the form data to the `add()` action of RecipesController. However, you can also use the same logic to create an edit form. The FormHelper uses the `$this->data` property to automatically detect whether to create an add or edit form. If `$this->data` contains an array element named after the form's model, and that array contains a non-empty value of the model's primary key, then the FormHelper will create an edit form for that record. For example, if we browse to `http://site.com/recipes/edit/5`, we might get the following:

```
1.  // controllers/recipes_controller.php:
2.  <?php
3.  function edit($id = null) {
4.      if (empty($this->data)) {
5.          $this->data = $this->Recipe->findById($id);
6.      } else {
7.          // Save logic goes here
8.      }
9.  }
10. ?>
11.
12. // views/recipes/edit.ctp:
13. // Since $this->data['Recipe']['id'] = 5, we should get an edit form
14. <?php echo $this->Form->create('Recipe'); ?>
15.
16. //Output:
17. <form id="RecipeForm" method="post" action="/recipes/edit/5">
18. <input type="hidden" name="_method" value="PUT" />
```

Since this is an edit form, a hidden input field is generated to override the default HTTP method.

The `$options` array is where most of the form configuration happens. This special array can contain a number of different key-value pairs that affect the way the form tag is generated.

7.3.1.1 $options['type']

This key is used to specify the type of form to be created. Valid values include ‘post’, ‘get’, ‘file’, ‘put’ and ‘delete’.

Supplying either ‘post’ or ‘get’ changes the form submission method accordingly.

```
1.      <?php echo $this->Form->create('User', array('type' => 'get')); ?>
2.
3.      //Output:
4.      <form id="UserAddForm" method="get" action="/users/add">
```

Specifying ‘file’ changes the form submission method to ‘post’, and includes an enctype of “multipart/form-data” on the form tag. This is to be used if there are any file elements inside the form. The absence of the proper enctype attribute will cause the file uploads not to function.

```
1.      <?php echo $this->Form->create('User', array('type' => 'file')); ?>
2.
3.      //Output:
4.      <form id="UserAddForm" enctype="multipart/form-data" method="post" action="/users/add">
```

When using ‘put’ or ‘delete’, your form will be functionally equivalent to a 'post' form, but when submitted, the HTTP request method will be overridden with 'PUT' or 'DELETE', respectively. This allows CakePHP to emulate proper REST support in web browsers.

7.3.1.2 $options[‘action’]

The action key allows you to point the form to a specific action in your current controller. For example, if you’d like to point the form to the login() action of the current controller, you would supply an $options array like the following:

```
1.      <?php echo $this->Form->create('User', array('action' => 'login')); ?>
2.
3.      //Output:
4.      <form id="UserLoginForm" method="post" action="/users/login">
5.      </form>
```

7.3.1.3 $options['url']

If the desired form action isn't in the current controller, you can specify a URL for the form action using the 'url' key of the $options array. The supplied URL can be relative to your CakePHP application, or can point to an external domain.

```
2.      <?php echo $this->Form->create(null, array('url' => '/recipes/add')); ?>
3.      // or
4.      <?php echo $this->Form->create(null, array('url' => array('controller' => 'recipes', 'action' => 'add')));
?>
5.
6.      //Output:
7.      <form method="post" action="/recipes/add">
8.
9.      <?php echo $this->Form->create(null, array(
10.         'url' => 'http://www.google.com/search',
11.         'type' => 'get'
12.     )); ?>
13.
14.     //Output:
15.     <form method="get" action="http://www.google.com/search">
```

Also check HtmlHelper::url method for more examples of different types of urls.

7.3.1.4 $options['default']

If 'default' has been set to boolean false, the form's submit action is changed so that pressing the submit button does not submit the form. If the form is meant to be submitted via AJAX, setting 'default' to false suppresses the form's default behavior so you can grab the data and submit it via AJAX instead.

7.3.1.5 $options['inputDefaults']

You can declare a set of default options for input() with the inputDefaults key to customize your default input creation.

```
echo $this->Form->create('User', array(
    'inputDefaults' => array(
        'label' => false,
        'div' => false
    )
));
```

All inputs created from that point forward would inherit the options declared in inputDefaults. You can override the defaultOptions by declaring the option in the input() call.

```
echo $this->Form->input('password'); // No div, no label
echo $this->Form->input('username', array('label' => 'Username')); // has a label element
```

7.3.2 Closing the Form

The FormHelper also includes an `end()` method that completes the form markup. Often, `end()` only outputs a closing form tag, but using `end()` also allows the FormHelper to insert needed hidden form elements other methods may be depending on.

```
<?php echo $this->Form->create(); ?>

<!-- Form elements go here -->

<?php echo $this->Form->end(); ?>
```

If a string is supplied as the first parameter to `end()`, the FormHelper outputs a submit button named accordingly along with the closing form tag.

```
<?php echo $this->Form->end('Finish'); ?>
```

Will output:

```
<div class="submit">
    <input type="submit" value="Finish" />
</div>
</form>
```

You can specify detail settings by passing an array to `end()`.

```php
<?php
$options = array(
    'label' => 'Update',
    'value' => 'Update!',
    'div' => array(
        'class' => 'glass-pill',
    )
);
echo $this->Form->end($options);
```

Will output:

```
<div class="glass-pill"><input type="submit" value="Update!" name="Update"></div>
```

See the API for further details.

7.3.3 Automagic Form Elements

First, let's look at some of the more automatic form creation methods in the FormHelper. The main method we'll look at is input(). This method will automatically inspect the model field it has been supplied in order to create an appropriate input for that field.

input(string $fieldName, array $options = array())

Column Type	Resulting Form Field

string (char, varchar, etc.)	text
boolean, tinyint(1)	checkbox
text	textarea
text, with name of password, passwd, or psword	password
date	day, month, and year selects
datetime, timestamp	day, month, year, hour, minute, and meridian selects
time	hour, minute, and meridian selects

For example, let's assume that my User model includes fields for a username (varchar), password (varchar), approved (datetime) and quote (text). I can use the input() method of the FormHelper to create appropriate inputs for all of these form fields.

```
        <?php echo $this->Form->create(); ?>

            <?php
                echo $this->Form->input('username');    //text
                echo $this->Form->input('password');    //password
                echo $this->Form->input('approved');    //day, month, year, hour, minute, meridian
                echo $this->Form->input('quote');       //textarea
            ?>

        <?php echo $this->Form->end('Add'); ?>
```

A more extensive example showing some options for a date field:

```
            echo $this->Form->input('birth_dt', array( 'label' => 'Date of birth'
                                        , 'dateFormat' => 'DMY'
                                        , 'minYear' => date('Y') - 70
                                        , 'maxYear' => date('Y') - 18 ));
```

Besides the specific input options found below you can specify any html attribute (for instance onfocus). For more information on $options and $htmlAttributes see HTML Helper.

And to round off, here's an example for creating a hasAndBelongsToMany select. Assume that User hasAndBelongsToMany Group. In your controller, set a camelCase plural variable (group -> groups in this case, or ExtraFunkyModel -> extraFunkyModels) with the select options. In the controller action you would put the following:

```
    $this->set('groups', $this->User->Group->find('list'));
```

And in the view a multiple select can be expected with this simple code:

```
    echo $this->Form->input('Group');
```

If you want to create a select field while using a belongsTo- or hasOne-Relation, you can add the following to your Users-controller (assuming your User belongsTo Group):

```
    $this->set('groups', $this->User->Group->find('list'));
```

Afterwards, add the following to your form-view:

```
    echo $this->Form->input('group_id');
```

If your model name consists of two or more words, e.g., "UserGroup", when passing the data using set() you should name your data in a pluralised and camelCased format as follows:

```
        $this->set('userGroups', $this->UserGroup->find('list'));
        // or
        $this->set('reallyInappropriateModelNames', $this->ReallyInappropriateModelName->find('list'));
```

7.3.3.1 Field naming convention

The Form helper is pretty smart. Whenever you specify a field name with the form helper methods, it'll automatically use the current model name to build an input with a format like the following:

```
        <input type="text" id="ModelnameFieldname" name="data[Modelname][fieldname]">
```

You can manually specify the model name by passing in Modelname.fieldname as the first parameter.

```
16.     echo $this->Form->input('Modelname.fieldname');
```

If you need to specify multiple fields using the same field name, thus creating an array that can be saved in one shot with saveAll(), use the following convention:

```
        <?php
           echo $this->Form->input('Modelname.0.fieldname');
           echo $this->Form->input('Modelname.1.fieldname');
        ?>

        <input type="text" id="Modelname0Fieldname" name="data[Modelname][0][fieldname]">
        <input type="text" id="Modelname1Fieldname" name="data[Modelname][1][fieldname]">
```

7.3.3.2 $options['type']

You can force the type of an input (and override model introspection) by specifying a type. In addition to the field types found in the table above, you can also create 'file', and 'password' inputs.

```
<?php echo $this->Form->input('field', array('type' => 'file')); ?>

Output:

<div class="input">
    <label for="UserField">Field</label>
    <input type="file" name="data[User][field]" value="" id="UserField" />
</div>
```

7.3.3.3 $options['before'], $options['between'], $options['separator'] and $options['after']

Use these keys if you need to inject some markup inside the output of the input() method.

```
<?php echo $this->Form->input('field', array(
    'before' => '--before--',
    'after' => '--after--',
    'between' => '--between---'
));?>

Output:

<div class="input">
--before--
<label for="UserField">Field</label>
--between---
<input name="data[User][field]" type="text" value="" id="UserField" />
--after--
</div>
```

For radio type input the 'separator' attribute can be used to inject markup to separate each input/label pair.

```
<?php echo $this->Form->input('field', array(
    'before' => '--before--',
    'after' => '--after--',
    'between' => '--between---',
    'separator' => '--separator--',
    'options' => array('1', '2')
));?>

Output:

<div class="input">
--before--
<input name="data[User][field]" type="radio" value="1" id="UserField1" />
<label for="UserField1">1</label>
--separator--
<input name="data[User][field]" type="radio" value="2" id="UserField2" />
<label for="UserField2">2</label>
--between---
--after--
</div>
```

For date and datetime type elements the 'separator' attribute can be used to change the string between select elements. Defaults to '-'.

7.3.3.4 $options['options']

This key allows you to manually specify options for a select input, or for a radio group. Unless the 'type' is specified as 'radio', the FormHelper will assume that the target output is a select input.

```
<?php echo $this->Form->input('field', array('options' => array(1,2,3,4,5))); ?>
```

Output:

```
<div class="input">
   <label for="UserField">Field</label>
   <select name="data[User][field]" id="UserField">
      <option value="0">1</option>
      <option value="1">2</option>
      <option value="2">3</option>
      <option value="3">4</option>
      <option value="4">5</option>
   </select>
</div>
```

Options can also be supplied as key-value pairs.

```
        <?php echo $this->Form->input('field', array('options' => array(
            'Value 1'=>'Label 1',
            'Value 2'=>'Label 2',
            'Value 3'=>'Label 3'
         ))); ?>
```

Output:

```
<div class="input">
   <label for="UserField">Field</label>
   <select name="data[User][field]" id="UserField">
      <option value="Value 1">Label 1</option>
      <option value="Value 2">Label 2</option>
      <option value="Value 3">Label 3</option>
   </select>
</div>
```

If you would like to generate a select with optgroups, just pass data in hierarchical format. Works on multiple checkboxes and radio buttons too, but instead of optgroups wraps elements in fieldsets.

```
<?php echo $this->Form->input('field', array('options' => array(
    'Label1' => array(
       'Value 1'=>'Label 1',
       'Value 2'=>'Label 2'
    ),
    'Label2' => array(
       'Value 3'=>'Label 3'
    )
 ))); ?>
```

Output:

```
<div class="input">
  <label for="UserField">Field</label>
  <select name="data[User][field]" id="UserField">
    <optgroup label="Label1">
      <option value="Value 1">Label 1</option>
      <option value="Value 2">Label 2</option>
    </optgroup>
    <optgroup label="Label2">
      <option value="Value 3">Label 3</option>
    </optgroup>
  </select>
</div>
```

7.3.3.5 $options['multiple']

If 'multiple' has been set to true for an input that outputs a select, the select will allow multiple selections.

```
echo $this->Form->input('Model.field', array( 'type' => 'select', 'multiple' => true ));
```

Alternatively set 'multiple' to 'checkbox' to output a list of related check boxes.

```
        echo $this->Form->input('Model.field', array(
            'type' => 'select',
            'multiple' => 'checkbox',
            'options' => array(
                    'Value 1' => 'Label 1',
                    'Value 2' => 'Label 2'
            )
        ));
```

Output:

```
<div class="input select">
  <label for="ModelField">Field</label>
  <input name="data[Model][field]" value="" id="ModelField" type="hidden">
  <div class="checkbox">
    <input name="data[Model][field][]" value="Value 1" id="ModelField1" type="checkbox">
    <label for="ModelField1">Label 1</label>
  </div>
  <div class="checkbox">
    <input name="data[Model][field][]" value="Value 2" id="ModelField2" type="checkbox">
    <label for="ModelField2">Label 2</label>
  </div>
</div>
```

7.3.3.6 $options['maxLength']

Defines the maximum number of characters allowed in a text input.

7.3.3.7 $options['div']

Use this option to set attributes of the input's containing div. Using a string value will set the div's class name. An array will set the div's attributes to those specified by the array's keys/values. Alternatively, you can set this key to false to disable the output of the div.

Setting the class name:

```
echo $this->Form->input('User.name', array('div' => 'class_name'));
```

Output:

```
<div class="class_name">
        <label for="UserName">Name</label>
        <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Setting multiple attributes:

```
echo $this->Form->input('User.name', array('div' => array('id' => 'mainDiv', 'title' => 'Div Title', 'style' => 'display:block')));
```

Output:

```
<div class="input text" id="mainDiv" title="Div Title" style="display:block">
        <label for="UserName">Name</label>
        <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Disabling div output:

```
<?php echo $this->Form->input('User.name', array('div' => false));?>
```

Output:

```
<label for="UserName">Name</label>
<input name="data[User][name]" type="text" value="" id="UserName" />
```

7.3.3.8 $options['label']

Set this key to the string you would like to be displayed within the label that usually accompanies the input.

```
<?php echo $this->Form->input( 'User.name', array( 'label' => 'The User Alias' ) );?>
```

Output:

```
<div class="input">
   <label for="UserName">The User Alias</label>
   <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Alternatively, set this key to false to disable the output of the label.

```
<?php echo $this->Form->input( 'User.name', array( 'label' => false ) ); ?>
```

Output:

```
<div class="input">
   <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Set this to an array to provide additional options for the label element. If you do this, you can use a text key in the array to customize the label text.

```
<?php echo $this->Form->input( 'User.name', array( 'label' => array('class' => 'thingy', 'text' => 'The User Alias') ) ); ?>
```

Output:

```
<div class="input">
   <label for="UserName" class="thingy">The User Alias</label>
   <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

7.3.3.9 $options['legend']

Some inputs like radio buttons will be automatically wrapped in a fieldset with a legend title derived from the fields name. The title can be overridden with this option. Setting this option to false will completely eliminate the fieldset.

7.3.3.10 $options['id']

Set this key to force the value of the DOM id for the input.

7.3.3.11 $options['error']

Using this key allows you to override the default model error messages and can be used, for example, to set i18n messages. It has a number of suboptions which control the wrapping element, wrapping element class name, and whether HTML in the error message will be escaped.

To disable error message output set the error key to false.

```
•       $this->Form->input('Model.field', array('error' => false));
```

To modify the wrapping element type and its class, use the following format:

```
12.   $this->Form->input('Model.field', array('error' => array('wrap' => 'span', 'class' => 'bzzz')));
```

To prevent HTML being automatically escaped in the error message output, set the escape suboption to false:

```
•       $this->Form->input('Model.field', array('error' => array('escape' => false)));
```

To override the model error messages use an associate array with the keyname of the validation rule:

```
•       $this->Form->input('Model.field', array('error' => array('tooShort' => __('This is not long enough', true)
)));
```

As seen above you can set the error message for each validation rule you have in your models. In addition you can provide i18n messages for your forms.

7.3.3.12 $options['default']

Used to set a default value for the input field. The value is used if the data passed to the form does not contain a value for the field (or if no data is passed at all).

Example usage:

```
<?php
    echo $this->Form->input('ingredient', array('default'=>'Sugar'));
?>
```

Example with select field (Size "Medium" will be selected as default):

```
<?php
    $sizes = array('s'=>'Small', 'm'=>'Medium', 'l'=>'Large');
    echo $this->Form->input('size', array('options'=>$sizes, 'default'=>'m'));
?>
```

You cannot use default to check a checkbox - instead you might set the value in $this->data in your controller, $this->Form->data in your view, or set the input option checked to true.

Date and datetime fields' default values can be set by using the 'selected' key.

7.3.3.13 $options['selected']

Used in combination with a select-type input (i.e. For types select, date, time, datetime). Set 'selected' to the value of the item you wish to be selected by default when the input is rendered.

```
echo $this->Form->input('close_time', array('type' => 'time', 'selected' => '13:30:00'));
```

The selected key for date and datetime inputs may also be a UNIX timestamp.

7.3.3.14 $options['rows'], $options['cols']

These two keys specify the number of rows and columns in a textarea input.

```
echo $this->Form->input('textarea', array('rows' => '5', 'cols' => '5'));
```

Output:

```
<div class="input text">
    <label for="FormTextarea">Textarea</label>
    <textarea name="data[Form][textarea]" cols="5" rows="5" id="FormTextarea" >
    </textarea>
</div>
```

7.3.3.15 $options['empty']

If set to true, forces the input to remain empty.

When passed to a select list, this creates a blank option with an empty value in your drop down list. If you want to have a empty value with text displayed instead of just a blank option, pass in a string to empty.

```
<?php echo $this->Form->input('field', array('options' => array(1,2,3,4,5), 'empty' => '(choose one)')); ?>
```

Output:

```
<div class="input">
   <label for="UserField">Field</label>
   <select name="data[User][field]" id="UserField">
      <option value="">(choose one)</option>
      <option value="0">1</option>
      <option value="1">2</option>
      <option value="2">3</option>
      <option value="3">4</option>
      <option value="4">5</option>
   </select>
</div>
```

If you need to set the default value in a password field to blank, use 'value' => '' instead.

Options can also supplied as key-value pairs.

7.3.3.16 $options['timeFormat']

Used to specify the format of the select inputs for a time-related set of inputs. Valid values include '12', '24', and 'none'.

7.3.3.17 $options['dateFormat']

Used to specify the format of the select inputs for a date-related set of inputs. Valid values include 'DMY', 'MDY', 'YMD', and 'NONE'.

7.3.3.18 $options['minYear'], $options['maxYear']

Used in combination with a date/datetime input. Defines the lower and/or upper end of values shown in the years select field.

7.3.3.19 $options['interval']

This option specifies the number of minutes between each option in the minutes select box.

```
•        <?php echo $this->Form->input('Model.time', array('type' => 'time', 'interval' => 15)); ?>
```

Would create 4 options in the minute select. One for each 15 minutes.

7.3.3.20 $options['class']

You can set the classname for an input field using $options['class']

```
echo $this->Form->input('title', array('class' => 'custom-class'));
```

7.3.3.21 $options['hiddenField']

For certain input types (checkboxes, radios) a hidden input is created so that the key in $this->data will exist even without a value specified.

```
<input type="hidden" name="data[Post][Published]" id="PostPublished_" value="0" />
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

This can be disabled by setting the $options['hiddenField'] = false.

```
echo $this->Form->checkbox('published', array('hiddenField' => false));
```

Which outputs:

```
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

If you want to create multiple blocks of inputs on a form that are all grouped together, you should use this parameter on all inputs except the first. If the hidden input is on the page in multiple places, only the last group of input's values will be saved

In this example, only the tertiary colors would be passed, and the primary colors would be overridden

```
<h2>Primary Colors</h2>
<input type="hidden" name="data[Color][Color]" id="Colors_" value="0" />
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsRed" />
```

```
7.      <label for="ColorsRed">Red</label>
8.      <input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsBlue" />
9.      <label for="ColorsBlue">Blue</label>
10.     <input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsYellow" />
11.     <label for="ColorsYellow">Yellow</label>
12.     <h2>Tertiary Colors</h2>
13.     <input type="hidden" name="data[Color][Color]" id="Colors_" value="0" />
14.     <input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsGreen" />
15.     <label for="ColorsGreen">Green</label>
16.     <input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsPurple" />
17.     <label for="ColorsPurple">Purple</label>
18.     <input type="checkbox" name="data[Addon][Addon][]" value="5" id="ColorsOrange" />
19.     <label for="ColorsOrange">Orange</label>
```

Disabling the 'hiddenField' on the second input group would prevent this behavior

7.3.4 File Fields

To add a file upload field to a form, you must first make sure that the form enctype is set to "multipart/form-data", so start off with a create function such as the following.

```
1.  echo $this->Form->create('Document', array('enctype' => 'multipart/form-data') );
2.  // OR
3.  echo $this->Form->create('Document', array('type' => 'file'));
```

Next add either of the two lines to your form view file.

```
•   echo $this->Form->input('Document.submittedfile', array('between'=>'<br />','type'=>'file'));
•   // or
•   echo $this->Form->file('Document.submittedfile');
```

Due to the limitations of HTML itself, it is not possible to put default values into input fields of type 'file'. Each time the form is displayed, the value inside will be empty.

Upon submission, file fields provide an expanded data array to the script receiving the form data.

For the example above, the values in the submitted data array would be organized as follows, if the CakePHP was installed on a Windows server. 'tmp_name' will have a different path in a Unix environment.

```
$this->data['Document']['submittedfile'] = array(
    'name' => conference_schedule.pdf
    'type' => application/pdf
    'tmp_name' => C:/WINDOWS/TEMP/php1EE.tmp
    'error' => 0
    'size' => 41737
);
```

This array is generated by PHP itself, so for more detail on the way PHP handles data passed via file fields read the PHP manual section on file uploads.

7.3.4.1 Validating Uploads

Below is an example validation method you could define in your model to validate whether a file has been successfully uploaded.

```
// Based on comment 8 from: http://bakery.cakephp.org/articles/view/improved-advance-validation-with-parameters
function isUploadedFile($params){
    $val = array_shift($params);
    if ((isset($val['error']) && $val['error'] == 0) ||
    (!empty( $val['tmp_name']) && $val['tmp_name'] != 'none')) {
        return is_uploaded_file($val['tmp_name']);
    }
    return false;
```

```
}
```

7.3.5 Form Element-Specific Methods

The rest of the methods available in the FormHelper are for creating specific form elements. Many of these methods also make use of a special $options parameter. In this case, however, $options is used primarily to specify HTML tag attributes (such as the value or DOM id of an element in the form).

```
<?php echo $this->Form->text('username', array('class' => 'users')); ?>
```

Will output:

```
<input name="data[User][username]" type="text" class="users" id="UserUsername" />
```

7.3.5.1 checkbox

checkbox(string $fieldName, array $options)

Creates a checkbox form element. This method also generates an associated hidden form input to force the submission of data for the specified field.

```
<?php echo $this->Form->checkbox('done'); ?>
```

Will output:

```
<input type="hidden" name="data[User][done]" value="0" id="UserDone_" />
<input type="checkbox" name="data[User][done]" value="1" id="UserDone" />
```

It is possible to specify the value of the checkbox by using the $options array:

```
<?php echo $this->Form->checkbox('done', array('value' => 555)); ?>
```

Will output:

```
<input type="hidden" name="data[User][done]" value="0" id="UserDone_" />
<input type="checkbox" name="data[User][done]" value="555" id="UserDone" />
```

If you don't want the Form helper to create a hidden input:

```
<?php echo $this->Form->checkbox('done', array('hiddenField' => false)); ?>
```

Will output:

```
<input type="checkbox" name="data[User][done]" value="1" id="UserDone" />
```

7.3.5.2 button

button(string $title, array $options = array())

Creates an HTML button with the specified title and a default type of "button". Setting `$options['type']` will output one of the three possible button types:

- submit: Same as the `$this->Form->submit` method - (the default).
- reset: Creates a form reset button.
- button: Creates a standard push button.

```
<?php
echo $this->Form->button('A Button');
echo $this->Form->button('Another Button', array('type'=>'button'));
echo $this->Form->button('Reset the Form', array('type'=>'reset'));
echo $this->Form->button('Submit Form', array('type'=>'submit'));
?>
```

Will output:

```
<button type="submit">A Button</button>
<button type="button">Another Button</button>
<button type="reset">Reset the Form</button>
<button type="submit">Submit Form</button>
```

The `button` input type allows for a special `$option` attribute called `'escape'` which accepts a bool and determines whether to HTML entity encode the $title of the button. Defaults to false.

```
<?php
    echo $this->Form->button('Submit Form', array('type'=>'submit','escape'=>true));
?>
```

7.3.5.3 year

`year(string $fieldName, int $minYear, int $maxYear, mixed $selected, array $attributes)`

Creates a select element populated with the years from `$minYear` to `$maxYear`, with the $selected year selected by default. HTML attributes may be supplied in $attributes. If `$attributes['empty']` is false, the select will not include an empty option.

```
<?php
echo $this->Form->year('purchased',2000,date('Y'));
?>
```

Will output:

```
<select name="data[User][purchased][year]" id="UserPurchasedYear">
<option value=""></option>
<option value="2009">2009</option>
<option value="2008">2008</option>
<option value="2007">2007</option>
<option value="2006">2006</option>
<option value="2005">2005</option>
<option value="2004">2004</option>
```

```
<option value="2003">2003</option>

<option value="2002">2002</option>
<option value="2001">2001</option>
<option value="2000">2000</option>
</select>
```

7.3.5.4 month

month(string $fieldName, mixed $selected, array $attributes)

Creates a select element populated with month names.

```
<?php
echo $this->Form->month('mob');
?>
```

Will output:

```
<select name="data[User][mob][month]" id="UserMobMonth">
<option value=""></option>
<option value="01">January</option>
<option value="02">February</option>
<option value="03">March</option>
<option value="04">April</option>
<option value="05">May</option>
<option value="06">June</option>
<option value="07">July</option>
<option value="08">August</option>
<option value="09">September</option>
<option value="10">October</option>
<option value="11">November</option>
<option value="12">December</option>
</select>
```

You can pass in your own array of months to be used by setting the 'monthNames' attribute, or have months displayed as numbers by passing false. (Note: the default months are internationalized and can be translated using localization.)

```
1. <?php
2. echo $this->Form->month('mob', null, array('monthNames' => false));
3. ?>
```

7.3.5.5 dateTime

dateTime($fieldName, $dateFormat = 'DMY', $timeFormat = '12', $selected = null, $attributes = array())

Creates a set of select inputs for date and time. Valid values for $dateformat are ‘DMY’, ‘MDY’, ‘YMD’ or ‘NONE’. Valid values for $timeFormat are ‘12’, ‘24’, and null.

You can specify not to display empty values by setting "array('empty' => false)" in the attributes parameter. You also can pre-select the current datetime by setting $selected = null and $attributes = array("empty" => false).

7.3.5.6 day

day(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)

Creates a select element populated with the (numerical) days of the month.

To create an empty option with prompt text of your choosing (e.g. the first option is 'Day'), you can supply the text as the final parameter as follows:

```
• <?php
• echo $this->Form->day('created');
• ?>
```

Will output:

```
<select name="data[User][created][day]" id="UserCreatedDay">
```

```
<option value=""></option>
<option value="01">1</option>
<option value="02">2</option>
<option value="03">3</option>
...
<option value="31">31</option>
</select>
```

7.3.5.7 hour

```
hour(string $fieldName, boolean $format24Hours, mixed $selected, array $attributes, boolean $showEmpty)
```

Creates a select element populated with the hours of the day.

7.3.5.8 minute

```
minute(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)
```

Creates a select element populated with the minutes of the hour.

7.3.5.9 meridian

```
meridian(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)
```

Creates a select element populated with ‘am’ and ‘pm’.

7.3.5.10 error

```
error(string $fieldName, mixed $text, array $options)
```

Shows a validation error message, specified by $text, for the given field, in the event that a validation error has occurred.

Options:

1. 'escape' bool Whether or not to html escape the contents of the error.
2. 'wrap' mixed Whether or not the error message should be wrapped in a div. If a string, will be used as the HTML tag to use.

3. 'class' string The classname for the error message

7.3.5.11 file

file(string $fieldName, array $options)

Creates a file input.

```
1. <?php
2. echo $this->Form->create('User',array('type'=>'file'));
3. echo $this->Form->file('avatar');
4. ?>
```

Will output:

```
<form enctype="multipart/form-data" method="post" action="/users/add">
<input name="data[User][avatar]" value="" id="UserAvatar" type="file">
```

When using `$this->Form->file()`, remember to set the form encoding-type, by setting the type option to 'file' in `$this->Form->create()`

7.3.5.12 hidden

hidden(string $fieldName, array $options)

Creates a hidden form input. Example:

```
1. <?php
2. echo $this->Form->hidden('id');
3. ?>
```

Will output:

```
<input name="data[User][id]" value="10" id="UserId" type="hidden">
```

7.3.5.13 isFieldError

isFieldError(string $fieldName)

Returns true if the supplied $fieldName has an active validation error.

```
1. <?php
2. if ($this->Form->isFieldError('gender')){
3.     echo $this->Form->error('gender');
4. }
5. ?>
```

When using `$this->Form->input()`, errors are rendered by default.

7.3.5.14 label

label(string $fieldName, string $text, array $attributes)

Creates a label tag, populated with $text.

```
1. <?php
2. echo $this->Form->label('status');
3. ?>
```

Will output:

```
<label for="UserStatus">Status</label>
```

7.3.5.15 password

password(string $fieldName, array $options)

Creates a password field.

```
1. <?php
2. echo $this->Form->password('password');
3. ?>
```

Will output:

```
<input name="data[User][password]" value="" id="UserPassword" type="password">
```

7.3.5.16 radio

radio(string $fieldName, array $options, array $attributes)

Creates a radio button input. Use `$attributes['value']` to set which value should be selected default.

Use `$attributes['separator']` to specify HTML in between radio buttons (e.g.
).

Radio elements are wrapped with a label and fieldset by default. Set `$attributes['legend']` to false to remove them.

```
• <?php
• $options=array('M'=>'Male','F'=>'Female');
• $attributes=array('legend'=>false);
• echo $this->Form->radio('gender',$options,$attributes);
• ?>
```

Will output:

```
<input name="data[User][gender]" id="UserGender_" value="" type="hidden">
<input name="data[User][gender]" id="UserGenderM" value="M" type="radio">
<label for="UserGenderM">Male</label>
<input name="data[User][gender]" id="UserGenderF" value="F" type="radio">
<label for="UserGenderF">Female</label>
```

If for some reason you don't want the hidden input, setting `$attributes['value']` to a selected value or boolean false will do just that.

7.3.5.17 select

select(string $fieldName, array $options, mixed $selected, array $attributes)

Creates a select element, populated with the items in $options, with the option specified by $selected shown as selected by default. If you wish to display your own default option, add your string value to the 'empty' key in the $attributes variable, or set it to false to turn off the default empty option

```
<?php
$options = array('M' => 'Male', 'F' => 'Female');
echo $this->Form->select('gender', $options)
?>
```

Will output:

```
<select name="data[User][gender]" id="UserGender">
<option value=""></option>
<option value="M">Male</option>
<option value="F">Female</option>
</select>
```

The select input type allows for a special $option attribute called 'escape' which accepts a bool and determines whether to HTML entity encode the contents of the select options. Defaults to true.

```
<?php
$options = array('M' => 'Male', 'F' => 'Female');
echo $this->Form->select('gender', $options, null, array('escape' => false));
?>
```

7.3.5.18 submit

submit(string $caption, array $options)

Creates a submit button with caption $caption. If the supplied $caption is a URL to an image (it contains a '.' character), the submit button will be rendered as an image.

It is enclosed between div tags by default; you can avoid this by declaring $options['div'] = false.

```
<?php
echo $this->Form->submit();
?>
```

Will output:

```
<div class="submit"><input value="Submit" type="submit"></div>
```

You can also pass a relative or absolute url to an image for the caption parameter instead of caption text.

```
<?php
echo $this->Form->submit('ok.png');
?>
```

Will output:

```
<div class="submit"><input type="image" src="/img/ok.png"></div>
```

7.3.5.19 text

text(string $fieldName, array $options)

Creates a text input field.

```
<?php
echo $this->Form->text('first_name');
?>
```

Will output:

```
<input name="data[User][first_name]" value="" id="UserFirstName" type="text">
```

7.3.5.20 textarea

textarea(string $fieldName, array $options)

Creates a textarea input field.

```
<?php
echo $this->Form->textarea('notes');
?>
```

Will output:

```
<textarea name="data[User][notes]" id="UserNotes"></textarea>
```

The textarea input type allows for the $options attribute of 'escape' which determines whether or not the contents of the textarea should be escaped. Defaults to true.

```
<?php
echo $this->Form->textarea('notes', array('escape' => false));
// OR....
```

```
        echo $this->Form->input('notes', array('type' => 'textarea', 'escape' => false);
        ?>
```

7.3.6 1.3 improvements

The FormHelper is one of the most frequently used classes in CakePHP, and has had several improvements made to it.

Entity depth limitations

In 1.2 there was a hard limit of 5 nested keys. This posed significant limitations on form input creation in some contexts. In 1.3 you can now create infinitely nested form element keys. Validation errors and value reading for arbitrary depths has also been added.

Model introspection

Support for adding 'required' classes, and properties like `maxlength` to hasMany and other associations has been improved. In the past only 1 model and a limited set of associations would be introspected. In 1.3 models are introspected as needed, providing validation and additional information such as maxlength.

Default options for input()

In the past if you needed to use `'div' => false`, or `'label' => false` you would need to set those options on each and every call to `input()`. Instead in 1.3 you can declare a set of default options for `input()` with the `inputDefaults` key.

```
1.      echo $this->Form->create('User', array(
2.              'inputDefaults' => array(
3.                  'label' => false,
4.                  'div' => false
5.              )
6.          ));
```

All inputs created from that point forward would inherit the options declared in inputDefaults. You can override the defaultOptions by declaring the option in the input() call.

```
1.  echo $this->Form->input('password'); // No div, no label
2.  echo $this->Form->input('username', array('label' => 'Username')); // has a label element
```

Omit attributes

You can now set any attribute key to null or false in an options/attributes array to omit that attribute from a particular html tag.

```
1.  echo $this->Form->input('username', array(
2.      'div' => array('class' => false)
3.  )); // Omits the 'class' attribute added by default to div tag
```

Accept-charset

Forms now get an accept-charset set automatically, it will match the value of `App.encoding`, it can be overridden or removed using the 'encoding' option when calling create().

```
1.  // To remove the accept-charset attribute.
2.  echo $this->Form->create('User', array('encoding' => null));
```

Removed parameters

Many methods such as `select`, `year`, `month`, `day`, `hour`, `minute`, `meridian` and `datetime` took a `$showEmpty` parameter, these have all been removed and rolled into the `$attributes` parameter using the `'empty'` key.

Default url

The default url for forms either was `add` or `edit` depending on whether or not a primary key was detected in the data array. In 1.3 the default url will be the current action, making the forms submit to the action you are currently on.

Disabling hidden inputs for radio and checkbox

The automatically generated hidden inputs for radio and checkbox inputs can be disabled by setting the `'hiddenField'` option to `false`.

button()

button() now creates button elements, these elements by default do not have html entity encoding enabled. You can enable html escaping using the `escape` option. The former features of `FormHelper::button` have been moved to `FormHelper::submit`.

submit()

Due to changes in `button()`, `submit()` can now generate reset, and other types of input buttons. Use the `type` option to change the default type of button generated. In addition to creating all types of buttons, `submit()` has `before` and `after` options that behave exactly like their counterparts in `input()`.

$options['format']

The HTML generated by the form helper is now more flexible than ever before. The $options parameter to Form::input() now supports an array of strings describing the template you would like said element to follow. It's just been recently added to SCM, and has a few bugs for non PHP 5.3 users, but should be quite useful for all. The supported array keys are `array('before', 'input', 'between', 'label', 'after', 'error')`.

7.4 HTML

- Edit
- Comments (0)
- History

The role of the HtmlHelper in CakePHP is to make HTML-related options easier, faster, and more resilient to change. Using this helper will enable your application to be more light on its feet, and more flexible on where it is placed in relation to the root of a domain.

Before we look at HtmlHelper's methods, you'll need to know about a few configuration and usage situations that will help you use this class. First in an effort to assuage those who dislike short tags (<?= ?>) or many echo() calls in their view code all methods of HtmlHelper are passed to the output() method. If you wish to enable automatic output of the generated helper HTML you can simply implement output() in your AppHelper.

```
1.      function output($str) {
2.          echo $str;
3.      }
```

Doing this will remove the need to add echo statements to your view code.

Many HtmlHelper methods also include a $htmlAttributes parameter, that allow you to tack on any extra attributes on your tags. Here are a few examples of how to use the $htmlAttributes parameter:

```
1.      Desired attributes: <tag class="someClass" />
2.      Array parameter: array('class'=>'someClass')
3.
4.      Desired attributes: <tag name="foo" value="bar" />
5.      Array parameter:  array('name' => 'foo', 'value' => 'bar')
```

The HtmlHelper is available in all views by default. If you're getting an error informing you that it isn't there, it's usually due to its name being missing from a manually configured $helpers controller variable.

7.4.1 Inserting Well-Formatted elements

The most important task the HtmlHelper accomplishes is creating well formed markup. Don't be afraid to use it often - you can cache views in CakePHP in order to save some CPU cycles when views are being rendered and delivered. This section will cover some of the methods of the HtmlHelper and how to use them.

7.4.1.1 charset

```
charset(string $charset=null)
```

Used to create a meta tag specifying the document's character. Defaults to UTF-8.

```
1.      <?php echo $this->Html->charset(); ?>
```

Will output:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Alternatively,

```
1.      <?php echo $this->Html->charset('ISO-8859-1'); ?>
```

Will output:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

7.4.1.2 css

css(mixed $path, string $rel = null, array $options = array())

Creates a link(s) to a CSS style-sheet. If key 'inline' is set to false in $options parameter, the link tags are added to the $scripts_for_layout variable which you can print inside the head tag of the document.

This method of CSS inclusion assumes that the CSS file specified resides inside the /app/webroot/css directory.

```
1.      <?php echo $this->Html->css('forms'); ?>
```

Will output:

```
<link rel="stylesheet" type="text/css" href="/css/forms.css" />
```

The first parameter can be an array to include multiple files.

```
1.        <?php echo $this->Html->css(array('forms','tables','menu')); ?>
```

Will output:

```
<link rel="stylesheet" type="text/css" href="/css/forms.css" />
<link rel="stylesheet" type="text/css" href="/css/tables.css" />
<link rel="stylesheet" type="text/css" href="/css/menu.css" />
```

7.4.1.3 meta

```
meta(string $type, string $url = null, array $attributes = array())
```

This method is handy for linking to external resources like RSS/Atom feeds and favicons. Like css(), you can specify whether or not you'd like this tag to appear inline or in the head tag by setting the 'inline' key in the $attributes parameter to false, ie - `array('inline' => false)`.

If you set the "type" attribute using the $attributes parameter, CakePHP contains a few shortcuts:

type	translated value
html	text/html
rss	application/rss+xml
atom	application/atom+xml
icon	image/x-icon

```
1.        <?php echo $this->Html->meta(
2.            'favicon.ico',
3.            '/favicon.ico',
4.            array('type' => 'icon')
5.        );?> //Output (line breaks added) </p>
6.        <link
```

```
            href="http://example.com/favicon.ico"
            title="favicon.ico" type="image/x-icon"
            rel="alternate"
        />

        <?php echo $this->Html->meta(
            'Comments',
            '/comments/index.rss',
            array('type' => 'rss'));
        ?>

        //Output (line breaks added)
        <link
            href="http://example.com/comments/index.rss"
            title="Comments"
            type="application/rss+xml"
            rel="alternate"
        />
```

This method can also be used to add the meta keywords and descriptions. Example:

```
        <?php echo $this->Html->meta(
            'keywords',
            'enter any meta keyword here'
        );?>
        //Output <meta name="keywords" content="enter any meta keyword here"/>
        //
        <?php echo $this->Html->meta(
            'description',
            'enter any meta description here'
```

```
10.         );?>
11.       //Output <meta name="description" content="enter any meta description here"/>
```

If you want to add a custom meta tag then the first parameter should be set to an array. To output a robots noindex tag use the following code:

```
1.         echo $this->Html->meta(array('name' => 'robots', 'content' => 'noindex'));
```

7.4.1.4 docType

`docType(string $type = 'xhtml-strict')`

Returns a (X)HTML doctype tag. Supply the doctype according to the following table:

type	translated value
html	text/html
html4-strict	HTML4 Strict
html4-trans	HTML4 Transitional
html4-frame	HTML4 Frameset
xhtml-strict	XHTML1 Strict
xhtml-trans	XHTML1 Transitional
xhtml-frame	XHTML1 Frameset
xhtml11	XHTML 1.1

```
1.       <?php echo $this->Html->docType(); ?>
```

```
2.      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3.      <?php echo $this->Html->docType('html4-trans'); ?>
4.      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
```

7.4.1.5 style

style(array $data, boolean $oneline = true)

Builds CSS style definitions based on the keys and values of the array passed to the method. Especially handy if your CSS file is dynamic.

```
1.      <?php echo $this->Html->style(array(
2.          'background'     => '#633',
3.          'border-bottom' => '1px solid #000',
4.          'padding' => '10px'
5.      )); ?>
```

Will output:

```
background:#633; border-bottom:1px solid #000; padding:10px;
```

7.4.1.6 image

image(string $path, array $htmlAttributes = array())

Creates a formatted image tag. The path supplied should be relative to `/app/webroot/img/`.

```
1.      <?php echo $this->Html->image('cake_logo.png', array('alt' => 'CakePHP'))?>
```

Will output:

```
<img src="/img/cake_logo.png" alt="CakePHP" />
```

To create an image link specify the link destination using the `url` option in `$htmlAttributes`.

```
1.      <?php echo $this->Html->image("recipes/6.jpg", array(
2.          "alt" => "Brownies",
3.          'url' => array('controller' => 'recipes', 'action' => 'view', 6)
4.      )); ?>
```

Will output:

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

7.4.1.7 link

`link(string $title, mixed $url = null, array $options = array(), string $confirmMessage = false)`

General purpose method for creating HTML links. Use `$options` to specify attributes for the element and whether or not the `$title` should be escaped.

```
1.      <?php echo $this->Html->link('Enter', '/pages/home', array('class' => 'button', 'target' => '_blank')); ?>
```

Will output:

```
<a href="/pages/home" class="button" target="_blank">Enter</a>
```

Specify `$confirmMessage` to display a javascript `confirm()` dialog.

```
1.      <?php echo $this->Html->link(
2.          'Delete',
3.          array('controller' => 'recipes', 'action' => 'delete', 6),
4.          array(),
5.          "Are you sure you wish to delete this recipe?"
```

```
6.      );?>
```

Will output:

```
<a href="/recipes/delete/6" onclick="return confirm('Are you sure you wish to delete this recipe?');">Delete</a>
```

Query strings can also be created with `link()`.

```
1.      <?php echo $this->Html->link('View image', array(
2.          'controller' => 'images',
3.          'action' => 'view',
4.          1,
5.          '?' => array('height' => 400, 'width' => 500))
6.      );
```

Will output:

```
<a href="/images/view/1?height=400&width=500">View image</a>
```

HTML special characters in `$title` will be converted to HTML entities. To disable this conversion, set the escape option to false in the `$options` array.

```
1.      <?php
2.      echo $this->Html->link(
3.          $this->Html->image("recipes/6.jpg", array("alt" => "Brownies")),
4.          "recipes/view/6",
5.          array('escape' => false)
6.      );
7.      ?>
```

Will output:

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Also check HtmlHelper::url method for more examples of different types of urls.

7.4.1.8 tag

tag(string $tag, string $text, array $htmlAttributes)

Returns text wrapped in a specified tag. If no text is specified then only the opening <tag> is returned.

```
1.      <?php echo $this->Html->tag('span', 'Hello World.', array('class' => 'welcome'));?>
2.
3.      //Output
4.      <span class="welcome">Hello World</span>
5.
6.      //No text specified.
7.      <?php echo $this->Html->tag('span', null, array('class' => 'welcome'));?>
8.
9.      //Output
10.     <span class="welcome">
```

Text is not escaped by default but you may use `$htmlOptions['escape'] = true` to escape your text. This replaces a fourth parameter `boolean $escape = false` that was available in previous versions.

7.4.1.9 div

div(string $class, string $text, array $options)

Used for creating div-wrapped sections of markup. The first parameter specifies a CSS class, and the second is used to supply the text to be wrapped by div tags. If the last parameter has been set to true, $text will be printed HTML-escaped.

If no text is specified, only an opening div tag is returned.

```
1.      <?php echo $this->Html->div('error', 'Please enter your credit card number.');?>
2.
3.      //Output
4.      <div class="error">Please enter your credit card number.</div>
```

7.4.1.10 para

para(string $class, string $text, array $htmlAttributes, boolean $escape = false)

Returns a text wrapped in a CSS-classed <p> tag. If no text is supplied, only a starting <p> tag is returned.

```
1.      <?php echo $this->Html->para(null, 'Hello World.');?>
2.
3.      //Output
4.      <p>Hello World.</p>
```

7.4.1.11 script

script(mixed $url, mixed $options)

Creates link(s) to a javascript file. If key `inline` is set to false in $options, the link tags are added to the $scripts_for_layout variable which you can print inside the head tag of the document.

Include a script file into the page. `$options['inline']` controls whether or not a script should be returned inline or added to $scripts_for_layout. `$options['once']` controls, whether or not you want to include this script once per request or more than once.

You can also use $options to set additional properties to the generated script tag. If an array of script tags is used, the attributes will be applied to all of the generated script tags.

This method of javascript file inclusion assumes that the javascript file specified resides inside the /app/webroot/js directory.

```
1.        <?php echo $this->Html->script('scripts'); ?>
```

Will output:

```
<script type="text/javascript" href="/js/scripts.js"></script>
```

You can link to files with absolute paths as well to link files that are not in `app/webroot/js`

```
1.        <?php echo $this->Html->script('/otherdir/script_file'); ?>
```

The first parameter can be an array to include multiple files.

```
1.        <?php echo $this->Html->script(array('jquery','wysiwyg','scripts')); ?>
```

Will output:

```
<script type="text/javascript" href="/js/jquery.js"></script>
<script type="text/javascript" href="/js/wysiwyg.js"></script>
<script type="text/javascript" href="/js/scripts.js"></script>
```

7.4.1.12 scriptBlock

scriptBlock($code, $options = array())

Generate a code block containing `$code` set `$options['inline']` to false to have the script block appear in `$scripts_for_layout`. Also new is the ability to add attributes to script tags. `$this->Html->scriptBlock('stuff', array('defer' => true));` will create a script tag with `defer="defer"` attribute.

7.4.1.13 scriptStart

scriptStart($options = array())

Begin a buffering code block. This code block will capture all output between `scriptStart()` and `scriptEnd()` and create an script tag. Options are the same as `scriptBlock()`

7.4.1.14 scriptEnd

scriptEnd()

End a buffering script block, returns the generated script element or null if the script block was opened with inline = false.

An example of using `scriptStart()` and `scriptEnd()` would be:

```
1.      $this->Html->scriptStart(array('inline' => false));
2.      echo $this->Js->alert('I am in the javascript');
3.      $this->Html->scriptEnd();
```

7.4.1.15 tableHeaders

tableHeaders(array $names, array $trOptions = null, array $thOptions = null)

Creates a row of table header cells to be placed inside of <table> tags.

```
1.      <?php echo $this->Html->tableHeaders(array('Date','Title','Active'));?>
2.
3.      //Output
4.      <tr>
5.          <th>Date</th>
6.          <th>Title</th>
7.          <th>Active</th>
8.      </tr>
9.
10.     <?php echo $this->Html->tableHeaders(
11.         array('Date','Title','Active'),
```

```
            array('class' => 'status'),
            array('class' => 'product_table')
        );?>

        //Output
        <tr class="status">
             <th class="product_table">Date</th>
             <th class="product_table">Title</th>
             <th class="product_table">Active</th>
        </tr>
```

7.4.1.16 tableCells

tableCells(array $data, array $oddTrOptions = null, array $evenTrOptions = null, $useCount = false, $continueOddEven = true)

Creates table cells, in rows, assigning <tr> attributes differently for odd- and even-numbered rows. Wrap a single table cell within an array() for specific <td>-attributes.

```
        <?php echo $this->Html->tableCells(array(
            array('Jul 7th, 2007', 'Best Brownies', 'Yes'),
            array('Jun 21st, 2007', 'Smart Cookies', 'Yes'),
            array('Aug 1st, 2006', 'Anti-Java Cake', 'No'),
        ));
        ?>

        //Output
        <tr><td>Jul 7th, 2007</td><td>Best Brownies</td><td>Yes</td></tr>
        <tr><td>Jun 21st, 2007</td><td>Smart Cookies</td><td>Yes</td></tr>
        <tr><td>Aug 1st, 2006</td><td>Anti-Java Cake</td><td>No</td></tr>
```

```
<?php echo $this->Html->tableCells(array(
    array('Jul 7th, 2007', array('Best Brownies', array('class'=>'highlight')) , 'Yes'),
    array('Jun 21st, 2007', 'Smart Cookies', 'Yes'),
    array('Aug 1st, 2006', 'Anti-Java Cake', array('No', array('id'=>'special'))),
));
?>

//Output
<tr><td>Jul 7th, 2007</td><td class="highlight">Best Brownies</td><td>Yes</td></tr>
<tr><td>Jun 21st, 2007</td><td>Smart Cookies</td><td>Yes</td></tr>
<tr><td>Aug 1st, 2006</td><td>Anti-Java Cake</td><td id="special">No</td></tr>

<?php echo $this->Html->tableCells(
    array(
        array('Red', 'Apple'),
        array('Orange', 'Orange'),
        array('Yellow', 'Banana'),
    ),
    array('class' => 'darker')
);
?>

//Output
<tr class="darker"><td>Red</td><td>Apple</td></tr>
<tr><td>Orange</td><td>Orange</td></tr>
<tr class="darker"><td>Yellow</td><td>Banana</td></tr>
```

7.4.1.17 url

url(mixed $url = NULL, boolean $full = false)

Returns an URL pointing to a combination of controller and action. If $url is empty, it returns the REQUEST_URI, otherwise it generates the url for the controller and action combo. If full is true, the full base URL will be prepended to the result.

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "bar"));?>

// Output
/posts/view/bar
```

Here are a few more usage examples:

URL with named parameters

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "foo" => "bar"));
?>

// Output
/posts/view/foo:bar
```

URL with extension

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "list",
```

```
            "ext" => "rss"));
        ?>

        // Output
        /posts/list.rss
```

URL (starting with '/') with the full base URL prepended.

```
        <?php echo $this->Html->url('/posts', true); ?>

        //Output
        http://somedomain.com/posts
```

URL with GET params and named anchor

```
        <?php echo $this->Html->url(array(
            "controller" => "posts",
            "action" => "search",
            "?" => array("foo" => "bar"),
            "#" => "first"));
        ?>

        //Output
        /posts/search?foo=bar#first
```

For further information check Router::url in the API.

7.4.2 Changing the tags output by HtmlHelper

The built in tag sets for `HtmlHelper` are XHTML compliant, however if you need to generate HTML for HTML4 you will need to create and load a new tags config file containing the tags you'd like to use. To change the tags used create `app/config/tags.php` containing:

```
1.      $tags = array(
2.          'metalink' => '<link href="%s"%s >',
3.          'input' => '<input name="%s" %s >',
4.          //...
5.      );
```

You can then load this tag set by calling `$html->loadConfig('tags');`

7.4.3 Creating breadcrumb trails with HtmlHelper

CakePHP has the built in ability to automatically create a breadcrumb trail in your app. To set this up, first add something similar to the following in your layout template.

```
1.      echo $this->Html->getCrumbs(' > ','Home');
```

Now, in your view you'll want to add the following to start the breadcrumb trails on each of the pages.

```
•       echo $this->Html->addCrumb('Users', '/users');
•       echo $this->Html->addCrumb('Add User', '/users/add');
```

This will add the output of "**Home > Users > Add User**" in your layout where getCrumbs was added.

7.5 Js

Since the beginning CakePHP's support for Javascript has been with Prototype/Scriptaculous. While we still think these are an excellent Javascript library, the community has been asking for support for other libraries. Rather than drop Prototype in favour of another Javascript library. We created an Adapter based helper, and included 3 of the most requested libraries. Prototype/Scriptaculous, Mootools/Mootools-more, and jQuery/jQuery UI. And while the API

is not as expansive as the previous AjaxHelper we feel that the adapter based solution allows for a more extensible solution giving developers the power and flexibility they need to address their specific application needs.

Javascript Engines form the backbone of the new JsHelper. A Javascript engine translates an abstract Javascript element into concrete Javascript code specific to the Javascript library being used. In addition they create an extensible system for others to use.

7.5.1 Using a specific Javascript engine

First of all download your preferred javascript library and place it in `app/webroot/js`

Then you must include the library in your page. To include it in all pages, add this line to the <head> section of `app/views/layouts/default.ctp` (copy this file from `cake/libs/view/layouts/default.ctp` if you have not created your own).

```
1.      echo $this->Html->script('jquery'); // Include jQuery library
```

Replace `jquery` with the name of your library file (.js will be added to the name).

By default scripts are cached, and you must explicitly print out the cache. To do this at the end of each page, include this line just before the ending `</body>` tag

```
1.      echo $this->Js->writeBuffer(); // Write cached scripts
```

You must include the library in your page and print the cache for the helper to function.

Javascript engine selection is declared when you include the helper in your controller.

```
1.      var $helpers = array('Js' => array('Jquery'));
```

The above would use the Jquery Engine in the instances of JsHelper in your views. If you do not declare a specific engine, the jQuery engine will be used as the default. As mentioned before, there are three engines implemented in the core, but we encourage the community to expand the library compatibility.

 # Using jQuery with other libraries

The jQuery library, and virtually all of its plugins are constrained within the jQuery namespace. As a general rule, "global" objects are stored inside the jQuery namespace as well, so you shouldn't get a clash between jQuery and any other library (like Prototype, MooTools, or YUI).

That said, there is one caveat: **By default, jQuery uses "$" as a shortcut for "jQuery"**

To override the "$" shortcut, use the jQueryObject variable.

```
1.      $this->Js->JqueryEngine->jQueryObject = '$j';
2.      print $this->Html->scriptBlock('var $j = jQuery.noConflict();',
3.          array('inline' => false)); //Tell jQuery to go into noconflict mode
```

7.5.1.1 Using the JsHelper inside customHelpers

Declare the JsHelper in the `$helpers` array in your customHelper.

```
1.      var $helpers = array('Js');
```

It is not possible to declare a javascript engine inside a custom helper. Doing that will have no effect.

If you are willing to use an other javascript engine than the default, do the helper setup in your controller as follows:

```
1.      var $helpers = array(
2.          'Js' => array('Prototype'),
3.          'CustomHelper'
4.      );
```

Be sure to declare the JsHelper and its engine **on top** of the `$helpers` array in your controller.

The selected javascript engine may disappear (replaced by the default) from the jsHelper object in your helper, if you miss to do so and you will get code that does not fit your javascript library.

7.5.2 Creating a Javascript Engine

Javascript engine helpers follow normal helper conventions, with a few additional restrictions. They must have the `Engine` suffix. `DojoHelper` is not good, `DojoEngineHelper` is correct. Furthermore, they should extend `JsBaseEngineHelper` in order to leverage the most of the new API.

7.5.3 Javascript engine usage

The `JsHelper` provides a few methods, and acts as a facade for the the Engine helper. You should not directly access the Engine helper except in rare occasions. Using the facade features of the `JsHelper` allows you to leverage the buffering and method chaining features built-in; (method chaining only works in PHP5).

The `JsHelper` by default buffers almost all script code generated, allowing you to collect scripts throughout the view, elements and layout, and output it in one place. Outputting buffered scripts is done with `$this->Js->writeBuffer();` this will return the buffer contents in a script tag. You can disable buffering wholesale with the `$bufferScripts` property or setting `buffer => false` in methods taking `$options`.

Since most methods in Javascript begin with a selection of elements in the DOM, `$this->Js->get()` returns a $this, allowing you to chain the methods using the selection. Method chaining allows you to write shorter, more expressive code. It should be noted that method chaining **Will not** work in PHP4.

```
1.      $this->Js->get('#foo')->event('click', $eventCode);
```

Is an example of method chaining. Method chaining is not possible in PHP4 and the above sample would be written like:

```
1.      $this->Js->get('#foo');
2.      $this->Js->event('click', $eventCode);
```

Common options

In attempts to simplify development where Js libraries can change, a common set of options is supported by `JsHelper`, these common options will be mapped out to the library specific options internally. If you are not planning on switching Javascript libraries, each library also supports all of its native callbacks and options.

Callback wrapping

By default all callback options are wrapped with the an anonymous function with the correct arguments. You can disable this behavior by supplying the `wrapCallbacks = false` in your options array.

7.5.3.1 Working with buffered scripts

One drawback to previous implementation of 'Ajax' type features was the scattering of script tags throughout your document, and the inability to buffer scripts added by elements in the layout. The new JsHelper if used correctly avoids both of those issues. It is recommended that you place `$this->Js->writeBuffer()` at the bottom of your layout file above the `</body>` tag. This will allow all scripts generated in layout elements to be output in one place. It should be noted that buffered scripts are handled separately from included script files.

writeBuffer($options = array())

Writes all Javascript generated so far to a code block or caches them to a file and returns a linked script.

Options

- `inline` - Set to true to have scripts output as a script block inline if `cache` is also true, a script link tag will be generated. (default true)
- `cache` - Set to true to have scripts cached to a file and linked in (default false)
- `clear` - Set to false to prevent script cache from being cleared (default true)
- `onDomReady` - wrap cached scripts in domready event (default true)
- `safe` - if an inline block is generated should it be wrapped in <![CDATA[...]]> (default true)

Creating a cache file with `writeBuffer()` requires that `webroot/js` be world writable and allows a browser to cache generated script resources for any page.

buffer($content)

Add `$content` to the internal script buffer.

getBuffer($clear = true)

Get the contents of the current buffer. Pass in false to not clear the buffer at the same time.

Buffering methods that are not normally buffered

Some methods in the helpers are buffered by default. The engines buffer the following methods by default:

- event
- sortable
- drag
- drop
- slider

Additionally you can force any other method in JsHelper to use the buffering. By appending an boolean to the end of the arguments you can force other methods to go into the buffer. For example the `each()` method does not normally buffer.

```
1.      $this->Js->each('alert("whoa!");', true);
```

The above would force the `each()` method to use the buffer. Conversely if you want a method that does buffer to not buffer, you can pass a `false` in as the last argument.

```
1.      $this->Js->event('click', 'alert("whoa!");', false);
```

This would force the event function which normally buffers to return its result.

7.5.4 Methods

The core Javascript Engines provide the same feature set across all libraries, there is also a subset of common options that are translated into library specific options. This is done to provide end developers with as unified an API as possible. The following list of methods are supported by all the Engines included in the CakePHP core. Whenever you see separate lists for `Options` and `Event Options` both sets of parameters are supplied in the `$options` array for the method.

object($data, $options = array())

Converts values into JSON. There are a few differences between this method and JavascriptHelper::object(). Most notably there is no affordance for `stringKeys` or `q` options found in the JavascriptHelper. Furthermore `$this->Js->object();` cannot make script tags.

Options:

- `prefix` - String prepended to the returned data.
- `postfix` - String appended to the returned data.

Example Use:

```
1.      $json = $this->Js->object($data);
```

sortable($options = array())

Sortable generates a javascript snippet to make a set of elements (usually a list) drag and drop sortable.

The normalized options are:

Options

- `containment` - Container for move action
- `handle` - Selector to handle element. Only this element will start sort action.

- `revert` - Whether or not to use an effect to move sortable into final position.
- `opacity` - Opacity of the placeholder
- `distance` - Distance a sortable must be dragged before sorting starts.

Event Options

- `start` - Event fired when sorting starts
- `sort` - Event fired during sorting
- `complete` - Event fired when sorting completes.

Other options are supported by each Javascript library, and you should check the documentation for your javascript library for more detailed information on its options and parameters.

Example use:

```
1.      $this->Js->get('#my-list');
2.          $this->Js->sortable(array(
3.              'distance' => 5,
4.              'containment' => 'parent',
5.              'start' => 'onStart',
6.              'complete' => 'onStop',
7.              'sort' => 'onSort',
8.              'wrapCallbacks' => false
9.          ));
```

Assuming you were using the jQuery engine, you would get the following code in your generated Javascript block:

```
1.      $("#myList").sortable({containment:"parent", distance:5, sort:onSort, start:onStart, stop:onStop});
```

request($url, $options = array())

Generate a javascript snippet to create an `XmlHttpRequest` or 'AJAX' request.

Event Options

- `complete` - Callback to fire on complete.
- `success` - Callback to fire on success.
- `before` - Callback to fire on request initialization.
- `error` - Callback to fire on request failure.

Options

- `method` - The method to make the request with defaults to GET in more libraries
- `async` - Whether or not you want an asynchronous request.
- `data` - Additional data to send.
- `update` - Dom id to update with the content of the request.
- `type` - Data type for response. 'json' and 'html' are supported. Default is html for most libraries.
- `evalScripts` - Whether or not <script> tags should be eval'ed.
- `dataExpression` - Should the `data` key be treated as a callback. Useful for supplying `$options['data']` as another Javascript expression.

Example use

```
1.      $this->Js->event('click',
2.      $this->Js->request(array(
3.      'action' => 'foo', param1), array(
4.      'async' => true,
5.      'update' => '#element')));
```

get($selector)

Set the internal 'selection' to a CSS selector. The active selection is used in subsequent operations until a new selection is made.

```
1.      $this->Js->get('#element');
```

The `JsHelper` now will reference all other element based methods on the selection of `#element`. To change the active selection, call `get()` again with a new element.

drag($options = array())

Make an element draggable.

Options

- `handle` - selector to the handle element.
- `snapGrid` - The pixel grid that movement snaps to, an array(x, y)
- `container` - The element that acts as a bounding box for the draggable element.

Event Options

- `start` - Event fired when the drag starts
- `drag` - Event fired on every step of the drag
- `stop` - Event fired when dragging stops (mouse release)

Example use

```
1.      $this->Js->get('#element');
2.      $this->Js->drag(array(
3.          'container' => '#content',
4.          'start' => 'onStart',
5.          'drag' => 'onDrag',
6.          'stop' => 'onStop',
7.          'snapGrid' => array(10, 10),
```

```
            'wrapCallbacks' => false
        ));
```

If you were using the jQuery engine the following code would be added to the buffer.

```
        $("#element").draggable({containment:"#content", drag:onDrag, grid:[10,10], start:onStart, stop:onStop});
```

drop($options = array())

Make an element accept draggable elements and act as a dropzone for dragged elements.

Options

- `accept` - Selector for elements this droppable will accept.
- `hoverclass` - Class to add to droppable when a draggable is over.

Event Options

- `drop` - Event fired when an element is dropped into the drop zone.
- `hover` - Event fired when a drag enters a drop zone.
- `leave` - Event fired when a drag is removed from a drop zone without being dropped.

Example use

```
        $this->Js->get('#element');
        $this->Js->drop(array(
            'accept' => '.items',
            'hover' => 'onHover',
            'leave' => 'onExit',
            'drop' => 'onDrop',
```

```
            'wrapCallbacks' => false
        ));
```

If you were using the jQuery engine the following code would be added to the buffer:

```
        <code class=
        "php">$("#element").droppable({accept:".items", drop:onDrop, out:onExit, over:onHover});</code>
```

"Note" about MootoolsEngine::drop

Droppables in Mootools function differently from other libraries. Droppables are implemented as an extension of Drag. So in addtion to making a get() selection for the droppable element. You must also provide a selector rule to the draggable element. Furthermore, Mootools droppables inherit all options from Drag.

slider()

Create snippet of Javascript that converts an element into a slider ui widget. See your libraries implementation for additional usage and features.

Options

- `handle` - The id of the element used in sliding.
- `direction` - The direction of the slider either 'vertical' or 'horizontal'
- `min` - The min value for the slider.
- `max` - The max value for the slider.
- `step` - The number of steps or ticks the slider will have.
- `value` - The initial offset of the slider.

Events

- `change` - Fired when the slider's value is updated

- `complete` - Fired when the user stops sliding the handle

Example use

```
1.      $this->Js->get('#element');
2.      $this->Js->slider(array(
3.          'complete' => 'onComplete',
4.          'change' => 'onChange',
5.          'min' => 0,
6.          'max' => 10,
7.          'value' => 2,
8.          'direction' => 'vertical',
9.          'wrapCallbacks' => false
10.     ));
```

If you were using the jQuery engine the following code would be added to the buffer:

```
1.      $("#element").slider({change:onChange, max:10, min:0, orientation:"vertical", stop:onComplete, value:2});
```

effect($name, $options = array())

Creates a basic effect. By default this method is not buffered and returns its result.

Supported effect names

The following effects are supported by all JsEngines

- `show` - reveal an element.
- `hide` - hide an element.
- `fadeIn` - Fade in an element.
- `fadeOut` - Fade out an element.

- `slideIn` - Slide an element in.
- `slideOut` - Slide an element out.

Options

- `speed` - Speed at which the animation should occur. Accepted values are 'slow', 'fast'. Not all effects use the speed option.

Example use

If you were using the jQuery engine.

```
1.      $this->Js->get('#element');
2.      $result = $this->Js->effect('fadeIn');
3.      //$result contains $("#foo").fadeIn();
```

event($type, $content, $options = array())

Bind an event to the current selection. `$type` can be any of the normal DOM events or a custom event type if your library supports them. `$content` should contain the function body for the callback. Callbacks will be wrapped with `function (event) { ... }` unless disabled with the `$options`.

Options

- `wrap` - Whether you want the callback wrapped in an anonymous function. (defaults to true)
- `stop` - Whether you want the event to stopped. (defaults to true)

Example use

```
1.      $this->Js->get('#some-link');
2.      $this->Js->event('click', $this->Js->alert('hey you!'));
```

If you were using the jQuery library you would get the following Javascript code.

```
1.      $('#some-link').bind('click', function (event) {
2.          alert('hey you!');
3.          return false;
4.      });
```

You can remove the `return false;` by passing setting the `stop` option to false.

```
1.      $this->Js->get('#some-link');
2.      $this->Js->event('click', $this->Js->alert('hey you!'), array('stop' => false));
```

If you were using the jQuery library you would the following Javascript code would be added to the buffer. Note that the default browser event is not cancelled.

```
1.      $('#some-link').bind('click', function (event) {
2.          alert('hey you!');
3.      });
```

domReady($callback)

Creates the special 'DOM ready' event. `writeBuffer()` automatically wraps the buffered scripts in a domReady method.

each($callback)

Create a snippet that iterates over the currently selected elements, and inserts `$callback`.

Example

```
1.      $this->Js->get('div.message');
2.      $this->Js->each('$(this).css({color: "red"});');
```

Using the jQuery engine would create the following Javascript

```
1.      $('div.message').each(function () { $(this).css({color: "red"});});
```

alert($message)

Create a javascript snippet containing an `alert()` snippet. By default, `alert` does not buffer, and returns the script snippet.

```
1.      $alert = $this->Js->alert('Hey there');
```

confirm($message)

Create a javascript snippet containing a `confirm()` snippet. By default, `confirm` does not buffer, and returns the script snippet.

```
1.      $alert = $this->Js->confirm('Are you sure?');
```

prompt($message, $default)

Create a javascript snippet containing a `prompt()` snippet. By default, `prompt` does not buffer, and returns the script snippet.

```
1.      $prompt = $this->Js->prompt('What is your favorite color?', 'blue');
```

submit()

Create a submit input button that enables `XmlHttpRequest` submitted forms. Options can include both those for FormHelper::submit() and JsBaseEngine::request(), JsBaseEngine::event();

Forms submitting with this method, cannot send files. Files do not transfer over `XmlHttpRequest` and require an iframe, or other more specialized setups that are beyond the scope of this helper.

Options

- `confirm` - Confirm message displayed before sending the request. Using confirm, does not replace any `before` callback methods in the generated XmlHttpRequest.
- `buffer` - Disable the buffering and return a script tag in addition to the link.
- `wrapCallbacks` - Set to false to disable automatic callback wrapping.

Example use

```
1.      echo $this->Js->submit('Save', array('update' => '#content'));
```

Will create a submit button with an attached onclick event. The click event will be buffered by default.

```
1.      echo $this->Js->submit('Save', array('update' => '#content', 'div' => false, 'type' => 'json', 'async' =>
false));
```

Shows how you can combine options that both `FormHelper::submit()` and `Js::request()` when using submit.

link($title, $url = null, $options = array())

Create an html anchor element that has a click event bound to it. Options can include both those for HtmlHelper::link() and JsBaseEngine::request(), JsBaseEngine::event(); `$htmlAttributes` is used to specify additional options that are supposed to be appended to the generated anchor element. If an option is not part of the standard attributes or `$htmlAttributes` it will be passed to `request()` as an option. If an id is not supplied, a randomly generated one will be created for each link generated.

Options

- `confirm` - Generate a confirm() dialog before sending the event.
- `id` - use a custom id.
- `htmlAttributes` - additional non-standard htmlAttributes. Standard attributes are class, id, rel, title, escape, onblur and onfocus.

- `buffer` - Disable the buffering and return a script tag in addition to the link.

Example use

```
1.        echo $this->Js->link('Page 2', array('page' => 2), array('update' => '#content'));
```

Will create a link pointing to `/page:2` and updating #content with the response.

You can use the `htmlAttributes` option to add in additional custom attributes.

```
1.        echo $this->Js->link('Page 2', array('page' => 2), array(
2.            'update' =&gt; '#content',
3.            'htmlAttributes' =&gt; array('other' =&gt; 'value')
4.        ));
5.        //Creates the following html
6.        <a href="/posts/index/page:2" other="value">Page 2</a>
```

serializeForm($options = array())

Serialize the form attached to $selector. Pass `true` for $isForm if the current selection is a form element. Converts the form or the form element attached to the current selection into a string/json object (depending on the library implementation) for use with XHR operations.

Options

- `isForm` - is the current selection a form, or an input? (defaults to false)
- `inline` - is the rendered statement going to be used inside another JS statement? (defaults to false)

Setting inline == false allows you to remove the trailing `;`. This is useful when you need to serialize a form element as part of another Javascript operation, or use the serialize method in an Object literal.

redirect($url)

Redirect the page to `$url` using `window.location`.

value($value)

Converts a PHP-native variable of any type to a JSON-equivalent representation. Escapes any string values into JSON compatible strings. UTF-8 characters will be escaped.

7.5.5 Ajax Pagination

Much like Ajax Pagination in 1.2, you can use the JsHelper to handle the creation of Ajax pagination links instead of plain HTML links.

7.5.5.1 Making Ajax Links

Before you can create ajax links you must include the Javascript library that matches the adapter you are using with `JsHelper`. By default the `JsHelper` uses jQuery. So in your layout include jQuery (or whichever library you are using). Also make sure to include `RequestHandlerComponent` in your components. Add the following to your controller:

```
1.      var $components = array('RequestHandler');
2.      var $helpers = array('Js');
```

Next link in the javascript library you want to use. For this example we'll be using jQuery.

```
1.      echo $this->Html->script('jquery');
```

Similar to 1.2 you need to tell the `PaginatorHelper` that you want to make Javascript enhanced links instead of plain HTML ones. To do so you use `options()`

```
1.      $this->Paginator->options(array(
2.          'update' => '#content',
3.          'evalScripts' => true
4.      ));
```

The `PaginatorHelper` now knows to make javascript enhanced links, and that those links should update the `#content` element. Of course this element must exist, and often times you want to wrap `$content_for_layout` with a div matching the id used for the `update` option. You also should set `evalScripts` to true if you are using the Mootools or Prototype adapters, without `evalScripts` these libraries will not be able to chain requests together. The `indicator` option is not supported by `JsHelper` and will be ignored.

You then create all the links as needed for your pagination features. Since the `JsHelper` automatically buffers all generated script content to reduce the number of `<script>` tags in your source code you **must** call write the buffer out. At the bottom of your view file. Be sure to include:

```
1.  echo $this->Js->writeBuffer();
```

If you omit this you will **not** be able to chain ajax pagination links. When you write the buffer, it is also cleared, so you don't have worry about the same Javascript being output twice.

Adding effects and transitions

Since `indicator` is no longer supported, you must add any indicator effects yourself.

```
1.  <html>
2.      <head>
3.          <?php echo $this->Html->script('jquery'); ?>
4.          //more stuff here.
5.      </head>
6.      <body>
7.      <div id="content">
8.          <?php echo $content_for_layout; ?>
9.      </div>
10.     <?php echo $this->Html->image('indicator.gif', array('id' => 'busy-indicator')); ?>
11.     </body>
12. </html>
```

Remember to place the indicator.gif file inside app/webroot/img folder. You may see a situation where the indicator.gif displays immediately upon the page load. You need to put in this css `#busy-indicator { display:none; }` in your main css file.

With the above layout, we've included an indicator image file, that will display a busy indicator animation that we will show and hide with the `JsHelper`. To do that we need to update our `options()` function.

```
1.      $this->Paginator->options(array(
2.          'update' => '#content',
3.          'evalScripts' => true,
4.          'before' => $this->Js->get('#busy-indicator')->effect('fadeIn', array('buffer' => false)),
5.          'complete' => $this->Js->get('#busy-indicator')->effect('fadeOut', array('buffer' => false)),
6.      ));
```

This will show/hide the busy-indicator element before and after the `#content` div is updated. Although `indicator` has been removed, the new features offered by `JsHelper` allow for more control and more complex effects to be created.

7.6 Javascript

The Javascript helper is used to aid in creating well formatted related javascript tags and codeblocks. There are several methods some of which are designed to work with the Prototype Javascript library.

The Javascript Helper is deprecated in 1.3 and will be removed in future versions of CakePHP. See the new JsHelper and HtmlHelper, as well as the migration guide for where JavascriptHelper's methods have moved to.

7.6.1 Methods

`codeBlock($script, $options = array('allowCache'=>true,'safe'=>true,'inline'=>true), $safe)`

- string $script - The JavaScript to be wrapped in SCRIPT tags
- array $options - Set of options:
    - allowCache: boolean, designates whether this block is cacheable using the current cache settings.
    - safe: boolean, whether this block should be wrapped in CDATA tags. Defaults to helper's object configuration.
    - inline: whether the block should be printed inline, or written to cached for later output (i.e. $scripts_for_layout).
- boolean $safe - DEPRECATED. Use $options['safe'] instead

codeBlock returns a formatted script element containing $script. But can also return null if Javascript helper is set to cache events. See JavascriptHelper::cacheEvents(). And can write in `$scripts_for_layout` if you set $options['inline'] to false.

`blockEnd()`

Ends a block of cached Javascript. Can return either a end script tag, or empties the buffer, adding the contents to the cachedEvents array. Its return value depends on the cache settings. See JavascriptHelper::cacheEvents()

`link($url, $inline = true)`

- mixed $url - String URL to JavaScript file, or an array of URLs.
- boolean $inline If true, the <script> tag will be printed inline, otherwise it will be printed in `$scripts_for_layout`

Creates a javascript link to a single or many javascript files. Can output inline or in `$scripts_for_layout`.

If the filename is prefixed with "/", the path will be relative to the base path of your application. Otherwise, the path will be relative to your JavaScript path, usually webroot/js.

`escapeString($string)`

- string $script - String that needs to get escaped.

Escape a string to be JavaScript friendly. Allowing it to safely be used in javascript blocks.

The characters that are escaped are:

- "\r\n" => '\n'
- "\r" => '\n'
- "\n" => '\n'
- "'" => '\''
- "'" => "\\'"

event($object, $event, $observer, $useCapture)

- string $object - DOM Object to be observed.
- string $event - type of event to observe ie 'click', 'over'.
- string $observer - Javascript function to call when event occurs.
- array $options - Set options: useCapture, allowCache, safe
    - boolean $options['useCapture'] - Whether to fire the event in the capture or bubble phase of event handling. Defaults false
    - boolean $options['allowCache'] - See JavascriptHelper::cacheEvents()
    - boolean $options['safe'] - Indicates whether <script /> blocks should be written 'safely,' i.e. wrapped in CDATA blocks

Attach a javascript event handler specified by $event to an element DOM element specified by $object. Object does not have to be an ID reference it can refer to any valid javascript object or CSS selectors. If a CSS selector is used the event handler is cached and should be retrieved with JavascriptHelper::getCache(). This method requires the Prototype library.

cacheEvents($file, $all)

- boolean $file - If true, code will be written to a file
- boolean $all - If true, all code written with JavascriptHelper will be sent to a file

Allows you to control how the JavaScript Helper caches event code generated by event(). If $all is set to true, all code generated by the helper is cached and can be retrieved with getCache() or written to file or page output with writeCache().

getCache($clear)

- boolean $clear - If set to true the cached javascript is cleared. Defaults to true.

Gets (and clears) the current JavaScript event cache

writeEvents($inline)

- boolean $inline - If true, returns JavaScript event code. Otherwise it is added to the output of $scripts_for_layout in the layout.

Returns cached javascript code. If $file was set to true with cacheEvents(), code is cached to a file and a script link to the cached events file is returned. If inline is true, the event code is returned inline. Else it is added to the $scripts_for_layout for the page.

includeScript($script)

- string $script - File name of script to include.

Includes the named $script. If $script is left blank the helper will include every script in your app/webroot/js directory. Includes the contents of each file inline. To create a script tag with an src attribute use link().

object($data, $options)

- array $data - Data to be converted
- array $options - Set of options: block, prefix, postfix, stringKeys, quoteKeys, q
    - boolean $options['block'] - Wraps return value in a <script /> block if true. Defaults to false.
    - string $options['prefix'] - Prepends the string to the returned data.
    - string $options['postfix'] - Appends the string to the returned data.
    - array $options['stringKeys'] - A list of array keys to be treated as a string.
    - boolean $options['quoteKeys'] - If false, treats $stringKey as a list of keys *not* to be quoted. Defaults to true.
    - string $options['q'] - The type of quote to use.

Generates a JavaScript object in JavaScript Object Notation (JSON) from $data array.

7.7 Number

The NumberHelper contains convenience methods that enable display numbers in common formats in your views. These methods include ways to format currency, percentages, data sizes, format numbers to specific precisions and also to give you more flexibility with formating numbers.

All of these functions return the formated number; They do not automatically echo the output into the view.

7.7.1 currency

```
currency(mixed $number, string $currency= 'USD', $options = array())
```

This method is used to display a number in common currency formats (EUR,GBP,USD). Usage in a view looks like:

```
1.      <?php echo $this->Number->currency($number,$currency); ?>
```

The first parameter, $number, should be a floating point number that represents the amount of money you are expressing. The second parameter is used to choose a predefined currency formatting scheme:

$currency	1234.56, formatted by currency type
EUR	€ 1.236,33
GBP	£ 1,236.33
USD	$ 1,236.33

The third parameter is an array of options for further defining the output. The following options are available:

Option	Description
before	The currency symbol to place before whole numbers ie. '$'

Option	Description
after	The currency symbol to place after decimal numbers ie. 'c'. Set to boolean false to use no decimal symbol. eg. 0.35 => $0.35.
zero	The text to use for zero values, can be a string or a number. ie. 0, 'Free!'
places	Number of decimal places to use. ie. 2
thousands	Thousands separator ie. ','
decimals	Decimal separator symbol ie. '.'
negative	Symbol for negative numbers. If equal to '()', the number will be wrapped with (and)
escape	Should the output be htmlentity escaped? Defaults to true

If a non-recognized $currency value is supplied, it is prepended to a USD formatted number. For example:

```
1.  <?php echo $this->Number->currency('1234.56', 'FOO'); ?>
2.
3.  //Outputs:
```

```
4.      FOO 1,234.56
```

7.7.2 precision

precision (mixed $number, int $precision = 3)

This method displays a number with the specified amount of precision (decimal places). It will round in order to maintain the level of precision defined.

```
1.      <?php echo $this->Number->precision(456.91873645, 2 ); ?>
2.
3.      //Outputs:
4.      456.92
```

7.7.3 toPercentage

toPercentage(mixed $number, int $precision = 2)

Like precision(), this method formats a number according to the supplied precision (where numbers are rounded to meet the given precision). This method also expresses the number as a percentage and prepends the output with a percent sign.

```
1.      <?php echo $this->Number->toPercentage(45.691873645); ?>
2.
3.      //Outputs:
4.      45.69%
```

7.7.4 toReadableSize

toReadableSize(string $data_size)

This method formats data sizes in human readable forms. It provides a shortcut way to convert bytes to KB, MB, GB, and TB. The size is displayed with a two-digit precision level, according to the size of data supplied (i.e. higher sizes are expressed in larger terms):

```
1.      echo $this->Number->toReadableSize(0);  // 0 Bytes
2.      echo $this->Number->toReadableSize(1024); // 1 KB
3.      echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
4.      echo $this->Number->toReadableSize(5368709120); // 5.00 GB
```

7.7.5 format

format (mixed $number, mixed $options=false)

This method gives you much more control over the formatting of numbers for use in your views (and is used as the main method by most of the other NumberHelper methods). Using this method might looks like:

```
1.      $this->Number->format($number, $options);
```

The $number parameter is the number that you are planning on formatting for output. With no $options supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The $options parameter is where the real magic for this method resides.

- If you pass an integer then this becomes the amount of precision or places for the function.
- If you pass an associated array, you can use the following keys:
    - places (integer): the amount of desired precision
    - before (string): to be put before the outputted number
    - escape (boolean): if you want the value in before to be escaped
    - decimals (string): used to delimit the decimal places in a number
    - thousands (string): used to mark off thousand, millions, … places

```
1.      echo $this->Number->format('123456.7890', array(
2.          'places' => 2,
3.          'before' => '¥ ',
```

```
4.              'escape' => false,
5.              'decimals' => '.',
6.              'thousands' => ','
7.          ));
8.          // output '¥ 123,456.79'
```

7.8 Paginator

The Pagination helper is used to output pagination controls such as page numbers and next/previous links.

See also Common Tasks With CakePHP - Pagination for additional information.

7.8.1 Methods

options($options = array())

- mixed options() Default options for pagination links. If a string is supplied - it is used as the DOM id element to update. See #options for list of keys.

options() sets all the options for the Paginator Helper. Supported options are:

format

Format of the counter. Supported formats are 'range' and 'pages' and custom which is the default. In the default mode the supplied string is parsed and tokens are replaced with actual values. The available tokens are:

- %page% - the current page displayed.
- %pages% - total number of pages.
- %current% - current number of records being shown.
- %count% - the total number of records in the result set.
- %start% - number of the first record being displayed.
- %end% - number of the last record being displayed.

Now that you know the available tokens you can use the counter() method to display all sorts of information about the returned results, for example:

```
1.      echo $this->Paginator->counter(array(
2.              'format' => 'Page %page% of %pages%,
3.                          showing %current% records out of %count% total,
4.                          starting on record %start%, ending on %end%'
5.      ));
```

separator

The separator between the actual page and the number of pages. Defaults to ' of '. This is used in conjunction with format = 'pages'

url

The url of the paginating action. url has a few sub options as well

- sort - the key that the records are sorted by
- direction - The direction of the sorting. Defaults to 'ASC'
- page - The page number to display

model

The name of the model being paginated.

escape

Defines if the title field for links should be HTML escaped. Defaults to true.

update

The DOM id of the element to update with the results of AJAX pagination calls. If not specified, regular links will be created.

indicator

DOM id of the element that will be shown as a loading or working indicator while doing AJAX requests.

link($title, $url = array(), $options = array())

- string $title - Title for the link.
- mixed $url Url for the action. See Router::url()
- array $options Options for the link. See options() for list of keys.

Creates a regular or AJAX link with pagination parameters

```
1.        echo $this->Paginator->link('Sort by title on page 5',
2.                array('sort' => 'title', 'page' => 5, 'direction' => 'desc'));
```

If created in the view for `/posts/index` Would create a link pointing at '/posts/index/page:5/sort:title/direction:desc'

7.9 RSS

The RSS helper makes generating XML for RSS feeds easy.

7.9.1 Creating an RSS feed with the RssHelper

This example assumes you have a Posts Controller and Post Model already created and want to make an alternative view for RSS.

Creating an xml/rss version of posts/index is a snap with CakePHP 1.3. After a few simple steps you can simply append the desired extension .rss to posts/index making your URL posts/index.rss. Before we jump too far ahead trying to get our webservice up and running we need to do a few things. First parseExtensions needs to be activated, this is done in app/config/routes.php

```
1.            Router::parseExtensions('rss');
```

In the call above we've activated the .rss extension. When using Router::parseExtensions() you can pass as many arguments or extensions as you want. This will activate each extension/content-type for use in your application. Now when the address posts/index.rss is requested you will get an xml version of your posts/index. However, first we need to edit the controller to add in the rss-specific code.

7.9.1.1 Controller Code

It is a good idea to add RequestHandler to your PostsController's $components array. This will allow a lot of automagic to occur.

```
1.          var $components = array('RequestHandler');
```

Our view will also use the TextHelper for formatting, so that should be added to the controller as well.

```
1.          var $helpers = array('Text');
```

Before we can make an RSS version of our posts/index we need to get a few things in order. It may be tempting to put the channel metadata in the controller action and pass it to your view using the Controller::set() method but this is inappropriate. That information can also go in the view. That will come later though, for now if you have a different set of logic for the data used to make the RSS feed and the data for the html view you can use the RequestHandler::isRss() method, otherwise your controller can stay the same.

```
1.      // Modify the Posts Controller action that corresponds to
2.      // the action which deliver the rss feed, which is the
3.      // index action in our example
4.      public function index(){
5.          if( $this->RequestHandler->isRss() ){
6.              $posts = $this->Post->find('all', array('limit' => 20, 'order' => 'Post.created DESC'));
7.              return $this->set(compact('posts'));
8.          }
9.          // this is not an Rss request, so deliver
10.         // data used by website's interface
11.         $this->paginate['Post'] = array('order' => 'Post.created DESC', 'limit' => 10);
12.
13.         $posts = $this->paginate();
14.         $this->set(compact('posts'));
15.     }
```

With all the View variables set we need to create an rss layout.

7.9.1.1.1 Layout

An Rss layout is very simple, put the following contents in `app/views/layouts/rss/default.ctp`:

```
1.      echo $this->Rss->header();
2.      if (!isset($documentData)) {
3.          $documentData = array();
4.      }
5.      if (!isset($channelData)) {
6.          $channelData = array();
7.      }
8.      if (!isset($channelData['title'])) {
9.          $channelData['title'] = $title_for_layout;
10.     }
11.     $channel = $this->Rss->channel(array(), $channelData, $content_for_layout);
12.     echo $this->Rss->document($documentData,$channel);
```

It doesn't look like much but thanks to the power in the RssHelper its doing a lot of lifting for us. We haven't set $documentData or $channelData in the controller, however in CakePHP 1.3 your views can pass variables back to the layout. Which is where our $channelData array will come from setting all of the meta data for our feed.

Next up is view file for my posts/index. Much like the layout file we created, we need to create a views/posts/rss/ directory and create a new index.ctp inside that folder. The contents of the file are below.

7.9.1.1.2 View

Our view, located at `app/views/posts/rss/index.ctp`, begins by setting the $documentData and $channelData variables for the layout, these contain all the metadata for our RSS feed. This is done by using the View::set() method which is analogous to the Controller::set() method. Here though we are passing the channel's metadata back to the layout.

```
1.          $this->set('documentData', array(
```

```
2.                 'xmlns:dc' => 'http://purl.org/dc/elements/1.1/'));
3.             $this->set('channelData', array(
4.                 'title' => __("Most Recent Posts", true),
5.                 'link' => $this->Html->url('/', true),
6.                 'description' => __("Most recent posts.", true),
7.                 'language' => 'en-us'));
```

The second part of the view generates the elements for the actual records of the feed. This is accomplished by looping through the data that has been passed to the view ($items) and using the RssHelper::item() method. The other method you can use, RssHelper::items() which takes a callback and an array of items for the feed. (The method I have seen used for the callback has always been called transformRss(). There is one downfall to this method, which is that you cannot use any of the other helper classes to prepare your data inside the callback method because the scope inside the method does not include anything that is not passed inside, thus not giving access to the TimeHelper or any other helper that you may need. The RssHelper::item() transforms the associative array into an element for each key value pair.

You will need to modify the $postLink variable as appropriate to your application.

```
1.             foreach ($posts as $post) {
2.                 $postTime = strtotime($post['Post']['created']);
3.
4.                 $postLink = array(
5.                     'controller' => 'posts',
6.                     'action' => 'view',
7.                     'year' => date('Y', $postTime),
8.                     'month' => date('m', $postTime),
9.                     'day' => date('d', $postTime),
10.                    $post['Post']['slug']);
11.                // You should import Sanitize
12.                App::import('Sanitize');
13.                // This is the part where we clean the body text for output as the description
14.                // of the rss item, this needs to have only text to make sure the feed validates
```

```
15.             $bodyText = preg_replace('=\(.*?\)=is', '', $post['Post']['body']);
16.             $bodyText = $this->Text->stripLinks($bodyText);
17.             $bodyText = Sanitize::stripAll($bodyText);
18.             $bodyText = $this->Text->truncate($bodyText, 400, array(
19.                 'ending' => '...',
20.                 'exact'  => true,
21.                 'html'   => true,
22.             ));
23.
24.             echo  $this->Rss->item(array(), array(
25.                 'title' => $post['Post']['title'],
26.                 'link' => $postLink,
27.                 'guid' => array('url' => $postLink, 'isPermaLink' => 'true'),
28.                 'description' =>  $bodyText,
29.                 'dc:creator' => $post['Post']['author'],
30.                 'pubDate' => $post['Post']['created']));
31.         }
```

You can see above that we can use the loop to prepare the data to be transformed into XML elements. It is important to filter out any non-plain text characters out of the description, especially if you are using a rich text editor for the body of your blog. In the code above we use the TextHelper::stripLinks() method and a few methods from the Sanitize class, but we recommend writing a comprehensive text cleaning helper to really scrub the text clean. Once we have set up the data for the feed, we can then use the RssHelper::item() method to create the XML in RSS format. Once you have all this setup, you can test your RSS feed by going to your site /posts/index.rss and you will see your new feed. It is always important that you validate your RSS feed before making it live. This can be done by visiting sites that validate the XML such as Feed Validator or the w3c site at http://validator.w3.org/feed/.

You may need to set the value of 'debug' in your core configuration to 1 or to 0 to get a valid feed, because of the various debug information added automagically under higher debug settings that break XML syntax or feed validation rules.

7.10 Session

As a natural counterpart to the Session Component, the Session Helper replicates most of the components functionality and makes it available in your view. The Session Helper is no longer automatically added to your view — so it is necessary to add it to the `$helpers` array in the controller.

The major difference between the Session Helper and the Session Component is that the helper does *not* have the ability to write to the session.

As with the Session Component, data is written to and read by using dot separated array structures.

```
1.          array('User' =>
2.                  array('username' => 'super@example.com')
3.          );
```

Given the previous array structure, the node would be accessed by User.username, with the dot indicating the nested array. This notation is used for all Session helper methods wherever a $key is used.

If you have `Session.start` set to false in your `config/core.php`, you need to call `$session->activate();` in your view before you can use any other method of Session helper. Just like you need to call `$this->Session->activate();` in your controller to activate Session component.

7.10.1 Methods

read($key)	Read from the Session. Returns a string or array depending on the contents of the session.
id()	Returns the current session ID.
check($key)	Check to see if a key is in the Session. Returns a boolean on the key's existence.
flash($key)	This will return the contents of the $_SESSION.Message. It is used in conjunction with the Session Component's setFlash() method.
error()	Returns the last error in the session if one exists.

7.10.2 flash

The flash method uses the default key set by `setFlash()`. You can also retrieve specific keys in the session. For example, the Auth component sets all of its Session messages under the 'auth' key

```
1.      // Controller code
2.      $this->Session->setFlash('My Message');
3.      // In view
4.      echo $this->Session->flash();
5.      // outputs "<div id='flashMessage' class='message'>My Message</div>"
6.      // output the AuthComponent Session message, if set.
7.      echo $this->Session->flash('auth');
```

 # Using Flash for Success and Failure

In some web sites, particularly administration backoffice web applications it is often expected that the result of an operation requested by the user has associated feedback as to whether the operation succeeded or not. This is a classic usage for the flash mechanism since we only want to show the user the result once and not keep the message.

One way to achieve this is to use Session->flash() with the layout parameter. With the layout parameter we can be in control of the resultant html for the message.

In the controller you might typically have code:

```
1.      if ($user_was_deleted) {
2.          $this->Session->setFlash('The user was deleted successfully.', 'flash_success');
3.      } else {
4.          $this->Session->setFlash('The user could not be deleted.', 'flash_failure');
5.      }
```

The flash_success and flash_failure parameter represents an element file to place in the root app/views/elements folder, e.g. app/views/elements/flash_success.ctp, app/views/elements/flash_failure.ctp

Inside the flash_success element file would be something like this:

```
1.      <div class="flash flash_success">
2.          <?php echo $message ?>
3.      </div>
```

The final step is in your main view file where the result is to be displayed to add simply

```
1.      <?php echo $this->Session->flash(); ?>
```

And of course you can then add to your CSS a selector for div.flash, div.flash_success and div.flash_failure

7.11 Text

The TextHelper contains methods to make text more usable and friendly in your views. It aids in enabling links, formatting urls, creating excerpts of text around chosen words or phrases, highlighting key words in blocks of text, and to gracefully truncating long stretches of text.

autoLinkEmails

autoLinkEmails(string $text, array $htmlOptions=array())

Adds links to the well-formed email addresses in $text, according to any options defined in `$htmlOptions` (see HtmlHelper::link()).

```
1.      $my_text = 'For more information regarding our world-famous pastries and desserts, contact info@example.com';
2.      $linked_text = $this->Text->autoLinkEmails($my_text);
```

Output:

```
For more information regarding our world-famous pastries and desserts,
contact <a href="mailto:info@example.com">info@example.com</a>
```

autoLinkUrls

autoLinkUrls(string $text, array $htmlOptions=array())

Same as in `autoLinkEmails()`, only this method searches for strings that start with https, http, ftp, or nntp and links them appropriately.

autoLink

autoLink(string $text, array $htmlOptions=array())

Performs the functionality in both `autoLinkUrls()` and `autoLinkEmails()` on the supplied `$text`. All URLs and emails are linked appropriately given the supplied `$htmlOptions`.

excerpt

excerpt(string $haystack, string $needle, int $radius=100, string $ending="...")

Extracts an excerpt from `$haystack` surrounding the `$needle` with a number of characters on each side determined by `$radius`, and suffixed with `$ending`. This method is especially handy for search results. The query string or keywords can be shown within the resulting document.

```
1.          echo $this->Text->excerpt($last_paragraph, 'method', 50);
```

Output:

```
mined by $radius, and suffixed with $ending. This method is especially handy for
search results. The query...
```

highlight

highlight(string $haystack, string $needle, array $options = array())

Highlights `$needle` in `$haystack` using the `$options['format']` string specified or a default string.

Options

- 'format' - string The piece of html with that the phrase will be highlighted

- 'html' - bool If true, will ignore any HTML tags, ensuring that only the correct text is highlighted

```
1.          echo                $this->Text->highlight($last_sentence,                'using',array('format'=>'<span
class="highlight">\1</span>');
```

Output:

```
Highlights $needle in $haystack <span class="highlight">using</span>
the $options['format'] string specified  or a default string.
```

 # stripLinks

```
stripLinks($text)
```

Strips the supplied `$text` of any HTML links.

 # toList

```
toList(array $list, $and='and')
```

Creates a comma-separated list where the last two items are joined with ‘and’.

```
1.          echo $this->Text->toList($colors);
```

Output:

```
red, orange, yellow, green, blue, indigo and violet
```

 # truncate

```
truncate(string $text, int $length=100, array $options)
```

Cuts a string to the `$length` and adds a suffix with `'ending'` if the text is longer than `$length`. If `'exact'` is passed as `false`, the truncation will occur after the next word ending. If `'html'` is passed as `true`, html tags will be respected and will not be cut off.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
        array(
            'ending' => '...',
            'exact' => true,
            'html' => false
        )
        echo $this->Text->truncate(
            'The killer crept forward and tripped on the rug.',
            22,
            array(
                'ending' => '...',
                'exact' => false
            )
        );
```

Output:

```
The killer crept...
```

 # trim

```
trim()
```

An alias for truncate.

7.12 Time

The Time Helper does what it says on the tin: saves you time. It allows for the quick processing of time related information. The Time Helper has two main tasks that it can perform:

1. It can format time strings.
2. It can test time (but cannot bend time, sorry).

7.12.1 Formatting

```
fromString( $date_string )
```

fromString takes a string and uses strtotime to convert it into a date object. If the string passed in is a number then it'll convert it into an integer, being the number of seconds since the Unix Epoch (January 1 1970 00:00:00 GMT). Passing in a string of "20081231" will create undesired results as it will covert it to the number of seconds from the Epoch, in this case "Fri, Aug 21st 1970, 06:07"

```
toQuarter( $date_string, $range = false )
```

toQuarter will return 1, 2, 3 or 4 depending on what quarter of the year the date falls in. If range is set to true, a two element array will be returned with start and end dates in the format "2008-03-31".

```
toUnix( $date_string )
```

toUnix is a wrapper for fromString.

```
toAtom( $date_string )
```

toAtom return a date string in the Atom format "2008-01-12T00:00:00Z"

```
toRSS( $date_string )
```

toRSS returns a date string in the RSS format "Sat, 12 Jan 2008 00:00:00 -0500"

```
nice( $date_string = null )
```

nice takes a date string and outputs it in the format "Tue, Jan 1st 2008, 19:25".

```
niceShort( $date_string = null )
```

niceShort takes a date string and outputs it in the format "Jan 1st 2008, 19:25". If the date object is today, the format will be "Today, 19:25". If the date object is yesterday, the format will be "Yesterday, 19:25".

```
daysAsSql( $begin, $end, $fieldName, $userOffset = NULL )
```

daysAsSql returns a string in the format "($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-25 23:59:59')". This is handy if you need to search for records between two dates inclusively.

```
dayAsSql( $date_string, $field_name )
```

dayAsSql creates a string in the same format as daysAsSql but only needs a single date object.

```
timeAgoInWords( $datetime_string, $options = array(), $backwards = null )
```

timeAgoInWords will take a datetime string (anything that is parsable by PHP's strtotime() function or MySQL's datetime format) and convert it into a friendly word format like, "3 weeks, 3 days ago". Passing in true for $backwards will specifically declare the time is set in the future, which uses the format "on 31/12/08".

Option	Description
format	a date format; default "on 31/12/08"
end	determines the cutoff point in which it no longer uses words and uses the date format instead; default "+1 month"

```
relativeTime( $date_string, $format = 'j/n/y' )
```

relativeTime is essentially an alias for timeAgoInWords.

```
gmt( $date_string = null )
```

gmt will return the date as an integer set to Greenwich Mean Time (GMT).

```
format( $format = 'd-m-Y', $date_string)
```

format is a wrapper for the PHP date function.

Format	Sample Output
nice	Tue, Jan 1st 2008, 19:25
niceShort	Jan 1st 2008, 19:25 Today, 19:25 Yesterday, 19:25
daysAsSql	($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-25 23:59:59')
dayAsSql	($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-21 23:59:59')
timeAgoInWords relativeTime	on 21/01/08 3 months, 3 weeks, 2 days ago 7 minutes ago 2 seconds ago
gmt	1200787200

7.12.2 Testing Time

- `isToday`
- `isThisWeek`
- `isThisMonth`
- `isThisYear`
- `wasYesterday`
- `isTomorrow`
- `wasWithinLast`

All of the above functions return true or false when passed a date string. `wasWithinLast` takes an additional `$time_interval` option:

```
$this->Time->wasWithinLast( $time_interval, $date_string )
```

`wasWithinLast` takes a time interval which is a string in the format "3 months" and accepts a time interval of seconds, minutes, hours, days, weeks, months and years (plural and not). If a time interval is not recognized (for example, if it is mistyped) then it will default to days.

7.13 XML

The XML Helper simplifies the output of XML documents.

7.13.1 serialize

```
serialize($data, $options = array())
```

- mixed $data - The content to be converted to XML
- mixed $options - The data formatting options. For a list of valid options, see `Xml::__construct()`
    - string $options['root'] - The name of the root element, defaults to '#document'
    - string $options['version'] - The XML version, defaults to '1.0'
    - string $options['encoding'] - Document encoding, defaults to 'UTF-8'
    - array $options['namespaces'] - An array of namespaces (as strings) used in this document
    - string $options['format'] - Specifies the format this document converts to when parsed or rendered out as text, either 'attributes' or 'tags', defaults to 'attributes'
    - array $options['tags'] - An array specifying any tag-specific formatting options, indexed by tag name. See XmlNode::normalize()

The serialize method takes an array and creates an XML string of the data. This is commonly used for serializing model data.

```
<?php
echo $this->Xml->serialize($data);
 format will be similar to:
 <model_name id="1" field_name="content" />
```

```
5.      ?>
```

The serialize method acts as a shortcut to instantiating the XML built-in class and using the toString method of that. If you need more control over serialization, you may wish to invoke the XML class directly.

You can modify how a data is serialized by using the *format* attribute. By default the data will be serialized as attributes. If you set the *format* as "tags" the data will be serialized as tags.

```
1.      pr($data);
```

```
Array
(
    [Baker] => Array
        (
            [0] => Array
                (
                    [name] => The Baker
                    [weight] => heavy
                )
            [1] => Array
                (
                    [name] => The Cook
                    [weight] => light-weight
                )
        )
)
```

```
1.      pr($this->Xml->serialize($data));
```

```
<baker>
     <baker name="The Baker" weight="heavy" />
     <baker name="The Cook" weight="light-weight" />
</baker>
```

```
1.      pr($this->Xml->serialize($data, array('format' => 'tags')));
```

```
<baker>
```

```
    <baker>
        <name><![CDATA[The Baker]]></name>
        <weight><![CDATA[heavy]]></weight>
    </baker>
    <baker>
        <name><![CDATA[The Cook]]></name>
        <weight><![CDATA[light-weight]]></weight>
    </baker>
</baker>
```

7.13.2 elem

The elem method allows you to build an XML node string with attributes and internal content, as well.

string elem (string $name, $attrib = array(), mixed $content = null, $endTag = true)

```
1.  echo $this->Xml->elem('count', array('namespace' => 'myNameSpace'), 'content');
2.  // generates: <myNameSpace:count>content</count>
```

If you want to wrap your text node with CDATA, the third argument should be an array containing two keys: 'cdata' and 'value'

```
1.  echo $this->Xml->elem('count', null, array('cdata'=>true,'value'=>'content'));
2.  // generates: <count><![CDATA[content]]></count>
```

7.13.3 header

The `header()` method is used to output the XML declaration.

```
1.  <?php
2.  echo $this->Xml->header();
3.  // generates: <?xml version="1.0" encoding="UTF-8" ?>
```

```
4.        ?>
```

You can pass in a different version number and encoding type as parameters of the header method.

```
1.        <?php
2.        echo $this->Xml->header(array('version'=>'1.1'));
3.        // generates: <?xml version="1.1" encoding="UTF-8" ?>
4.        ?>
```

8 Core Utility Libraries

CakePHP includes general-purpose utility libraries which can be called from anywhere in your application, such as Set and HttpSocket.

8.1 App

App is a very small utility library. It only contains the **import** method. But, with the import method, you can accomplish a lot.

```
2.        // examples
3.        App::Import('Core','File');
4.        App::Import('Model','Post');
5.        App::import('Vendor', 'geshi');
6.        App::import('Vendor', 'flickr/flickr');
7.        App::import('Vendor', 'SomeName', array('file' => 'some.name.php'));
8.        App::import('Vendor', 'WellNamed', array('file' => 'services'.DS.'well.named.php'));
```

You can read more about it in the book or the API documentation

8.2 Inflector

The Inflector class takes a string and can manipulate it to handle word variations such as pluralizations or camelizing and is normally accessed statically.

Example: `Inflector::pluralize('example')` returns "examples".

8.2.1 Class methods

	Input	Output
pluralize	Apple, Orange, Person, Man	Apples, Oranges, People, Men
singularize	Apples, Oranges, People, Men	Apple, Orange, Person, Man
camelize	Apple_pie, some_thing, people_person	ApplePie, SomeThing, PeoplePerson
underscore	It should be noted that underscore will only convert camelCase formatted words. Words that contains spaces will be lower-cased, but will not contain an underscore.	
	applePie, someThing	apple_pie, some_thing
humanize	apple_pie, some_thing, people_person	Apple Pie, Some Thing, People Person
tableize	Apple, UserProfileSetting, Person	apples, user_profile_settings, people
classify	apples, user_profile_settings, people	Apple, UserProfileSetting, Person
variable	apples, user_result, people_people	apples, userResult, peoplePeople
slug	Slug converts special characters into latin versions and converting unmatched characters and spaces to underscores. The slug method expects UTF-8 encoding.	
	apple purée	apple_puree

8.3 String

The String class includes convenience methods for creating and manipulating strings and is normally accessed statically. Example: `String::uuid()`.

8.3.1 uuid

The uuid method is used to generate unique identifiers as per RFC 4122. The uuid is a 128bit string in the format of 485fc381-e790-47a3-9794-1337c0a8fe68.

```
1.      String::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

8.3.2 tokenize

`string tokenize ($data, $separator = ',', $leftBound = '(', $rightBound = ')')`

Tokenizes a string using `$separator`, ignoring any instance of `$separator` that appears between `$leftBound` and `$rightBound`.

8.3.3 insert

string insert ($string, $data, $options = array())

The insert method is used to create string templates and to allow for key/value replacements.

```
1.      String::insert('My name is :name and I am :age years old.', array('name' => 'Bob', 'age' => '65'));
2.      // generates: "My name is Bob and I am 65 years old."
```

8.3.4 cleanInsert

string cleanInsert ($string, $options = array())

Cleans up a Set::insert formatted string with given $options depending on the 'clean' key in $options. The default method used is text but html is also available. The goal of this function is to replace all whitespace and unneeded markup around placeholders that did not get replaced by Set::insert.

You can use the following options in the options array:

```
        $options = array(
            'clean' => array(
                'method' => 'text', // or html
            ),
            'before' => '',
            'after' => ''
        );
```

8.4 Xml

The Xml class provides an easy way to parse and generate XML fragments and documents. It is an all PHP solution and requires only the Xml/Expat extension to be installed. This library is provided mostly as a convenience to those that do not have access to SimpleXML.

8.4.1 Xml parsing

Parsing Xml with the Xml class requires you to have a string containing the xml you wish to parse.

```
        $input = '<' . '?xml version="1.0" encoding="UTF-8" ?' . '>
            <container>
                <element id="first-el">
                    <name>My element</name>
                    <size>20</size>
                </element>
                <element>
                    <name>Your element</name>
                    <size>30</size>
                </element>
            </container>';
        $xml =  new Xml($input);
```

This would create an Xml document object that can then be manipulated and traversed, and reconverted back into a string.

With the sample above you could do the following.

```
1.        echo $xml->children[0]->children[0]->name;
2.        // outputs 'element'
3.        echo $xml->children[0]->children[0]->children[0]->children[0]->value;
4.        // outputs 'My Element'
5.        echo $xml->children[0]->child('element')->attributes['id'];
6.        //outputs 'first-el'
```

In addition to the above it often makes it easier to obtain data from XML if you convert the Xml document object to a array.

```
4.        $xml = new Xml($input);
5.        // This converts the Xml document object to a formatted array
6.        $xmlAsArray = Set::reverse($xml);
7.        // Or you can convert simply by calling toArray();
8.        $xmlAsArray = $xml->toArray();
```

8.5 Set

Array management, if done right, can be a very powerful and useful tool for building smarter, more optimized code. CakePHP offers a very useful set of static utilities in the Set class that allow you to do just that.

CakePHP's Set class can be called from any model or controller in the same way Inflector is called. Example: Set::combine().

8.5.1 Set-compatible Path syntax

The Path syntax is used by (for example) sort, and is used to define a path.

Usage example (using Set::sort()):

```
$a = array(
    0 => array('Person' => array('name' => 'Jeff')),
    1 => array('Shirt' => array('color' => 'black'))
);
$result = Set::sort($a, '{n}.Person.name', 'asc');
/* $result now looks like:
    Array
    (
        [0] => Array
            (
                [Shirt] => Array
                    (
                        [color] => black
                    )
            )
        [1] => Array
            (
                [Person] => Array
                    (
                        [name] => Jeff
                    )
            )
    )
*/
```

As you can see in the example above, some things are wrapped in {}'s, others not. In the table below, you can see which options are available.

Expression	Definition
{n}	Represents a numeric key

{s}	Represents a string
Foo	Any string (without enclosing brackets) is treated like a string literal.
{[a-z]+}	Any string enclosed in brackets (besides {n} and {s}) is interpreted as a regular expression.

This section needs to be expanded.

8.5.2 insert

```
array Set::insert ($list, $path, $data = null)
```

Inserts $data into an array as defined by $path.

```
1.      $a = array(
2.          'pages' => array('name' => 'page')
3.      );
4.      $result = Set::insert($a, 'files', array('name' => 'files'));
5.      /* $result now looks like:
6.          Array
7.          (
8.              [pages] => Array
9.                  (
10.                     [name] => page
11.                 )
12.             [files] => Array
13.                 (
14.                     [name] => files
15.                 )
16.         )
17.     */
18.     $a = array(
```

```
19.             'pages' => array('name' => 'page')
20.         );
21.         $result = Set::insert($a, 'pages.name', array());
22.         /* $result now looks like:
23.             Array
24.             (
25.                 [pages] => Array
26.                     (
27.                         [name] => Array
28.                             (
29.                             )
30.                     )
31.             )
32.         */
33.         $a = array(
34.             'pages' => array(
35.                 0 => array('name' => 'main'),
36.                 1 => array('name' => 'about')
37.             )
38.         );
39.         $result = Set::insert($a, 'pages.1.vars', array('title' => 'page title'));
40.         /* $result now looks like:
41.             Array
42.             (
43.                 [pages] => Array
44.                     (
45.                         [0] => Array
46.                             (
47.                                 [name] => main
48.                             )
49.                         [1] => Array
```

```
                (
                    [name] => about
                    [vars] => Array
                        (
                            [title] => page title
                        )
                )
            )
        )
    */
```

8.5.3 sort

array Set::sort ($data, $path, $dir)

Sorts an array by any value, determined by a Set-compatible path.

```
    $a = array(
        0 => array('Person' => array('name' => 'Jeff')),
        1 => array('Shirt' => array('color' => 'black'))
    );
    $result = Set::sort($a, '{n}.Person.name', 'asc');
    /* $result now looks like:
        Array
        (
            [0] => Array
                (
                    [Shirt] => Array
                        (
                            [color] => black
```

```
                            )
                    )
                [1] => Array
                    (
                        [Person] => Array
                            (
                                [name] => Jeff
                            )
                    )
            )
        */
        $result = Set::sort($a, '{n}.Shirt', 'asc');
        /* $result now looks like:
            Array
            (
                [0] => Array
                    (
                        [Person] => Array
                            (
                                [name] => Jeff
                            )
                    )
                [1] => Array
                    (
                        [Shirt] => Array
                            (
                                [color] => black
                            )
                    )
            )
        */
```

```
45.       $result = Set::sort($a, '{n}', 'desc');
46.       /* $result now looks like:
47.           Array
48.           (
49.               [0] => Array
50.                   (
51.                       [Shirt] => Array
52.                           (
53.                               [color] => black
54.                           )
55.                   )
56.               [1] => Array
57.                   (
58.                       [Person] => Array
59.                           (
60.                               [name] => Jeff
61.                           )
62.                   )
63.           )
64.       */
65.       $a = array(
66.           array(7,6,4),
67.           array(3,4,5),
68.           array(3,2,1),
69.       );
70.       $result = Set::sort($a, '{n}.{n}', 'asc');
71.       /* $result now looks like:
72.           Array
73.           (
74.               [0] => Array
75.                   (
```

```
                    [0] => 3
                    [1] => 2
                    [2] => 1
                )
            [1] => Array
                (
                    [0] => 3
                    [1] => 4
                    [2] => 5
                )
            [2] => Array
                (
                    [0] => 7
                    [1] => 6
                    [2] => 4
                )
        )
    */
```

8.5.4 reverse

`array Set::reverse ($object)`

Set::reverse is basically the opposite of Set::map. It converts an object into an array. If $object is not an object, reverse will simply return $object.

```
    $result = Set::reverse(null);
    // Null
    $result = Set::reverse(false);
    // false
    $a = array(
```

```
7.            'Post' => array('id'=> 1, 'title' => 'First Post'),
8.            'Comment' => array(
9.                array('id'=> 1, 'title' => 'First Comment'),
10.               array('id'=> 2, 'title' => 'Second Comment')
11.           ),
12.           'Tag' => array(
13.               array('id'=> 1, 'title' => 'First Tag'),
14.               array('id'=> 2, 'title' => 'Second Tag')
15.           ),
16.       );
17.       $map = Set::map($a); // Turn $a into a class object
18.       /* $map now looks like:
19.           stdClass Object
20.           (
21.               [_name_] => Post
22.               [id] => 1
23.               [title] => First Post
24.               [Comment] => Array
25.                   (
26.                       [0] => stdClass Object
27.                           (
28.                               [id] => 1
29.                               [title] => First Comment
30.                           )
31.                       [1] => stdClass Object
32.                           (
33.                               [id] => 2
34.                               [title] => Second Comment
35.                           )
36.                   )
37.               [Tag] => Array
```

```
                (
                    [0] => stdClass Object
                        (
                            [id] => 1
                            [title] => First Tag
                        )
                    [1] => stdClass Object
                        (
                            [id] => 2
                            [title] => Second Tag
                        )
                )
        )
    */
    $result = Set::reverse($map);
    /* $result now looks like:
        Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First Post
                    [Comment] => Array
                        (
                            [0] => Array
                                (
                                    [id] => 1
                                    [title] => First Comment
                                )
                            [1] => Array
                                (
```

```
                                    [id] => 2
                                    [title] => Second Comment
                                )
                        )
                    [Tag] => Array
                        (
                            [0] => Array
                                (
                                    [id] => 1
                                    [title] => First Tag
                                )
                            [1] => Array
                                (
                                    [id] => 2
                                    [title] => Second Tag
                                )
                        )
                )
        )
    */
    $result = Set::reverse($a['Post']); // Just return the array
    /* $result now looks like:
        Array
        (
            [id] => 1
            [title] => First Post
        )
    */
```

8.5.5 combine

array Set::combine ($data, $path1 = null, $path2 = null, $groupPath = null)

Creates an associative array using a $path1 as the path to build its keys, and optionally $path2 as path to get the values. If $path2 is not specified, all values will be initialized to null (useful for Set::merge). You can optionally group the values by what is obtained when following the path specified in $groupPath.

```
1.      $result = Set::combine(array(), '{n}.User.id', '{n}.User.Data');
2.      // $result == array();
3.      $result = Set::combine('', '{n}.User.id', '{n}.User.Data');
4.      // $result == array();
5.      $a = array(
6.          array(
7.              'User' => array(
8.                  'id' => 2,
9.                  'group_id' => 1,
10.                 'Data' => array(
11.                     'user' => 'mariano.iglesias',
12.                     'name' => 'Mariano Iglesias'
13.                 )
14.             )
15.         ),
16.         array(
17.             'User' => array(
18.                 'id' => 14,
19.                 'group_id' => 2,
20.                 'Data' => array(
21.                     'user' => 'phpnut',
22.                     'name' => 'Larry E. Masters'
23.                 )
24.             )
25.         ),
```

```
        array(
            'User' => array(
                'id' => 25,
                'group_id' => 1,
                'Data' => array(
                    'user' => 'gwoo',
                    'name' => 'The Gwoo'
                )
            )
        )
    );
    $result = Set::combine($a, '{n}.User.id');
    /* $result now looks like:
        Array
        (
            [2] =>
            [14] =>
            [25] =>
        )
    */
    $result = Set::combine($a, '{n}.User.id', '{n}.User.non-existant');
    /* $result now looks like:
        Array
        (
            [2] =>
            [14] =>
            [25] =>
        )
    */
    $result = Set::combine($a, '{n}.User.id', '{n}.User.Data');
    /* $result now looks like:
```

```
57.           Array
58.           (
59.               [2] => Array
60.                   (
61.                       [user] => mariano.iglesias
62.                       [name] => Mariano Iglesias
63.                   )
64.               [14] => Array
65.                   (
66.                       [user] => phpnut
67.                       [name] => Larry E. Masters
68.                   )
69.               [25] => Array
70.                   (
71.                       [user] => gwoo
72.                       [name] => The Gwoo
73.                   )
74.           )
75.       */
76.       $result = Set::combine($a, '{n}.User.id', '{n}.User.Data.name');
77.       /* $result now looks like:
78.           Array
79.           (
80.               [2] => Mariano Iglesias
81.               [14] => Larry E. Masters
82.               [25] => The Gwoo
83.           )
84.       */
85.       $result = Set::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
86.       /* $result now looks like:
87.           Array
```

```
88.           (
89.               [1] => Array
90.                   (
91.                       [2] => Array
92.                           (
93.                               [user] => mariano.iglesias
94.                               [name] => Mariano Iglesias
95.                           )
96.                       [25] => Array
97.                           (
98.                               [user] => gwoo
99.                               [name] => The Gwoo
100.                          )
101.                  )
102.              [2] => Array
103.                  (
104.                      [14] => Array
105.                          (
106.                              [user] => phpnut
107.                              [name] => Larry E. Masters
108.                          )
109.                  )
110.          )
111.      */
112.      $result = Set::combine($a, '{n}.User.id', '{n}.User.Data.name', '{n}.User.group_id');
113.      /* $result now looks like:
114.          Array
115.          (
116.              [1] => Array
117.                  (
118.                      [2] => Mariano Iglesias
```

```
                    [25] => The Gwoo
                )
            [2] => Array
                (
                    [14] => Larry E. Masters
                )
        )
    */
    $result = Set::combine($a, '{n}.User.id', array('{0}: {1}', '{n}.User.Data.user', '{n}.User.Data.name'), '{n}.User.group_id');
    /* $result now looks like:
        Array
        (
            [1] => Array
                (
                    [2] => mariano.iglesias: Mariano Iglesias
                    [25] => gwoo: The Gwoo
                )
            [2] => Array
                (
                    [14] => phpnut: Larry E. Masters
                )
        )
    */
    $result = Set::combine($a, array('{0}: {1}', '{n}.User.Data.user', '{n}.User.Data.name'), '{n}.User.id');
    /* $result now looks like:
        Array
        (
            [mariano.iglesias: Mariano Iglesias] => 2
            [phpnut: Larry E. Masters] => 14
            [gwoo: The Gwoo] => 25
```

```
149.         )
150.     */
151.     $result = Set::combine($a, array('{1}: {0}', '{n}.User.Data.user', '{n}.User.Data.name'), '{n}.User.id');
152.     /* $result now looks like:
153.         Array
154.         (
155.             [Mariano Iglesias: mariano.iglesias] => 2
156.             [Larry E. Masters: phpnut] => 14
157.             [The Gwoo: gwoo] => 25
158.         )
159.     */
160.     $result = Set::combine($a, array('%1$s: %2$d', '{n}.User.Data.user', '{n}.User.id'), '{n}.User.Data.name');
161.     /* $result now looks like:
162.         Array
163.         (
164.             [mariano.iglesias: 2] => Mariano Iglesias
165.             [phpnut: 14] => Larry E. Masters
166.             [gwoo: 25] => The Gwoo
167.         )
168.     */
169.     $result = Set::combine($a, array('%2$d: %1$s', '{n}.User.Data.user', '{n}.User.id'), '{n}.User.Data.name');
170.     /* $result now looks like:
171.         Array
172.         (
173.             [2: mariano.iglesias] => Mariano Iglesias
174.             [14: phpnut] => Larry E. Masters
175.             [25: gwoo] => The Gwoo
176.         )
177.     */
```

8.5.6 normalize

```
array Set::normalize ($list, $assoc = true, $sep = ',', $trim = true)
```

Normalizes a string or array list.

```
7.      $a = array('Tree', 'CounterCache',
8.              'Upload' => array(
9.                  'folder' => 'products',
10.                 'fields' => array('image_1_id', 'image_2_id', 'image_3_id', 'image_4_id', 'image_5_id')));
11.     $b =  array('Cacheable' => array('enabled' => false),
12.             'Limit',
13.             'Bindable',
14.             'Validator',
15.             'Transactional');
16.     $result = Set::normalize($a);
17.     /* $result now looks like:
18.         Array
19.         (
20.             [Tree] =>
21.             [CounterCache] =>
22.             [Upload] => Array
23.                 (
24.                     [folder] => products
25.                     [fields] => Array
26.                         (
27.                             [0] => image_1_id
28.                             [1] => image_2_id
29.                             [2] => image_3_id
30.                             [3] => image_4_id
31.                             [4] => image_5_id
32.                         )
```

```
                )
            )
        */
        $result = Set::normalize($b);
        /* $result now looks like:
            Array
            (
                [Cacheable] => Array
                    (
                        [enabled] =>
                    )
                [Limit] =>
                [Bindable] =>
                [Validator] =>
                [Transactional] =>
            )
        */
        $result = Set::merge($a, $b); // Now merge the two and normalize
        /* $result now looks like:
            Array
            (
                [0] => Tree
                [1] => CounterCache
                [Upload] => Array
                    (
                        [folder] => products
                        [fields] => Array
                            (
                                [0] => image_1_id
                                [1] => image_2_id
                                [2] => image_3_id
```

```
                                [3] => image_4_id
                                [4] => image_5_id
                            )
                    )
                [Cacheable] => Array
                    (
                        [enabled] =>
                    )
                [2] => Limit
                [3] => Bindable
                [4] => Validator
                [5] => Transactional
            )
        */
        $result = Set::normalize(Set::merge($a, $b));
        /* $result now looks like:
            Array
            (
                [Tree] =>
                [CounterCache] =>
                [Upload] => Array
                    (
                        [folder] => products
                        [fields] => Array
                            (
                                [0] => image_1_id
                                [1] => image_2_id
                                [2] => image_3_id
                                [3] => image_4_id
                                [4] => image_5_id
                            )
```

```
95.                     )
96.                 [Cacheable] => Array
97.                     (
98.                         [enabled] =>
99.                     )
100.                [Limit] =>
101.                [Bindable] =>
102.                [Validator] =>
103.                [Transactional] =>
104.            )
105.    */
```

8.5.7 countDim

integer Set::countDim ($array = null, $all = false, $count = 0)

Counts the dimensions of an array. If $all is set to false (which is the default) it will only consider the dimension of the first element in the array.

```php
1.      $data = array('one', '2', 'three');
2.      $result = Set::countDim($data);
3.      // $result == 1
4.      $data = array('1' => '1.1', '2', '3');
5.      $result = Set::countDim($data);
6.      // $result == 1
7.      $data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => '3.1.1'));
8.      $result = Set::countDim($data);
9.      // $result == 2
10.     $data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
11.     $result = Set::countDim($data);
12.     // $result == 1
```

```
13.       $data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
14.       $result = Set::countDim($data, true);
15.       // $result == 2
16.       $data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1' => '3.1.1.1')));
17.       $result = Set::countDim($data);
18.       // $result == 2
19.       $data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1' => '3.1.1.1')));
20.       $result = Set::countDim($data, true);
21.       // $result == 3
22.       $data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1' => '2.1.1.1'))), '3'
    => array('3.1' => array('3.1.1' => '3.1.1.1')));
23.       $result = Set::countDim($data, true);
24.       // $result == 4
25.       $data   =   array('1'   =>   array('1.1'   =>   '1.1.1'),   array('2'   =>   array('2.1'   =>   array('2.1.1'   =>
    array('2.1.1.1')))), '3' => array('3.1' => array('3.1.1' => '3.1.1.1')));
26.       $result = Set::countDim($data, true);
27.       // $result == 5
28.       $data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1' => array('2.1.1.1'
    => '2.1.1.1.1')))), '3' => array('3.1' => array('3.1.1' => '3.1.1.1')));
29.       $result = Set::countDim($data, true);
30.       // $result == 5
31.       $set = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1' => array('2.1.1.1' =>
    '2.1.1.1.1')))), '3' => array('3.1' => array('3.1.1' => '3.1.1.1')));
32.       $result = Set::countDim($set, false, 0);
33.       // $result == 2
34.       $result = Set::countDim($set, true);
35.       // $result == 5
36.
```

8.5.8 diff

array Set::diff ($val1, $val2 = null)

Computes the difference between a Set and an array, two Sets, or two arrays

```
$a = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about')
);
$b = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about'),
    2 => array('name' => 'contact')
);
$result = Set::diff($a, $b);
/* $result now looks like:
    Array
    (
        [2] => Array
            (
                [name] => contact
            )
    )
*/
$result = Set::diff($a, array());
/* $result now looks like:
    Array
    (
        [0] => Array
            (
                [name] => main
            )
```

```
            [1] => Array
                (
                    [name] => about
                )
        )
    */
    $result = Set::diff(array(), $b);
    /* $result now looks like:
        Array
        (
            [0] => Array
                (
                    [name] => main
                )
            [1] => Array
                (
                    [name] => about
                )
            [2] => Array
                (
                    [name] => contact
                )
        )
    */
    $b = array(
        0 => array('name' => 'me'),
        1 => array('name' => 'about')
    );
    $result = Set::diff($a, $b);
    /* $result now looks like:
        Array
```

```
59.             (
60.                 [0] => Array
61.                     (
62.                         [name] => main
63.                     )
64.             )
65.         */
```

8.5.9 check

boolean/array Set::check ($data, $path = null)

Checks if a particular path is set in an array. If $path is empty, $data will be returned instead of a boolean value.

```
4.          $set = array(
5.              'My Index 1' => array('First' => 'The first item')
6.          );
7.          $result = Set::check($set, 'My Index 1.First');
8.          // $result == True
9.          $result = Set::check($set, 'My Index 1');
10.         // $result == True
11.         $result = Set::check($set, array());
12.         // $result == array('My Index 1' => array('First' => 'The first item'))
13.         $set = array(
14.             'My Index 1' => array('First' =>
15.                 array('Second' =>
16.                     array('Third' =>
17.                         array('Fourth' => 'Heavy. Nesting.'))))
18.         );
19.         $result = Set::check($set, 'My Index 1.First.Second');
```

```
20.       // $result == True
21.       $result = Set::check($set, 'My Index 1.First.Second.Third');
22.       // $result == True
23.       $result = Set::check($set, 'My Index 1.First.Second.Third.Fourth');
24.       // $result == True
25.       $result = Set::check($set, 'My Index 1.First.Seconds.Third.Fourth');
26.       // $result == False
```

8.5.10 remove

array Set::remove ($list, $path = null)

Removes an element from a Set or array as defined by $path.

```
1.        $a = array(
2.            'pages'      => array('name' => 'page'),
3.            'files'         => array('name' => 'files')
4.        );
5.        $result = Set::remove($a, 'files');
6.        /* $result now looks like:
7.            Array
8.            (
9.                [pages] => Array
10.                   (
11.                       [name] => page
12.                   )
13.           )
14.       */
```

8.5.11 classicExtract

```
array Set::classicExtract ($data, $path = null)
```

Gets a value from an array or object that is contained in a given path using an array path syntax, i.e.:

- "{n}.Person.{[a-z]+}" - Where "{n}" represents a numeric key, "Person" represents a string literal
- "{[a-z]+}" (i.e. any string literal enclosed in brackets besides {n} and {s}) is interpreted as a regular expression.

Example 1

```
1.      $a = array(
2.          array('Article' => array('id' => 1, 'title' => 'Article 1')),
3.          array('Article' => array('id' => 2, 'title' => 'Article 2')),
4.          array('Article' => array('id' => 3, 'title' => 'Article 3')));
5.      $result = Set::classicExtract($a, '{n}.Article.id');
6.      /* $result now looks like:
7.          Array
8.          (
9.              [0] => 1
10.             [1] => 2
11.             [2] => 3
12.         )
13.     */
14.     $result = Set::classicExtract($a, '{n}.Article.title');
15.     /* $result now looks like:
16.         Array
17.         (
18.             [0] => Article 1
19.             [1] => Article 2
20.             [2] => Article 3
21.         )
22.     */
```

```
$result = Set::classicExtract($a, '1.Article.title');
// $result == "Article 2"
$result = Set::classicExtract($a, '3.Article.title');
// $result == null
```

Example 2

```
$a = array(
    0 => array('pages' => array('name' => 'page')),
    1 => array('fruites'=> array('name' => 'fruit')),
    'test' => array(array('name' => 'jippi')),
    'dot.test' => array(array('name' => 'jippi'))
);
$result = Set::classicExtract($a, '{n}.{s}.name');
/* $result now looks like:
Array
    (
        [0] => Array
            (
                [0] => page
            )
        [1] => Array
            (
                [0] => fruit
            )
    )
*/
$result = Set::classicExtract($a, '{s}.{n}.name');
/* $result now looks like:
    Array
    (
        [0] => Array
            (
```

```
                    [0] => jippi
                )
            [1] => Array
                (
                    [0] => jippi
                )
        )
    */
    $result = Set::classicExtract($a,'{\w+}.{\w+}.name');
    /* $result now looks like:
        Array
        (
            [0] => Array
                (
                    [pages] => page
                )
            [1] => Array
                (
                    [fruites] => fruit
                )
            [test] => Array
                (
                    [0] => jippi
                )
            [dot.test] => Array
                (
                    [0] => jippi
                )
        )
    */
    $result = Set::classicExtract($a,'{\d+}.{\w+}.name');
```

```
58.       /* $result now looks like:
59.           Array
60.           (
61.               [0] => Array
62.                   (
63.                       [pages] => page
64.                   )
65.               [1] => Array
66.                   (
67.                       [fruites] => fruit
68.                   )
69.           )
70.       */
71.       $result = Set::classicExtract($a,'{n}.{\w+}.name');
72.       /* $result now looks like:
73.           Array
74.           (
75.               [0] => Array
76.                   (
77.                       [pages] => page
78.                   )
79.               [1] => Array
80.                   (
81.                       [fruites] => fruit
82.                   )
83.           )
84.       */
85.       $result = Set::classicExtract($a,'{s}.{\d+}.name');
86.       /* $result now looks like:
87.           Array
88.           (
```

```
89.                 [0] => Array
90.                     (
91.                         [0] => jippi
92.                     )
93.                 [1] => Array
94.                     (
95.                         [0] => jippi
96.                     )
97.             )
98.         */
99.         $result = Set::classicExtract($a,'{s}');
100.        /* $result now looks like:
101.            Array
102.            (
103.                [0] => Array
104.                    (
105.                        [0] => Array
106.                            (
107.                                [name] => jippi
108.                            )
109.                    )
110.                [1] => Array
111.                    (
112.                        [0] => Array
113.                            (
114.                                [name] => jippi
115.                            )
116.                    )
117.            )
118.        */
119.        $result = Set::classicExtract($a,'{[a-z]}');
```

```
120.      /* $result now looks like:
121.          Array
122.          (
123.              [test] => Array
124.                  (
125.                      [0] => Array
126.                          (
127.                              [name] => jippi
128.                          )
129.                  )
130.              [dot.test] => Array
131.                  (
132.                      [0] => Array
133.                          (
134.                              [name] => jippi
135.                          )
136.                  )
137.          )
138.      */
139.      $result = Set::classicExtract($a, '{dot\.test}.{n}');
140.      /* $result now looks like:
141.          Array
142.          (
143.              [dot.test] => Array
144.                  (
145.                      [0] => Array
146.                          (
147.                              [name] => jippi
148.                          )
149.                  )
150.          )
```

```
151.        */
```

8.5.12 matches

boolean Set::matches ($conditions, $data=array(), $i = null, $length=null)

Set::matches can be used to see if a single item or a given xpath match certain conditions.

```
4.          $a = array(
5.              array('Article' => array('id' => 1, 'title' => 'Article 1')),
6.              array('Article' => array('id' => 2, 'title' => 'Article 2')),
7.              array('Article' => array('id' => 3, 'title' => 'Article 3')));
8.          $res=Set::matches(array('id>2'), $a[1]['Article']);
9.          // returns false
10.         $res=Set::matches(array('id>=2'), $a[1]['Article']);
11.         // returns true
12.         $res=Set::matches(array('id>=3'), $a[1]['Article']);
13.         // returns false
14.         $res=Set::matches(array('id<=2'), $a[1]['Article']);
15.         // returns true
16.         $res=Set::matches(array('id<2'), $a[1]['Article']);
17.         // returns false
18.         $res=Set::matches(array('id>1'), $a[1]['Article']);
19.         // returns true
20.         $res=Set::matches(array('id>1', 'id<3', 'id!=0'), $a[1]['Article']);
21.         // returns true
22.         $res=Set::matches(array('3'), null, 3);
23.         // returns true
24.         $res=Set::matches(array('5'), null, 5);
25.         // returns true
```

```
26.         $res=Set::matches(array('id'), $a[1]['Article']);
27.         // returns true
28.         $res=Set::matches(array('id', 'title'), $a[1]['Article']);
29.         // returns true
30.         $res=Set::matches(array('non-existant'), $a[1]['Article']);
31.         // returns false
32.         $res=Set::matches('/Article[id=2]', $a);
33.         // returns true
34.         $res=Set::matches('/Article[id=4]', $a);
35.         // returns false
36.         $res=Set::matches(array(), $a);
37.         // returns true
```

8.5.13 extract

array Set::extract ($path, $data=null, $options=array())

Set::extract uses basic XPath 2.0 syntax to return subsets of your data from a find or a find all. This function allows you to retrieve your data quickly without having to loop through multi dimentional arrays or traverse through tree structures.

If $path is an array or $data is empty it the call is delegated to Set::classicExtract.

```
5.         // Common Usage:
6.         $users = $this->User->find("all");
7.         $results = Set::extract('/User/id', $users);
8.         // results returns:
9.         // array(1,2,3,4,5,...);
```

Currently implemented selectors:

Selector	Note
/User/id	Similar to the classic {n}.User.id
/User[2]/name	Selects the name of the second User
/User[id<2]	Selects all Users with an id < 2
/User[id>2][<5]	Selects all Users with an id > 2 but < 5
/Post/Comment[author_name=john]/../name	Selects the name of all Posts that have at least one Comment written by john
/Posts[title]	Selects all Posts that have a 'title' key
/Comment/.[1]	Selects the contents of the first comment
/Comment/.[:last]	Selects the last comment
/Comment/.[:first]	Selects the first comment
/Comment[text=/cakephp/i]	Selects all comments that have a text matching the regex /cakephp/i
/Comment/@*	Selects the key names of all comments

Currently only absolute paths starting with a single '/' are supported. Please report any bugs as you find them. Suggestions for additional features are welcome.

To learn more about Set::extract() refer to function testExtract() in /cake/tests/cases/libs/set.test.php.

8.5.14 format

```
array Set::format ($data, $format, $keys)
```

Returns a series of values extracted from an array, formatted in a format string.

```
4.        $data = array(
5.            array('Person' => array('first_name' => 'Nate', 'last_name' => 'Abele', 'city' => 'Boston', 'state' =>
   'MA', 'something' => '42')),
6.            array('Person' => array('first_name' => 'Larry', 'last_name' => 'Masters', 'city' => 'Boondock', 'state'
   => 'TN', 'something' => '{0}')),
7.            array('Person' => array('first_name' => 'Garrett', 'last_name' => 'Woodworth', 'city' => 'Venice Beach',
   'state' => 'CA', 'something' => '{1}')));
8.        $res = Set::format($data, '{1}, {0}', array('{n}.Person.first_name', '{n}.Person.last_name'));
9.        /*
10.       Array
11.       (
12.           [0] => Abele, Nate
13.           [1] => Masters, Larry
14.           [2] => Woodworth, Garrett
15.       )
16.       */
17.       $res = Set::format($data, '{0}, {1}', array('{n}.Person.city', '{n}.Person.state'));
18.       /*
19.       Array
20.       (
21.           [0] => Boston, MA
22.           [1] => Boondock, TN
23.           [2] => Venice Beach, CA
24.       )
25.       */
26.       $res = Set::format($data, '{{0}, {1}}', array('{n}.Person.city', '{n}.Person.state'));
27.       /*
28.       Array
29.       (
```

```
        [0] => {Boston, MA}
        [1] => {Boondock, TN}
        [2] => {Venice Beach, CA}
    )
    */
    $res = Set::format($data, '{%2$d, %1$s}', array('{n}.Person.something', '{n}.Person.something'));
    /*
    Array
    (
        [0] => {42, 42}
        [1] => {0, {0}}
        [2] => {0, {1}}
    )
    */
    $res = Set::format($data, '%2$d, %1$s', array('{n}.Person.first_name', '{n}.Person.something'));
    /*
    Array
    (
        [0] => 42, Nate
        [1] => 0, Larry
        [2] => 0, Garrett
    )
    */
    $res = Set::format($data, '%1$s, %2$d', array('{n}.Person.first_name', '{n}.Person.something'));
    /*
    Array
    (
        [0] => Nate, 42
        [1] => Larry, 0
        [2] => Garrett, 0
    )
```

```
61.         */
```

8.5.15 enum

string Set::enum ($select, $list=null)

The enum method works well when using html select elements. It returns a value from an array list if the key exists.

If a comma separated $list is passed arrays are numeric with the key of the first being 0 $list = 'no, yes' would translate to $list = array(0 => 'no', 1 => 'yes');

If an array is used, keys can be strings example: array('no' => 0, 'yes' => 1);

$list defaults to 0 = no 1 = yes if param is not passed

```
6.          $res = Set::enum(1, 'one, two');
7.          // $res is 'two'
8.          $res = Set::enum('no', array('no' => 0, 'yes' => 1));
9.          // $res is 0
10.         $res = Set::enum('first', array('first' => 'one', 'second' => 'two'));
11.         // $res is 'one'
```

8.5.16 numeric

boolean Set::numeric ($array=null)

Checks to see if all the values in the array are numeric

```
4.              $data = array('one');
5.              $res = Set::numeric(array_keys($data));
```

```
6.
7.          // $res is true
8.
9.          $data = array(1 => 'one');
10.         $res = Set::numeric($data);
11.         // $res is false
12.
13.         $data = array('one');
14.         $res = Set::numeric($data);
15.
16.         // $res is false
17.
18.         $data = array('one' => 'two');
19.         $res = Set::numeric($data);
20.
21.         // $res is false
22.
23.         $data = array('one' => 1);
24.         $res = Set::numeric($data);
25.
26.         // $res is true
27.
28.         $data = array(0);
29.         $res = Set::numeric($data);
30.
31.         // $res is true
32.
33.         $data = array('one', 'two', 'three', 'four', 'five');
34.         $res = Set::numeric(array_keys($data));
35.
```

```
        // $res is true

        $data = array(1 => 'one', 2 => 'two', 3 => 'three', 4 => 'four', 5 => 'five');
        $res = Set::numeric(array_keys($data));

        // $res is true

        $data = array('1' => 'one', 2 => 'two', 3 => 'three', 4 => 'four', 5 => 'five');
        $res = Set::numeric(array_keys($data));

        // $res is true

        $data = array('one', 2 => 'two', 3 => 'three', 4 => 'four', 'a' => 'five');
        $res = Set::numeric(array_keys($data));

        // $res is false
```

8.5.17 map

object Set::map ($class = 'stdClass', $tmp = 'stdClass')

This method Maps the contents of the Set object to an object hierarchy while maintaining numeric keys as arrays of objects.

Basically, the map function turns array items into initialized class objects. By default it turns an array into a stdClass Object, however you can map values into any type of class. Example: Set::map($array_of_values, 'nameOfYourClass');

```
    $data = array(
        array(
            "IndexedPage" => array(
                "id" => 1,
```

```
8.                      "url" => 'http://blah.com/',
9.                      'hash' => '68a9f053b19526d08e36c6a9ad150737933816a5',
10.                     'get_vars' => '',
11.                     'redirect' => '',
12.                     'created' => "1195055503",
13.                     'updated' => "1195055503",
14.                 )
15.             ),
16.             array(
17.                 "IndexedPage" => array(
18.                     "id" => 2,
19.                     "url" => 'http://blah.com/',
20.                     'hash' => '68a9f053b19526d08e36c6a9ad150737933816a5',
21.                     'get_vars' => '',
22.                     'redirect' => '',
23.                     'created' => "1195055503",
24.                     'updated' => "1195055503",
25.                 ),
26.             )
27.         );
28.         $mapped = Set::map($data);
29.         /* $mapped now looks like:
30.             Array
31.             (
32.                 [0] => stdClass Object
33.                     (
34.                         [_name_] => IndexedPage
35.                         [id] => 1
36.                         [url] => http://blah.com/
37.                         [hash] => 68a9f053b19526d08e36c6a9ad150737933816a5
38.                         [get_vars] =>
```

```
                    [redirect] =>
                    [created] => 1195055503
                    [updated] => 1195055503
                )
            [1] => stdClass Object
                (
                    [_name_] => IndexedPage
                    [id] => 2
                    [url] => http://blah.com/
                    [hash] => 68a9f053b19526d08e36c6a9ad150737933816a5
                    [get_vars] =>
                    [redirect] =>
                    [created] => 1195055503
                    [updated] => 1195055503
                )
        )
    */
```

Using Set::map() with a custom class for second parameter:

```
class MyClass {
    function sayHi() {
        echo 'Hi!';
    }
}
$mapped = Set::map($data, 'MyClass');
//Now you can access all the properties as in the example above,
//but also you can call MyClass's methods
$mapped->[0]->sayHi();
```

8.5.18 pushDiff

```
array Set::pushDiff ($array1, $array2)
```

This function merges two arrays and pushes the differences in array2 to the bottom of the resultant array.

Example 1

```
1.      $array1 = array('ModelOne' => array('id'=>1001, 'field_one'=>'a1.m1.f1', 'field_two'=>'a1.m1.f2'));
2.      $array2   =   array('ModelOne'   =>   array('id'=>1003,   'field_one'=>'a3.m1.f1',   'field_two'=>'a3.m1.f2',
'field_three'=>'a3.m1.f3'));
3.      $res = Set::pushDiff($array1, $array2);
4.      /* $res now looks like:
5.          Array
6.          (
7.              [ModelOne] => Array
8.                  (
9.                      [id] => 1001
10.                     [field_one] => a1.m1.f1
11.                     [field_two] => a1.m1.f2
12.                     [field_three] => a3.m1.f3
13.                 )
14.         )
15.     */
```

Example 2

```
1.      $array1 = array("a"=>"b", 1 => 20938, "c"=>"string");
2.      $array2 = array("b"=>"b", 3 => 238, "c"=>"string", array("extra_field"));
3.      $res = Set::pushDiff($array1, $array2);
4.      /* $res now looks like:
5.          Array
6.          (
7.              [a] => b
8.              [1] => 20938
```

```
                [c] => string
                [b] => b
                [3] => 238
                [4] => Array
                    (
                        [0] => extra_field
                    )
            )
        */
```

8.5.19 filter

array Set::filter ($var, $isArray=null)

Filters empty elements out of a route array, excluding '0'.

```
        $res = Set::filter(array('0', false, true, 0, array('one thing', 'I can tell you', 'is you got to be', false)));
        /* $res now looks like:
            Array (
                [0] => 0
                [2] => 1
                [3] => 0
                [4] => Array
                    (
                        [0] => one thing
                        [1] => I can tell you
                        [2] => is you got to be
                        [3] =>
                    )
```

```
14.             )
15.         */
```

8.5.20 merge

array Set::merge ($arr1, $arr2=null)

This function can be thought of as a hybrid between PHP's array_merge and array_merge_recursive. The difference to the two is that if an array key contains another array then the function behaves recursive (unlike array_merge) but does not do if for keys containing strings (unlike array_merge_recursive). See the unit test for more information.

This function will work with an unlimited amount of arguments and typecasts non-array parameters into arrays.

```
1.          $arry1 = array(
2.              array(
3.                  'id' => '48c2570e-dfa8-4c32-a35e-0d71cbdd56cb',
4.                  'name' => 'mysql raleigh-workshop-08 < 2008-09-05.sql ',
5.                  'description' => 'Importing an sql dump'
6.              ),
7.              array(
8.                  'id' => '48c257a8-cf7c-4af2-ac2f-114ecbdd56cb',
9.                  'name' => 'pbpaste | grep -i Unpaid | pbcopy',
10.                 'description' => 'Remove all lines that say "Unpaid".',
11.             )
12.         );
13.         $arry2 = 4;
14.         $arry3 = array(0=>"test array", "cats"=>"dogs", "people" => 1267);
15.         $arry4 = array("cats"=>"felines", "dog"=>"angry");
16.         $res = Set::merge($arry1, $arry2, $arry3, $arry4);
17.         /* $res now looks like:
```

```
Array
(
    [0] => Array
        (
            [id] => 48c2570e-dfa8-4c32-a35e-0d71cbdd56cb
            [name] => mysql raleigh-workshop-08 < 2008-09-05.sql
            [description] => Importing an sql dump
        )
    [1] => Array
        (
            [id] => 48c257a8-cf7c-4af2-ac2f-114ecbdd56cb
            [name] => pbpaste | grep -i Unpaid | pbcopy
            [description] => Remove all lines that say "Unpaid".
        )
    [2] => 4
    [3] => test array
    [cats] => felines
    [people] => 1267
    [dog] => angry
)
*/
```

8.5.21 contains

boolean Set::contains ($val1, $val2 = null)

Determines if one Set or array contains the exact keys and values of another.

```
$a = array(
    0 => array('name' => 'main'),
```

```
            1 => array('name' => 'about')
        );
        $b = array(
            0 => array('name' => 'main'),
            1 => array('name' => 'about'),
            2 => array('name' => 'contact'),
            'a' => 'b'
        );
        $result = Set::contains($a, $a);
        // True
        $result = Set::contains($a, $b);
        // False
        $result = Set::contains($b, $a);
        // True
```

8.6 Security

The security library handles basic security measures such as providing methods for hashing and encrypting data.

8.7 Cache

The Cache class in CakePHP provides a generic frontend for several backend caching systems. Different Cache configurations and engines can be setup in your app/config/core.php

8.7.1 Cache::read()

Cache::read($key, $config = null)

Cache::read() is used to read the cached value stored under `$key` from the `$config`. If $config is null the default config will be used. `Cache::read()` will return the cached value if it is a valid cache or `false` if the cache has expired or doesn't exist. The contents of the cache might evaluate false, so make sure you use the strict comparison operator `===` or `!==`.

For example:

```
1.        $cloud = Cache::read('cloud');
2.        if ($cloud !== false) {
3.            return $cloud;
4.        }
5.        // generate cloud data
6.        // ...
7.        // store data in cache
8.        Cache::write('cloud', $cloud);
9.        return $cloud;
```

8.7.2 Cache::write()

Cache::write($key, $value, $config = null);

Cache::write() will write a $value to the Cache. You can read or delete this value later by refering to it by `$key`. You may specify an optional configuration to store the cache in as well. If no `$config` is specified default will be used. Cache::write() can store any type of object and is ideal for storing results of model finds.

```
1.            if (($posts = Cache::read('posts')) === false) {
2.                $posts = $this->Post->find('all');
3.                Cache::write('posts', $posts);
4.            }
```

Using Cache::write() and Cache::read() to easily reduce the number of trips made to the database to fetch posts.

8.7.3 Cache::delete()

Cache::delete($key, $config = null)

`Cache::delete()` will allow you to completely remove a cached object from the Cache store.

8.7.4 Cache::config()

`Cache::config()` is used to create additional Cache configurations. These additional configurations can have different duration, engines, paths, or prefixes than your default cache config. Using multiple cache configurations can help reduce the number of times you need to use `Cache::set()` as well as centralize all your cache settings.

> You must specify which engine to use. It does **not** default to File.

```
1.      Cache::config('short', array(
2.          'engine' => 'File',
3.          'duration'=> '+1 hours',
4.          'path' => CACHE,
5.          'prefix' => 'cake_short_'
6.      ));
7.      // long
8.      Cache::config('long', array(
9.          'engine' => 'File',
10.         'duration'=> '+1 week',
11.         'probability'=> 100,
12.         'path' => CACHE . 'long' . DS,
13.     ));
```

By placing the above code in your `app/config/core.php` you will have two additional Cache configurations. The name of these configurations 'short' or 'long' is used as the `$config` parameter for `Cache::write()` and `Cache::read()`.

You can provide custom `Cache` adapters in `app/libs` as well as in plugins using `$plugin/libs`. App/plugin cache engines can also override the core engines. Cache adapters must be in a cache directory. If you had a cache engine named `MyCustomCacheEngine` it would be placed in either

`app/libs/cache/my_custom_cache.php` as an app/libs. Or in `$plugin/libs/cache/my_custom_cache.php` as part of a plugin. Cache configs from plugins need to use the plugin dot syntax.

```
1.      Cache::config('custom', array(
2.          'engine' => 'CachePack.MyCustomCache',
3.          ...
4.      ));
```

App and Plugin cache engines should be configured in `app/bootstrap.php`. If you try to configure them in core.php they will not work correctly.

8.7.5 Cache::set()

`Cache::set()` allows you to temporarily override a cache configs settings for one operation (usually a read or write). If you use `Cache::set()` to change the settings for a write, you should also use `Cache::set()` before reading the data back in. If you fail to do so, the default settings will be used when the cache key is read.

```
1.      Cache::set(array('duration' => '+30 days'));
2.      Cache::write('results', $data);
3.      // Later on
4.      Cache::set(array('duration' => '+30 days'));
5.      $results = Cache::read('results');
```

If you find yourself repeatedly calling `Cache::set()` perhaps you should create a new Cache configuration. This will remove the need to call `Cache::set()`.

8.8 HttpSocket

CakePHP includes an HttpSocket class which can be used easily for making requests, such as those to web services.

8.8.1 get

The get method makes a simple HTTP GET request returning the results.

```
string get($uri, $query, $request)
```

`$uri` is the web address where the request is being made; `$query` is any query string parameters, either in string form: "param1=foo&param2=bar" or as a keyed array: array('param1' => 'foo', 'param2' => 'bar').

```
1.      App::import('Core', 'HttpSocket');
2.      $HttpSocket = new HttpSocket();
3.      $results = $HttpSocket->get('http://www.google.com/search', 'q=cakephp');
4.      //returns html for Google's search results for the query "cakephp"
```

8.8.2 post

The post method makes a simple HTTP POST request returning the results.

```
string function post ($uri, $data, $request)
```

The parameters for the `post` method are almost the same as the get method, `$uri` is the web address where the request is being made; `$query` is the data to be posted, either in string form: "param1=foo&param2=bar" or as a keyed array: array('param1' => 'foo', 'param2' => 'bar').

```
1.      App::import('Core', 'HttpSocket');
2.      $HttpSocket = new HttpSocket();
3.      $results = $HttpSocket->post('www.somesite.com/add', array('name' => 'test', 'type' => 'user'));
4.      //$results contains what is returned from the post.
```

8.8.3 request

The base request method, which is called from all the wrappers (get, post, put, delete). Returns the results of the request.

string function request($request)

$request is a keyed array of various options. Here is the format and default settings:

```
1.      var $request = array(
2.          'method' => 'GET',
3.          'uri' => array(
4.              'scheme' => 'http',
5.              'host' => null,
6.              'port' => 80,
7.              'user' => null,
8.              'pass' => null,
9.              'path' => null,
10.             'query' => null,
11.             'fragment' => null
12.         ),
13.         'auth' => array(
14.             'method' => 'Basic',
15.             'user' => null,
16.             'pass' => null
17.         ),
18.         'version' => '1.1',
19.         'body' => '',
20.         'line' => null,
21.         'header' => array(
22.             'Connection' => 'close',
23.             'User-Agent' => 'CakePHP'
24.         ),
25.         'raw' => null,
26.         'cookies' => array()
27.     );
```

8.9 Router

Router can be used to parse urls into arrays containing indexes for the controller, action, and any parameters, and the opposite: to convert url arrays (eg. array('controller'=>'posts', 'action'=>'index')) to string urls.

Read more about ways to configure the Router here: http://book.cakephp.org/view/945/Routes-Configuration

Router also include other utility methods for dealing with urls.

9 Core Console Applications

CakePHP features a number of console applications out of the box. Some of these applications are used in concert with other CakePHP features (like ACL or i18n), and others are for general use in getting you to launch quicker.

This section explains how to use the core console applications packaged with CakePHP.

Before you dive in here, you may want to check out the CakePHP Console section covered earlier. Console setup isn't covered here, so if you've never used the console before, check it out.

9.1 Code Generation with Bake

You've already learned about scaffolding in CakePHP: a simple way to get up and running with only a database and some bare classes. CakePHP's Bake console is another effort to get you up and running in CakePHP – fast. The Bake console can create any of CakePHP's basic ingredients: models, views and controllers. And we aren't just talking skeleton classes: Bake can create a fully functional application in just a few minutes. In fact, Bake is a natural step to take once an application has been scaffolded.

Those new to Bake (especially Windows users) may find the Bake screencast helpful in setting things up before continuing.

Depending on the configuration of your setup, you may have to set execute rights on the cake bash script or call it using `./cake bake`. The cake console is run using the PHP CLI (command line interface). If you have problems running the script, ensure that you have the PHP CLI installed and that it has the proper modules enabled (eg: MySQL).

When running Bake for the first time, you'll be prompted to create a Database Configuration file, if you haven't created one already.

After you've created a Database Configuration file, running Bake will present you with the following options:

```
---------------------------------------------------------------
App : app
Path: /path-to/project/app
---------------------------------------------------------------
Interactive Bake Shell
---------------------------------------------------------------
[D]atabase Configuration
[M]odel
[V]iew
[C]ontroller
[P]roject
[F]ixture
[T]est case
[Q]uit
What would you like to Bake? (D/M/V/C/P/F/T/Q)
>
```

Alternatively, you can run any of these commands directly from the command line:

```
$ cake bake db_config
$ cake bake model
$ cake bake view
$ cake bake controller
$ cake bake project
$ cake bake fixture
$ cake bake test
$ cake bake plugin plugin_name
$ cake bake all
```

9.1.1 Bake improvements in 1.3

For 1.3 bake has had a significant overhaul, and a number of features and enhancements have been built in.

- Two new tasks (FixtureTask and TestTask) are accessible from the main bake menu
- A third task (TemplateTask) has been added for use in your shells.
- All the different bake tasks now allow you to use connections other than default for baking. Using the `-connection` parameter.
- Plugin support has been greatly improved. You can use either `-plugin PluginName` or `Plugin.class`.
- Questions have been clarified, and made easier to understand.
- Multiple validations on models has been added.
- Self Associated models using `parent_id` are now detected. For example if your model is named Thread, a ParentThread and ChildThread association will be created.
- Fixtures and Tests can be baked separately.
- Baked Tests include as many fixtures as they know about, including plugin detection (plugin detection does not work on PHP4).

So with the laundry list of features, we'll take some time to look at some of the new commands, new parameters and updated features.

New FixtureTask, TestTask and TemplateTask.

Fixture and test baking were a bit of a pain in the past. You could only generate tests when baking the classes, and fixtures could only be generated when baking models. This made adding tests to your applications later or even regenerating fixtures with new schemas a bit painful. For 1.3 we've separated out Fixture and Test making them separate tasks. This allows you to re-run them and regenerate tests and fixtures at any point in your development process.

In addition to being rebuildable at any time, baked tests are now attempt to find as many fixtures as possible. In the past getting into testing often involved fighting through numerous 'Missing Table' errors. With more advanced fixture detection we hope to make testing easier and more accessible.

Test cases also generate skeleton test methods for every non-inherited public method in your classes. Saving you one extra step.

`TemplateTask` is a behind the scenes task, and it handles file generation from templates. In previous versions of CakePHP baked views were template based, but all other code was not. With 1.3 almost all the content in the files generated by bake are controlled by templates and the `TemplateTask`.

The `FixtureTask` not only generates fixtures with dummy data, but using the interactive options or the `-records` option you can enable fixture generation using live data.

New bake command

New commands have been added to make baking easier and faster. Controller, Model, View baking all feature an `all` subcommand, that builds everything at once and makes speedy rebuilds easy.

```
1.      cake bake model all
```

Would bake all the models for an application in one shot. Similarly `cake bake controller all` would bake all controllers and `cake bake view all` would generate all view files. Parameters on the `ControllerTask` have changed as well. `cake bake controller scaffold` is now `cake bake controller public`. `ViewTask` has had an `-admin` flag added, using `-admin` will allow you to bake views for actions that begin with `Routing.admin`

As mentioned before `cake bake fixture` and `cake bake test` are new, and have several subcommands each. `cake bake fixture all` will regenerate all the basic fixtures for your application. The `-count` parameter allows you to set the number of fake records that are created. By running fixture task interactively you can generate fixtures using the data in your live tables. You can use `cake bake test <type> <class>` to create test cases for already created objects in your app. Type should be one of the standard CakePHP types ('component', 'controller', 'model', 'helper', 'behavior') but doesn't have to be. Class should be an existing object of the chosen type.

Templates Galore

New in bake for 1.3 is the addition of more templates. In 1.2 baked views used templates that could be changed to modify the view files bake generated. In 1.3 templates are used to generate all output from bake. There are separate templates for controllers, controller action sets, fixtures, models, test cases, and the view files from 1.2. As well as more templates, you can also have multiple template sets or, bake themes. Bake themes can be provided in your app, or as part of plugins. An example plugin path for bake theme would be `app/plugins/bake_theme/vendors/shells/templates/dark_red/`. An app bake theme called `blue_bunny` would be placed in `app/vendors/shells/templates/blue_bunny`. You can look at `cake/console/templates/default/` to see what directories and files are required of a bake theme. However, like view files, if your bake theme doesn't implement a template, other installed themes will be checked until the correct template is found.

Additional plugin support.

New in 1.3 are additional ways to specify plugin names when using bake. In addition to `cake bake plugin Todo controller Posts`, there are two new forms. `cake bake controller Todo.Posts` and `cake bake controller Posts -plugin Todo`. The plugin parameter can be while using interactive bake as well. `cake bake controller -plugin Todo`, for example will allow you to use interactive bake to add controllers to your Todo plugin. Additional / multiple plugin paths are supported as well. In the past bake required your plugin to be in app/plugins. In 1.3 bake will find which of the pluginPaths the named plugin is located on, and add the files there.

9.2 Schema management and migrations

The SchemaShell provides a functionality to create schema objects, schema sql dumps as well as create snapshots and restore database snapshots.

9.2.1 Generating and using Schema files

A generated schema file allows you to easily transport a database agnostic schema. You can generate a schema file of your database using:

```
$ cake schema generate
```

This will generate a schema.php file in your `app/config/schema` directory.

The schema shell will only process tables for which there are models defined. To force the schema shell to process all the tables, you must add the `-f` option in the command line.

To later rebuild the database schema from your previously made schema.php file run:

```
$ cake schema create
```

This will drop and create the tables based on the contents of the schema.php.

Schema files can also be used to generate sql dump files. To generate a sql file containing the `CREATE TABLE` statements, run:

```
$ cake schema dump -write filename.sql
```

Where filename.sql is the desired filename for the sql dump. If you omit filename.sql the sql dump will be output to the console but not written to a file.

9.2.2 Migrations with CakePHP schema shell

Migrations allow for versioning of your database schema, so that as you develop features you have an easy and database agnostic way to distribute database changes. Migrations are achieved through either SCM controlled schema files or schema snapshots. Versioning a schema file with the schema shell is quite easy. If you already have a schema file created running

```
$ cake schema generate
```

Will bring up the following choices:

```
Generating Schema...
Schema file exists.
 [O]verwrite
 [S]napshot
 [Q]uit
Would you like to do? (o/s/q)
```

Choosing [s] (snapshot) will create an incremented schema.php. So if you have schema.php, it will create schema_2.php and so on. You can then restore to any of these schema files at any time by running:

```
$ cake schema update -s 2
```

Where 2 is the snapshot number you wish to run. The schema shell will prompt you to confirm you wish to perform the `ALTER` statements that represent the difference between the existing database the currently executing schema file.

You can perform a dry run by adding a `-dry` to your command.

9.3 Modify default HTML produced by "baked" templates

If you wish to modify the default HTML output produced by the "bake" command, follow these simple steps:

For baking custom views:

1. Go into: cake/console/templates/default/views
2. Notice the 4 files there
3. Copy them to your: app/vendors/shells/templates/[themename]/views
4. Make changes to the HTML output to control the way "bake" builds your views

The `[themename]` path segment should be the name of the bake theme that you are creating. Bake theme names need to be unique, so don't use 'default'.

For baking custom projects:

1. Go into: cake/console/templates/skel
2. Notice the base application files there
3. Copy them to your: app/vendors/shells/templates/skel
4. Make changes to the HTML output to control the way "bake" builds your views
5. Pass the skeleton path parameter to the project task

```
1.      cake bake project -skel vendors/shells/templates/skel
```

Notes

- You must run the specific project task `cake bake project` so that the path parameter can be passed.
- The template path is relative to the current path of the Command Line Interface.
- Since the full path to the skeleton needs to be manually entered, you can specify any directory holding your template build you want, including using multiple templates. (Unless Cake starts supporting overriding the skel folder like it does for views)

10 Deployment

Steps to deploy on a Hosting Server

11 Tutorials & Examples

In this section, you can walk through typical CakePHP applications to see how all of the pieces come together.

Alternatively, you can refer to CakeForge and the Bakery for existing applications and components. Don't forget that you can also view the source code of this cook book.

11.1 Blog

Welcome to CakePHP. You're probably checking out this tutorial because you want to learn more about how CakePHP works. It's our aim to increase productivity and make coding more enjoyable: we hope you'll see this as you dive into the code.

This tutorial will walk you through the creation of a simple blog application. We'll be getting and installing Cake, creating and configuring a database, and creating enough application logic to list, add, edit, and delete blog posts.

Here's what you'll need:

1. A running web server. We're going to assume you're using Apache, though the instructions for using other servers should be very similar. We might have to play a little with the server configuration, but most folks can get Cake up and running without any configuration at all.
2. A database server. We're going to be using MySQL server in this tutorial. You'll need to know enough about SQL in order to create a database: Cake will be taking the reins from there.
3. Basic PHP knowledge. The more object-oriented programming you've done, the better: but fear not if you're a procedural fan.
4. Finally, you'll need a basic knowledge of the MVC programming pattern. A quick overview can be found in Chapter "Beginning With CakePHP", Section : Understanding Model-View-Controller. Don't worry: its only a half a page or so.

Let's get started!

11.1.1 Getting Cake

First, let's get a copy of fresh Cake code.

To get a fresh download, visit the CakePHP project on github: http://github.com/cakephp/cakephp/downloads and download the latest release of 1.3

You can also clone the the repository using git. `git clone git://github.com/cakephp/cakephp.git`

Regardless of how you downloaded it, place the code inside of your DocumentRoot. Once finished, your directory setup should look something like the following:

```
/path_to_document_root
    /app
    /cake
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

Now might be a good time to learn a bit about how Cake's directory structure works: check out Chapter "Basic Principles of CakePHP", Section : CakePHP File Structure.

11.1.2 Creating the Blog Database

Next, lets set up the underlying database for our blog. if you haven't already done so, create an empty database for use in this tutorial, with a name of your choice. Right now, we'll just create a single table to store our posts. We'll also throw in a few posts right now to use for testing purposes. Execute the following SQL statements into your database:

```
/* First, create our posts table: */
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
/* Then insert some posts for testing: */
INSERT INTO posts (title,body,created)
```

```
11.         VALUES ('The title', 'This is the post body.', NOW());
12.     INSERT INTO posts (title,body,created)
13.         VALUES ('A title once again', 'And the post body follows.', NOW());
14.     INSERT INTO posts (title,body,created)
15.         VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

The choices on table and column names are not arbitrary. If you follow Cake's database naming conventions, and Cake's class naming conventions (both outlined in "CakePHP Conventions"), you'll be able to take advantage of a lot of free functionality and avoid configuration. Cake is flexible enough to accomodate even the worst legacy database schema, but adhering to convention will save you time.

Check out "CakePHP Conventions" for more information, but suffice it to say that naming our table 'posts' automatically hooks it to our Post model, and having fields called 'modified' and 'created' will be automagically managed by Cake.

11.1.3 Cake Database Configuration

Onward and upward: let's tell Cake where our database is and how to connect to it. For many, this is the first and last time you configure anything.

A copy of CakePHP's database configuration file is found in `/app/config/database.php.default`. Make a copy of this file in the same directory, but name it `database.php`.

The config file should be pretty straightforward: just replace the values in the `$default` array with those that apply to your setup. A sample completed configuration array might look something like the following:

```
1.     var $default = array(
2.         'driver' => 'mysql',
3.         'persistent' => 'false',
4.         'host' => 'localhost',
5.         'port' => '',
6.         'login' => 'cakeBlog',
7.         'password' => 'c4k3-rUl3Z',
```

```
8.          'database' => 'cake_blog_tutorial',
9.          'schema' => '',
10.         'prefix' => '',
11.         'encoding' => ''
12.     );
```

Once you've saved your new `database.php` file, you should be able to open your browser and see the Cake welcome page. It should also tell you that your database connection file was found, and that Cake can successfully connect to the database.

11.1.4 Optional Configuration

There are three other items that can be configured. Most developers complete these laundry-list items, but they're not required for this tutorial. One is defining a custom string (or "salt") for use in security hashes. The second is defining a custom number (or "seed") for use in encryption. The third item is allowing CakePHP write access to its `tmp` folder.

The security salt is used for generating hashes. Change the default salt value by editing `/app/config/core.php` line 203. It doesn't much matter what the new value is, as long as it's not easily guessed.

```
1.      <?php
2.      /**
3.       * A random string used in security hashing methods.
4.       */
5.      Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
6.      ?>
```

The cipher seed is used for encrypt/decrypt strings. Change the default seed value by editing `/app/config/core.php` line 208. It doesn't much matter what the new value is, as long as it's not easily guessed.

```
1.      <?php
2.      /**
```

```
3.          * A random numeric string (digits only) used to encrypt/decrypt strings.
4.          */
5.         Configure::write('Security.cipherSeed', '74857126596251478436398467 51');
6.         ?>
```

The final task is to make the `app/tmp` directory web-writable. The best way to do this is to find out what user your webserver runs as (`<?php echo `whoami`; ?>`) and change the ownership of the `app/tmp` directory to that user. The final command you run (in *nix) might look something like this.

```
1.         $ chown -R www-data app/tmp
```

If for some reason CakePHP can't write to that directory, you'll be informed by a warning while not in production mode.

11.1.5 A Note on mod_rewrite

Occasionally a new user will run in to mod_rewrite issues, so I'll mention them marginally here. If the CakePHP welcome page looks a little funny (no images or css styles), it probably means mod_rewrite isn't functioning on your system. Here are some tips to help get you up and running:

1. Make sure that an `.htaccess` override is allowed: in your `httpd.conf`, you should have a section that defines a section for each Directory on your server. Make sure the `AllowOverride` is set to `All` for the correct Directory. For security and performance reasons, do *not* set `AllowOverride` to `All` in `<Directory />`. Instead, look for the `<Directory>` block that refers to your actual website directory.
2. Make sure you are editing the correct `httpd.conf` rather than a user- or site-specific `httpd.conf`.
3. For some reason or another, you might have obtained a copy of CakePHP without the needed `.htaccess` files. This sometimes happens because some operating systems treat files that start with '.' as hidden, and don't copy them. Make sure your copy of CakePHP is from the downloads section of the site or our git repository.
4. Make sure Apache is loading up mod_rewrite correctly! You should see something like `LoadModule rewrite_module libexec/httpd/mod_rewrite.so` or (for Apache 1.3) `AddModule mod_rewrite.c` in your `httpd.conf`.

If you don't want or can't get mod_rewrite (or some other compatible module) up and running on your server, you'll need to use Cake's built in pretty URLs. In `/app/config/core.php`, uncomment the line that looks like:

```
1.      Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

Also remove these `.htaccess` files:

```
1.              /.htaccess
2.              /app/.htaccess
3.              /app/webroot/.htaccess
4.
```

This will make your URLs look like `www.example.com/index.php/controllername/actionname/param` rather than `www.example.com/controllername/actionname/param`.

If you are installing CakePHP on a webserver besides Apache, you can find instructions for getting URL rewriting working for other servers under the Installation section.

11.1.6 Create a Post Model

The Model class is the bread and butter of CakePHP applications. By creating a CakePHP model that will interact with our database, we'll have the foundation in place needed to do our view, add, edit, and delete operations later.

CakePHP's model class files go in `/app/models`, and the file we'll be creating will be saved to `/app/models/post.php`. The completed file should look like this:

```
1.      <?php
2.      class Post extends AppModel {
3.          var $name = 'Post';
4.      }
5.      ?>
```

Naming convention is very important in CakePHP. By naming our model Post, CakePHP can automatically infer that this model will be used in the PostsController, and will be tied to a database table called `posts`.

CakePHP will dynamically create a model object for you, if it cannot find a corresponding file in `/app/models`. This also means, that if you accidentally name your file wrong (i.e. `Post.php` or `posts.php`) CakePHP will not recognize any of your settings and will use the defaults instead.

The `$name` variable is always a good idea to add, and is used to overcome some class name oddness in PHP4.

For more on models, such as table prefixes, callbacks, and validation, check out the Models chapter of the Manual.

11.1.7 Create a Posts Controller

Next, we'll create a controller for our posts. The controller is where all the business logic for post interaction will happen. In a nutshell, it's the place where you play with the models and get post-related work done. We'll place this new controller in a file called `posts_controller.php` inside the `/app/controllers` directory. Here's what the basic controller should look like:

```
1.  <?php
2.  class PostsController extends AppController {
3.      var $helpers = array ('Html','Form');
4.      var $name = 'Posts';
5.  }
6.  ?>
```

Now, lets add an action to our controller. Actions often represent a single function or interface in an application. For example, when users request `www.example.com/posts/index` (which is also the same as `www.example.com/posts/`), they might expect to see a listing of posts. The code for that action would look something like this:

```
1.  <?php
2.  class PostsController extends AppController {
3.      var $helpers = array ('Html','Form');
```

```
4.          var $name = 'Posts';
5.          function index() {
6.              $this->set('posts', $this->Post->find('all'));
7.          }
8.      }
9.      ?>
```

Let me explain the action a bit. By defining function `index()` in our PostsController, users can now access the logic there by requesting `www.example.com/posts/index`. Similarly, if we were to define a function called `foobar()`, users would be able to access that at `www.example.com/posts/foobar`.

You may be tempted to name your controllers and actions a certain way to obtain a certain URL. Resist that temptation. Follow CakePHP conventions (plural controller names, etc.) and create readable, understandable action names. You can map URLs to your code using "routes" covered later on.

The single instruction in the action uses `set()` to pass data from the controller to the view (which we'll create next). The line sets the view variable called 'posts' equal to the return value of the `find('all')` method of the Post model. Our Post model is automatically available at `$this->Post` because we've followed Cake's naming conventions.

To learn more about Cake's controllers, check out Chapter "Developing with CakePHP" section: "Controllers".

11.1.8 Creating Post Views

Now that we have our data flowing to our model, and our application logic and flow defined by our controller, let's create a view for the index action we created above.

Cake views are just presentation-flavored fragments that fit inside an application's layout. For most applications they're HTML mixed with PHP, but they may end up as XML, CSV, or even binary data.

Layouts are presentation code that is wrapped around a view, and can be defined and switched between, but for now, let's just use the default.

Remember in the last section how we assigned the 'posts' variable to the view using the `set()` method? That would hand down data to the view that would look something like this:

```
// print_r($posts) output:

Array
(
        [0] => Array
                (
                        [Post] => Array
                                (
                                        [id] => 1
                                        [title] => The title
                                        [body] => This is the post body.
                                        [created] => 2008-02-13 18:34:55
                                        [modified] =>
                                )
                )
        [1] => Array
                (
                        [Post] => Array
                                (
                                        [id] => 2
                                        [title] => A title once again
                                        [body] => And the post body follows.
                                        [created] => 2008-02-13 18:34:56
                                        [modified] =>
                                )
                )
        [2] => Array
                (
                        [Post] => Array
                                (
                                        [id] => 3
                                        [title] => Title strikes back
                                        [body] => This is really exciting! Not.
                                        [created] => 2008-02-13 18:34:57
                                        [modified] =>
```

```
                                    )
                )
)
```

Cake's view files are stored in `/app/views` inside a folder named after the controller they correspond to (we'll have to create a folder named 'posts' in this case). To format this post data in a nice table, our view code might look something like this:

```
1.      <!-- File: /app/views/posts/index.ctp -->
2.      <h1>Blog posts</h1>
3.      <table>
4.          <tr>
5.              <th>Id</th>
6.              <th>Title</th>
7.              <th>Created</th>
8.          </tr>
9.          <!-- Here is where we loop through our $posts array, printing out post info -->
10.         <?php foreach ($posts as $post): ?>
11.         <tr>
12.             <td><?php echo $post['Post']['id']; ?></td>
13.             <td>
14.           <?php echo $this->Html->link($post['Post']['title'],
15.     array('controller' => 'posts', 'action' => 'view', $post['Post']['id'])); ?>
16.             </td>
17.         <td><?php echo $post['Post']['created']; ?></td>
18.         </tr>
19.       <?php endforeach; ?>
20.
21.     </table>
```

Hopefully this should look somewhat simple.

You might have noticed the use of an object called `$this->Html`. This is an instance of the CakePHP `HtmlHelper` class. CakePHP comes with a set of view helpers that make things like linking, form output, JavaScript and Ajax a snap. You can learn more about how to use them in Chapter "Built-in Helpers", but what's important to note here is that the `link()` method will generate an HTML link with the given title (the first parameter) and URL (the second parameter).

When specifying URLs in Cake, it is recommended that you use the array format. This is explained in more detail in the section on Routes. Using the array format for URLs allows you to take advantage of CakePHP's reverse routing capabilities. You can also specify URLs relative to the base of the application in the form of `/controller/action/param1/param2`.

At this point, you should be able to point your browser to `http://www.example.com/posts/index`. You should see your view, correctly formatted with the title and table listing of the posts.

If you happened to have clicked on one of the links we created in this view (that link a post's title to a URL `/posts/view/some_id`), you were probably informed by CakePHP that the action hasn't yet been defined. If you were not so informed, either something has gone wrong, or you actually did define it already, in which case you are very sneaky. Otherwise, we'll create it in the PostsController now:

```
1.  <?php
2.  class PostsController extends AppController {
3.      var $helpers = array('Html', 'Form');
4.      var $name = 'Posts';
5.      function index() {
6.          $this->set('posts', $this->Post->find('all'));
7.      }
8.      function view($id = null) {
9.          $this->Post->id = $id;
10.         $this->set('post', $this->Post->read());
11.     }
12. }
13. ?>
```

The `set()` call should look familiar. Notice we're using `read()` rather than `find('all')` because we only really want a single post's information.

Notice that our view action takes a parameter: the ID of the post we'd like to see. This parameter is handed to the action through the requested URL. If a user requests `/posts/view/3`, then the value '3' is passed as `$id`.

Now let's create the view for our new 'view' action and place it in `/app/views/posts/view.ctp`.

```
<!-- File: /app/views/posts/view.ctp -->
<h1><?php echo $post['Post']['title']?></h1>
<p><small>Created: <?php echo $post['Post']['created']?></small></p>
<p><?php echo $post['Post']['body']?></p>
```

Verify that this is working by trying the links at `/posts/index` or manually requesting a post by accessing `/posts/view/1`.

11.1.9 Adding Posts

Reading from the database and showing us the posts is a great start, but let's allow for the adding of new posts.

First, start by creating an `add()` action in the PostsController:

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';
    var $components = array('Session');
    function index() {
        $this->set('posts', $this->Post->find('all'));
    }
    function view($id) {
        $this->Post->id = $id;
        $this->set('post', $this->Post->read());
    }
    function add() {
        if (!empty($this->data)) {
```

```
14.                     if ($this->Post->save($this->data)) {
15.                         $this->Session->setFlash('Your post has been saved.');
16.                         $this->redirect(array('action' => 'index'));
17.                     }
18.                 }
19.             }
20.         }
21.         ?>
```

You need to include the SessionComponent - and SessionHelper - in any controller where you will use it. If necessary, include it in your AppController.

Here's what the `add()` action does: if the submitted form data isn't empty, try to save the data using the Post model. If for some reason it doesn't save, just render the view. This gives us a chance to show the user validation errors or other warnings.

When a user uses a form to POST data to your application, that information is available in `$this->data`. You can use the `pr()` or `debug` functions to print it out if you want to see what it looks like.

We use the `Session` component's `setFlash()` function to set a message to a session variable to be displayed on the page after redirection. In the layout we have `$session->flash()` which displays the message and clears the corresponding session variable. The controller's `redirect` function redirects to another URL. The param `array('action'=>'index)` translates to URL `/posts` i.e the index action of posts controller. You can refer to Router::url function on the api to see the formats in which you can specify a URL for various cake functions.

Calling the `save()` method will check for validation errors and abort the save if any occur. We'll discuss how those errors are handled in the following sections.

11.1.10 Data Validation

Cake goes a long way in taking the monotony out of form input validation. Everyone hates coding up endless forms and their validation routines. CakePHP makes it easier and faster.

To take advantage of the validation features, you'll need to use Cake's FormHelper in your views. The FormHelper is available by default to all views at `$this->Form`.

Here's our add view:

```
1.      <!-- File: /app/views/posts/add.ctp -->
2.
3.      <h1>Add Post</h1>
4.      <?php
5.      echo $this->Form->create('Post');
6.      echo $this->Form->input('title');
7.      echo $this->Form->input('body', array('rows' => '3'));
8.      echo $this->Form->end('Save Post');
9.      ?>
```

Here, we use the FormHelper to generate the opening tag for an HTML form. Here's the HTML that `$this->Form->create()` generates:

```
1.      <form id="PostAddForm" method="post" action="/posts/add">
```

If `create()` is called with no parameters supplied, it assumes you are building a form that submits to the current controller's `add()` action (or `edit()` action when `id` is included in the form data), via POST.

The `$this->Form->input()` method is used to create form elements of the same name. The first parameter tells CakePHP which field they correspond to, and the second parameter allows you to specify a wide array of options - in this case, the number of rows for the textarea. There's a bit of introspection and automagic here: `input()` will output different form elements based on the model field specified.

The `$this->Form->end()` call generates a submit button and ends the form. If a string is supplied as the first parameter to `end()`, the FormHelper outputs a submit button named accordingly along with the closing form tag. Again, refer to Chapter "Built-in Helpers" for more on helpers.

Now let's go back and update our `/app/views/posts/index.ctp` view to include a new "Add Post" link. Before the `<table>`, add the following line:

```
1.      <?php echo $this->Html->link('Add Post', array('controller' => 'posts', 'action' => 'add')); ?>
```

You may be wondering: how do I tell CakePHP about my validation requirements? Validation rules are defined in the model. Let's look back at our Post model and make a few adjustments:

```
1.      <?php
2.      class Post extends AppModel
3.      {
4.          var $name = 'Post';
5.          var $validate = array(
6.              'title' => array(
7.                  'rule' => 'notEmpty'
8.              ),
9.              'body' => array(
10.                 'rule' => 'notEmpty'
11.             )
12.         );
13.     }
14.     ?>
```

The `$validate` array tells CakePHP how to validate your data when the `save()` method is called. Here, I've specified that both the body and title fields must not be empty. CakePHP's validation engine is strong, with a number of pre-built rules (credit card numbers, email addresses, etc.) and flexibility for adding your own validation rules. For more information on that setup, check the Data Validation chapter.

Now that you have your validation rules in place, use the app to try to add a post with an empty title or body to see how it works. Since we've used the `input()` method of the FormHelper to create our form elements, our validation error messages will be shown automatically.

11.1.11 Deleting Posts

Next, let's make a way for users to delete posts. Start with a `delete()` action in the PostsController:

```
1.      function delete($id) {
2.          if ($this->Post->delete($id)) {
3.              $this->Session->setFlash('The post with id: ' . $id . ' has been deleted.');
4.              $this->redirect(array('action' => 'index'));
5.          }
6.      }
```

This logic deletes the post specified by $id, and uses `$this->Session->setFlash()` to show the user a confirmation message after redirecting them on to /posts.

Because we're just executing some logic and redirecting, this action has no view. You might want to update your index view with links that allow users to delete posts, however:

```
1.      <!-- File: /app/views/posts/index.ctp -->
2.      <h1>Blog posts</h1>
3.      <p><?php echo $this->Html->link('Add Post', array('action' => 'add')); ?></p>
4.      <table>
5.          <tr>
6.              <th>Id</th>
7.              <th>Title</th>
8.                      <th>Actions</th>
9.              <th>Created</th>
10.         </tr>
11.     <!-- Here's where we loop through our $posts array, printing out post info -->
12.       <?php foreach ($posts as $post): ?>
13.         <tr>
14.         <td><?php echo $post['Post']['id']; ?></td>
15.             <td>
16.         <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view', $post['Post']['id']));?>
17.             </td>
```

```
18.                 <td>
19.                 <?php echo $this->Html->link('Delete', array('action' => 'delete', $post['Post']['id']), null, 'Are you sure?')?>
20.                 </td>
21.             <td><?php echo $post['Post']['created']; ?></td>
22.             </tr>
23.           <?php endforeach; ?>
24.
25.         </table>
```

This view code also uses the HtmlHelper to prompt the user with a JavaScript confirmation dialog before they attempt to delete a post.

11.1.12 ing Posts

Post editing: here we go. You're a CakePHP pro by now, so you should have picked up a pattern. Make the action, then the view. Here's what the `edit()` action of the PostsController would look like:

```
1.         function edit($id = null) {
2.             $this->Post->id = $id;
3.             if (empty($this->data)) {
4.                 $this->data = $this->Post->read();
5.             } else {
6.                 if ($this->Post->save($this->data)) {
7.                     $this->Session->setFlash('Your post has been updated.');
8.                     $this->redirect(array('action' => 'index'));
9.                 }
10.             }
11.         }
```

This action first checks for submitted form data. If nothing was submitted, it finds the Post and hands it to the view. If some data *has* been submitted, try to save the data using Post model (or kick back and show the user the validation errors).

The edit view might look something like this:

```
1.      <!-- File: /app/views/posts/edit.ctp -->
2.
3.      <h1> Post</h1>
4.      <?php
5.          echo $this->Form->create('Post', array('action' => 'edit'));
6.          echo $this->Form->input('title');
7.          echo $this->Form->input('body', array('rows' => '3'));
8.          echo $this->Form->input('id', array('type' => 'hidden'));
9.          echo $this->Form->end('Save Post');
10.     ?>
```

This view outputs the edit form (with the values populated), along with any necessary validation error messages.

One thing to note here: CakePHP will assume that you are editing a model if the 'id' field is present in the data array. If no 'id' is present (look back at our add view), Cake will assume that you are inserting a new model when `save()` is called.

You can now update your index view with links to edit specific posts:

```
1.      <!-- File: /app/views/posts/index.ctp  (edit links added) -->
2.
3.      <h1>Blog posts</h1>
4.      <p><?php echo $this->Html->link("Add Post", array('action' => 'add')); ?></p>
5.      <table>
6.          <tr>
7.              <th>Id</th>
8.              <th>Title</th>
```

```
9.                      <th>Action</th>
10.                 <th>Created</th>
11.             </tr>
12.         <!-- Here's where we loop through our $posts array, printing out post info -->
13.         <?php foreach ($posts as $post): ?>
14.             <tr>
15.             <td><?php echo $post['Post']['id']; ?></td>
16.                 <td>
17.               <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view', $post['Post']['id']));?>
18.                     </td>
19.                     <td>
20.                   <?php echo $this->Html->link(
21.                       'Delete',
22.                       array('action' => 'delete', $post['Post']['id']),
23.                       null,
24.                       'Are you sure?'
25.                   )?>
26.                   <?php echo $this->Html->link('', array('action' => 'edit', $post['Post']['id']));?>
27.                 </td>
28.             <td><?php echo $post['Post']['created']; ?></td>
29.             </tr>
30.         <?php endforeach; ?>
31.
32.         </table>
```

11.1.13 Routes

For some, CakePHP's default routing works well enough. Developers who are sensitive to user-friendliness and general search engine compatibility will appreciate the way that CakePHP's URLs map to specific actions. So we'll just make a quick change to routes in this tutorial.

For more information on advanced routing techniques, see "Routes Configuration".

By default, CakePHP responds to a request for the root of your site (i.e. http://www.example.com) using its PagesController, rendering a view called "home". Instead, we'll replace this with our PostsController by creating a routing rule.

Cake's routing is found in `/app/config/routes.php`. You'll want to comment out or remove the line that defines the default root route. It looks like this:

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

This line connects the URL '/' with the default CakePHP home page. We want it to connect with our own controller, so replace that line with this one:

```
1.      Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

This should connect users requesting '/' to the index() action of our PostsController.

CakePHP also makes use of 'reverse routing' - if with the above route defined you pass `array('controller' => 'posts', 'action' => 'index')` to a function expecting an array, the resultant URL used will be '/'. It's therefore a good idea to always use arrays for URLs as this means your routes define where a URL goes, and also ensures that links point to the same place too.

11.1.14 Conclusion

Creating applications this way will win you peace, honor, love, and money beyond even your wildest fantasies. Simple, isn't it? Keep in mind that this tutorial was very basic. CakePHP has *many* more features to offer, and is flexible in ways we didn't wish to cover here for simplicity's sake. Use the rest of this manual as a guide for building more feature-rich applications.

Now that you've created a basic Cake application you're ready for the real thing. Start your own project, read the rest of the Manual and API.

If you need help, come see us in #cakephp. Welcome to CakePHP!

Suggested Follow-up Reading

These are common tasks people learning CakePHP usually want to study next:

1. Layouts: Customizing your website layout
2. Elements: Including and reusing view snippets
3. Scaffolding: Prototyping before creating code
4. Baking: Generating basic CRUD code
5. Authentication: User registration and login

11.2 Simple Acl controlled Application

In this tutorial you will create a simple application with authentication and access control lists. This tutorial assumes you have read the Blog tutorial, and you are familiar with Bake. You should have some experience with CakePHP, and be familiar with MVC concepts. This tutorial is a brief introduction to the `AuthComponent` and `AclComponent`.

What you will need

1. A running web server. We're going to assume you're using Apache, though the instructions for using other servers should be very similar. We might have to play a little with the server configuration, but most folks can get Cake up and running without any configuration at all.
2. A database server. We're going to be using MySQL in this tutorial. You'll need to know enough about SQL in order to create a database: Cake will be taking the reins from there.
3. Basic PHP knowledge. The more object-oriented programming you've done, the better: but fear not if you're a procedural fan.

11.2.1 Preparing our Application

First, let's get a copy of fresh Cake code.

To get a fresh download, visit the CakePHP project at Cakeforge: http://github.com/cakephp/cakephp/downloads and download the stable release. For this tutorial you need the latest 1.3 release.

You can also checkout/export a fresh copy of our trunk code at: git://github.com/cakephp/cakephp.git

Once you've got a fresh copy of cake setup your database.php config file, and change the value of Security.salt in your app/config/core.php. From there we will build a simple database schema to build our application on. Execute the following SQL statements into your database.

```
CREATE TABLE users (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    password CHAR(40) NOT NULL,
    group_id INT(11) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE groups (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE posts (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id INT(11) NOT NULL,
    title VARCHAR(255) NOT NULL,
    body TEXT,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE widgets (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    part_no VARCHAR(12),
    quantity INT(11)
);
```

These are the tables we will be using to build the rest of our application. Once we have the table structure in the database we can start cooking. Use cake bake to quickly create your models, controllers, and views.

To use cake bake, call "`cake bake all`" and this will list the 4 tables you inserted into mySQL. Select "1. Group", and follow the prompts. Repeat for the other 3 tables, and this will have generated the 4 controllers, models and your views for you.

Avoid using Scaffold here. The generation of the ACOs will be seriously affected if you bake the controllers with the Scaffold feature.

While baking the Models cake will automagically detect the associations between your Models (or relations between your tables). Let cake supply the correct hasMany and belongsTo associations. If you are prompted to pick hasOne or hasMany, generally speaking you'll need a hasMany (only) relationships for this tutorial.

Leave out admin routing for now, this is a complicated enough subject without them. Also be sure **not** to add either the Acl or Auth Components to any of your controllers as you are baking them. We'll be doing that soon enough. You should now have models, controllers, and baked views for your users, groups, posts and widgets.

11.2.2 Preparing to Add Auth

We now have a functioning CRUD application. Bake should have setup all the relations we need, if not add them in now. There are a few other pieces that need to be added before we can add the Auth and Acl components. First add a login and logout action to your `UsersController`.

```
1.      function login() {
2.          //Auth Magic
3.      }
4.
5.      function logout() {
6.          //Leave empty for now.
7.      }
```

Then create the following view file for login at app/views/users/login.ctp:

```
<?php
$this->Session->flash('auth');
echo $this->Form->create('User', array('action' => 'login'));
echo $this->Form->inputs(array(
    'legend' => __('Login', true),
    'username',
    'password'
));
echo $this->Form->end('Login');
?>
```

We don't need to worry about adding anything to hash passwords, as AuthComponent will do this for us automatically when creating/editing users, and when they login, once configured properly. Furthermore, if you hash incoming passwords manually `AuthComponent` will not be able to log you in at all. As it will hash them again, and they will not match.

Next we need to make some modifications to `AppController`. If you don't have `/app/app_controller.php`, create it. Note that this goes in `/app/`, not `/app/controllers/`. Since we want our entire site controlled with Auth and Acl, we will set them up in `AppController`.

```
<?php
class AppController extends Controller {
    var $components = array('Acl', 'Auth', 'Session');
    var $helpers = array('Html', 'Form', 'Session');
    function beforeFilter() {
        //Configure AuthComponent
        $this->Auth->authorize = 'actions';
        $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');
        $this->Auth->logoutRedirect = array('controller' => 'users', 'action' => 'logout');
        $this->Auth->loginRedirect = array('controller' => 'posts', 'action' => 'add');
    }
}
```

```
13.         ?>
```

Before we set up the ACL at all we will need to add some users and groups. With `AuthComponent` in use we will not be able to access any of our actions, as we are not logged in. We will now add some exceptions so `AuthComponent` will allow us to create some groups and users. In **both** your `GroupsController` and your `UsersController` Add the following.

```
1.          function beforeFilter() {
2.              parent::beforeFilter();
3.              $this->Auth->allow(array('*'));
4.          }
```

These statements tell AuthComponent to allow public access to all actions. This is only temporary and will be removed once we get a few users and groups into our database. Don't add any users or groups just yet though.

11.2.3 Initialize the Db Acl tables

Before we create any users or groups we will want to connect them to the Acl. However, we do not at this time have any Acl tables and if you try to view any pages right now, you will get a missing table error ("Error: Database table acos for model Aco was not found."). To remove these errors we need to run a schema file. In a shell run the following:

```
cake schema create DbAcl
```

This schema will prompt you to drop and create the tables. Say yes to dropping and creating the tables.

If you don't have shell access, or are having trouble using the console, you can run the sql file found in `/path/to/app/config/schema/db_acl.sql`.

With the controllers setup for data entry, and the Acl tables initialized we are ready to go right? Not entirely, we still have a bit of work to do in the user and group models. Namely, making them auto-magically attach to the Acl.

11.2.4 Acts As a Requester

For Auth and Acl to work properly we need to associate our users and groups to rows in the Acl tables. In order to do this we will use the `AclBehavior`. The `AclBehavior` allows for the automagic connection of models with the Acl tables. Its use requires an implementation of `parentNode()` on your model. In our `User` model we will add the following.

```
1.      var $name = 'User';
2.      var $belongsTo = array('Group');
3.      var $actsAs = array('Acl' => array('type' => 'requester'));
4.
5.      function parentNode() {
6.          if (!$this->id && empty($this->data)) {
7.              return null;
8.          }
9.          if (isset($this->data['User']['group_id'])) {
10.         $groupId = $this->data['User']['group_id'];
11.         } else {
12.             $groupId = $this->field('group_id');
13.         }
14.         if (!$groupId) {
15.         return null;
16.         } else {
17.             return array('Group' => array('id' => $groupId));
18.         }
19.     }
```

Then in our `Group` Model Add the following:

```
1.      var $actsAs = array('Acl' => array('type' => 'requester'));
2.
3.      function parentNode() {
4.          return null;
```

```
5.          }
```

What this does, is tie the `Group` and `User` models to the Acl, and tell CakePHP that every-time you make a User or Group you want an entry on the `aros` table as well. This makes Acl management a piece of cake as your AROs become transparently tied to your `users` and `groups` tables. So anytime you create or delete a user/group the Aro table is updated.

Our controllers and models are now prepped for adding some initial data, and our `Group` and `User` models are bound to the Acl table. So add some groups and users using the baked forms by browsing to `http://example.com/groups/add` and `http://example.com/users/add`. I made the following groups:

- administrators
- managers
- users

I also created a user in each group so I had a user of each different access group to test with later. Write everything down or use easy passwords so you don't forget. If you do a `SELECT * FROM aros;` from a mysql prompt you should get something like the following:

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    4 |
|  2 |      NULL | Group |           2 | NULL  |    5 |    8 |
|  3 |      NULL | Group |           3 | NULL  |    9 |   12 |
|  4 |         1 | User  |           1 | NULL  |    2 |    3 |
|  5 |         2 | User  |           2 | NULL  |    6 |    7 |
|  6 |         3 | User  |           3 | NULL  |   10 |   11 |
+----+-----------+-------+-------------+-------+------+------+
6 rows in set (0.00 sec)
```

This shows us that we have 3 groups and 3 users. The users are nested inside the groups, which means we can set permissions on a per-group or per-user basis.

11.2.4.1 Group-only ACL

In case we want simplified per-group only permissions, we need to implement `bindNode()` in `User` model.

```
1.      function bindNode($user) {
2.          return array('model' => 'Group', 'foreign_key' => $user['User']['group_id']);
3.      }
```

This method will tell ACL to skip checking `User` Aro's and to check only `Group` Aro's.

Every user has to have assigned `group_id` for this to work.

In this case our `aros` table will look like this:

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    2 |
|  2 |      NULL | Group |           2 | NULL  |    3 |    4 |
|  3 |      NULL | Group |           3 | NULL  |    5 |    6 |
+----+-----------+-------+-------------+-------+------+------+
3 rows in set (0.00 sec)
```

11.2.5 Creating ACOs (Access Control Objects)

-
- Comments (1)
- History

Now that we have our users and groups (aros), we can begin inputting our existing controllers into the Acl and setting permissions for our groups and users, as well as enabling login / logout.

Our ARO are automatically creating themselves when new users and groups are created. What about a way to auto-generate ACOs from our controllers and their actions? Well unfortunately there is no magic way in CakePHP's core to accomplish this. The core classes offer a few ways to manually create ACO's though. You can create ACO objects from the Acl shell or You can use the `AclComponent`. Creating Acos from the shell looks like:

```
cake acl create aco root controllers
```

While using the AclComponent would look like:

```
1.      $this->Acl->Aco->create(array('parent_id' => null, 'alias' => 'controllers'));
2.      $this->Acl->Aco->save();
```

Both of these examples would create our 'root' or top level ACO which is going to be called 'controllers'. The purpose of this root node is to make it easy to allow/deny access on a global application scope, and allow the use of the Acl for purposes not related to controllers/actions such as checking model record permissions. As we will be using a global root ACO we need to make a small modification to our `AuthComponent` configuration. `AuthComponent` needs to know about the existence of this root node, so that when making ACL checks it can use the correct node path when looking up controllers/actions. In `AppController` add the following to the `beforeFilter`:

```
1.      $this->Auth->actionPath = 'controllers/';
```

11.2.6 An Automated tool for creating ACOs

As mentioned before, there is no pre-built way to input all of our controllers and actions into the Acl. However, we all hate doing repetitive things like typing in what could be hundreds of actions in a large application. We've whipped up an automated set of functions to build the ACO table. These functions will look at every controller in your application. It will add any non-private, non `Controller` methods to the Acl table, nicely nested underneath the owning controller. You can add and run this in your `AppController` or any controller for that matter, just be sure to remove it before putting your application into production.

```
1.          function build_acl() {
2.              if (!Configure::read('debug')) {
```

```
3.                      return $this->_stop();
4.                  }
5.                  $log = array();
6.                  $aco =& $this->Acl->Aco;
7.                  $root = $aco->node('controllers');
8.                  if (!$root) {
9.                      $aco->create(array('parent_id' => null, 'model' => null, 'alias' => 'controllers'));
10.                     $root = $aco->save();
11.                     $root['Aco']['id'] = $aco->id;
12.                     $log[] = 'Created Aco node for controllers';
13.                 } else {
14.                     $root = $root[0];
15.                 }
16.                 App::import('Core', 'File');
17.                 $Controllers = App::objects('controller');
18.                 $appIndex = array_search('App', $Controllers);
19.                 if ($appIndex !== false ) {
20.                     unset($Controllers[$appIndex]);
21.                 }
22.                 $baseMethods = get_class_methods('Controller');
23.                 $baseMethods[] = 'build_acl';
24.                 $Plugins = $this->_getPluginControllerNames();
25.                 $Controllers = array_merge($Controllers, $Plugins);
26.                 // look at each controller in app/controllers
27.                 foreach ($Controllers as $ctrlName) {
28.                     $methods = $this->_getClassMethods($this->_getPluginControllerPath($ctrlName));
29.                     // Do all Plugins First
30.                     if ($this->_isPlugin($ctrlName)){
31.                         $pluginNode = $aco->node('controllers/'.$this->_getPluginName($ctrlName));
32.                         if (!$pluginNode) {
```

```
33.                             $aco->create(array('parent_id' => $root['Aco']['id'], 'model' => null, 'alias' => $this->_getPluginName($ctrlName)));
34.                             $pluginNode = $aco->save();
35.                             $pluginNode['Aco']['id'] = $aco->id;
36.                             $log[] = 'Created Aco node for ' . $this->_getPluginName($ctrlName) . ' Plugin';
37.                         }
38.                     }
39.                     // find / make controller node
40.                     $controllerNode = $aco->node('controllers/'.$ctrlName);
41.                     if (!$controllerNode) {
42.                         if ($this->_isPlugin($ctrlName)){
43.                             $pluginNode = $aco->node('controllers/' . $this->_getPluginName($ctrlName));
44.                             $aco->create(array('parent_id' => $pluginNode['0']['Aco']['id'], 'model' => null, 'alias' => $this->_getPluginControllerName($ctrlName)));
45.                             $controllerNode = $aco->save();
46.                             $controllerNode['Aco']['id'] = $aco->id;
47.                             $log[] = 'Created Aco node for ' . $this->_getPluginControllerName($ctrlName) . ' ' . $this->_getPluginName($ctrlName) . ' Plugin Controller';
48.                         } else {
49.                             $aco->create(array('parent_id' => $root['Aco']['id'], 'model' => null, 'alias' => $ctrlName));
50.                             $controllerNode = $aco->save();
51.                             $controllerNode['Aco']['id'] = $aco->id;
52.                             $log[] = 'Created Aco node for ' . $ctrlName;
53.                         }
54.                     } else {
55.                         $controllerNode = $controllerNode[0];
56.                     }
57.                     //clean the methods. to remove those in Controller and private actions.
58.                     foreach ($methods as $k => $method) {
59.                         if (strpos($method, '_', 0) === 0) {
```

```
                            unset($methods[$k]);
                            continue;
                        }
                        if (in_array($method, $baseMethods)) {
                            unset($methods[$k]);
                            continue;
                        }
                        $methodNode = $aco->node('controllers/'.$ctrlName.'/'.$method);
                        if (!$methodNode) {
                            $aco->create(array('parent_id' => $controllerNode['Aco']['id'], 'model' => null, 'alias' => $method));
                            $methodNode = $aco->save();
                            $log[] = 'Created Aco node for '. $method;
                        }
                    }
                }
                if(count($log)>0) {
                    debug($log);
                }
            }
            function _getClassMethods($ctrlName = null) {
                App::import('Controller', $ctrlName);
                if (strlen(strstr($ctrlName, '.')) > 0) {
                    // plugin's controller
                    $num = strpos($ctrlName, '.');
                    $ctrlName = substr($ctrlName, $num+1);
                }
                $ctrlclass = $ctrlName . 'Controller';
                $methods = get_class_methods($ctrlclass);
                // Add scaffold defaults if scaffolds are being used
                $properties = get_class_vars($ctrlclass);
```

```
90.                 if (array_key_exists('scaffold',$properties)) {
91.                     if($properties['scaffold'] == 'admin') {
92.                         $methods    =    array_merge($methods,    array('admin_add',    'admin_edit',    'admin_index',
    'admin_view', 'admin_delete'));
93.                     } else {
94.                         $methods = array_merge($methods, array('add', 'edit', 'index', 'view', 'delete'));
95.                     }
96.                 }
97.                 return $methods;
98.             }
99.             function _isPlugin($ctrlName = null) {
100.                 $arr = String::tokenize($ctrlName, '/');
101.                 if (count($arr) > 1) {
102.                     return true;
103.                 } else {
104.                     return false;
105.                 }
106.             }
107.             function _getPluginControllerPath($ctrlName = null) {
108.                 $arr = String::tokenize($ctrlName, '/');
109.                 if (count($arr) == 2) {
110.                     return $arr[0] . '.' . $arr[1];
111.                 } else {
112.                     return $arr[0];
113.                 }
114.             }
115.             function _getPluginName($ctrlName = null) {
116.                 $arr = String::tokenize($ctrlName, '/');
117.                 if (count($arr) == 2) {
118.                     return $arr[0];
119.                 } else {
```

```
120.                 return false;
121.             }
122.         }
123.         function _getPluginControllerName($ctrlName = null) {
124.             $arr = String::tokenize($ctrlName, '/');
125.             if (count($arr) == 2) {
126.                 return $arr[1];
127.             } else {
128.                 return false;
129.             }
130.         }
131.     /**
132.      * Get the names of the plugin controllers ...
133.      *
134.      * This function will get an array of the plugin controller names, and
135.      * also makes sure the controllers are available for us to get the
136.      * method names by doing an App::import for each plugin controller.
137.      *
138.      * @return array of plugin names.
139.      *
140.      */
141.         function _getPluginControllerNames() {
142.             App::import('Core', 'File', 'Folder');
143.             $paths = Configure::getInstance();
144.             $folder =& new Folder();
145.             $folder->cd(APP . 'plugins');
146.             // Get the list of plugins
147.             $Plugins = $folder->read();
148.             $Plugins = $Plugins[0];
149.             $arr = array();
150.             // Loop through the plugins
```

```
151.             foreach($Plugins as $pluginName) {
152.                 // Change directory to the plugin
153.                 $didCD = $folder->cd(APP . 'plugins'. DS . $pluginName . DS . 'controllers');
154.                 // Get a list of the files that have a file name that ends
155.                 // with controller.php
156.                 $files = $folder->findRecursive('.*_controller\.php');
157.                 // Loop through the controllers we found in the plugins directory
158.                 foreach($files as $fileName) {
159.                     // Get the base file name
160.                     $file = basename($fileName);
161.                     // Get the controller name
162.                     $file = Inflector::camelize(substr($file, 0, strlen($file)-strlen('_controller.php')));
163.                     if (!preg_match('/^'. Inflector::humanize($pluginName). 'App/', $file)) {
164.                         if (!App::import('Controller', $pluginName.'.'.$file)) {
165.                             debug('Error importing '.$file.' for plugin '.$pluginName);
166.                         } else {
167.                             /// Now prepend the Plugin name ...
168.                             // This is required to allow us to fetch the method names.
169.                             $arr[] = Inflector::humanize($pluginName) . "/" . $file;
170.                         }
171.                     }
172.                 }
173.             }
174.             return $arr;
175.         }
```

Now run the action in your browser, eg. `http://localhost/groups/build_acl`, This will build your ACO table.

You might want to keep this function around as it will add new ACO's for all of the controllers & actions that are in your application any time you run it. It does not remove nodes for actions that no longer exist though. Now that all the heavy lifting is done, we need to set up some permissions, and remove the code that disabled `AuthComponent` earlier.

The original code on this page did not take into account that you might use plugins for your application, and in order for you to have seamless plugin support in your Acl-controlled application, we have updated the above code to automatically include the correct plugins wherever necessary. Note that running this action will place some debug statements at the top of your browser page as to what Plugin/Controller/Action was added to the ACO tree and what was not.

11.2.7 Setting up permissions

Creating permissions much like creating ACO's has no magic solution, nor will I be providing one. To allow ARO's access to ACO's from the shell interface use the AclShell. For more information on how to use it consult the aclShell help which can be accessed by running:

```
cake acl help
```

Note: * needs to be quoted ('*')

In order to allow with the `AclComponent` we would use the following code syntax in a custom method:

```
1.      $this->Acl->allow($aroAlias, $acoAlias);
```

We are going to add in a few allow/deny statements now. Add the following to a temporary function in your `UsersController` and visit the address in your browser to run them (e.g. http://localhost/cake/app/users/initdb). If you do a `SELECT * FROM aros_acos` you should see a whole pile of 1's and -1's. Once you've confirmed your permissions are set, remove the function.

```
1.      function initDB() {
2.          $group =& $this->User->Group;
3.          //Allow admins to everything
4.          $group->id = 1;
5.          $this->Acl->allow($group, 'controllers');
6.
7.          //allow managers to posts and widgets
8.          $group->id = 2;
```

```
9.              $this->Acl->deny($group, 'controllers');
10.             $this->Acl->allow($group, 'controllers/Posts');
11.             $this->Acl->allow($group, 'controllers/Widgets');
12.
13.             //allow users to only add and edit on posts and widgets
14.             $group->id = 3;
15.             $this->Acl->deny($group, 'controllers');
16.             $this->Acl->allow($group, 'controllers/Posts/add');
17.             $this->Acl->allow($group, 'controllers/Posts/edit');
18.             $this->Acl->allow($group, 'controllers/Widgets/add');
19.             $this->Acl->allow($group, 'controllers/Widgets/edit');
20.             //we add an exit to avoid an ugly "missing views" error message
21.             echo "all done";
22.             exit;
23.         }
```

We now have set up some basic access rules. We've allowed administrators to everything. Managers can access everything in posts and widgets. While users can only access add and edit in posts & widgets.

We had to get a reference of a `Group` model and modify its id to be able to specify the ARO we wanted, this is due to how `AclBehavior` works. `AclBehavior` does not set the alias field in the `aros` table so we must use an object reference or an array to reference the ARO we want.

You may have noticed that I deliberately left out index and view from my Acl permissions. We are going to make view and index public actions in `PostsController` and `WidgetsController`. This allows non-authorized users to view these pages, making them public pages. However, at any time you can remove these actions from `AuthComponent::allowedActions` and the permissions for view and edit will revert to those in the Acl.

Now we want to take out the references to `Auth->allowedActions` in your users and groups controllers. Then add the following to your posts and widgets controllers:

```
1.          function beforeFilter() {
```

```
2.          parent::beforeFilter();
3.          $this->Auth->allowedActions = array('index', 'view');
4.      }
```

This removes the 'off switches' we put in earlier on the users and groups controllers, and gives public access on the index and view actions in posts and widgets controllers. In `AppController::beforeFilter()` add the following:

```
1.      $this->Auth->allowedActions = array('display');
```

This makes the 'display' action public. This will keep our PagesController::display() public. This is important as often the default routing has this action as the home page for you application.

11.2.8 Logging in

Our application is now under access control, and any attempt to view non-public pages will redirect you to the login page. However, we will need to create a login view before anyone can login. Add the following to `app/views/users/login.ctp` if you haven't done so already.

```
1.      <h2>Login</h2>
2.      <?php
3.      echo $this->Form->create('User', array('url' => array('controller' => 'users', 'action' =>'login')));
4.      echo $this->Form->input('User.username');
5.      echo $this->Form->input('User.password');
6.      echo $this->Form->end('Login');
7.      ?>
```

If a user is already logged in, redirect him by adding this to your UsersController:

```
1.      function login() {
2.          if ($this->Session->read('Auth.User')) {
3.              $this->Session->setFlash('You are logged in!');
```

```
4.                 $this->redirect('/', null, false);
5.             }
6.         }
```

You may also want to add a flash() for Auth messages to your layout. Copy the default core layout - found at `cake/libs/view/layouts/default.ctp` - to your app layouts folder if you haven't done so already. In `app/views/layouts/default.ctp` add

```
1.         echo $this->Session->flash('auth');
```

You should now be able to login and everything should work auto-magically. When access is denied Auth messages will be displayed if you added the `echo $this->Session->flash('auth')`

11.2.9 Logout

Now onto the logout. Earlier we left this function blank, now is the time to fill it. In `UsersController::logout()` add the following:

```
1.         $this->Session->setFlash('Good-Bye');
2.         $this->redirect($this->Auth->logout());
```

This sets a Session flash message and logs out the User using Auth's logout method. Auth's logout method basically deletes the Auth Session Key and returns a url that can be used in a redirect. If there is other session data that needs to be deleted as well add that code here.

11.2.10 All done

You should now have an application controlled by Auth and Acl. Users permissions are set at the group level, but you can set them by user at the same time. You can also set permissions on a global and per-controller and per-action basis. Furthermore, you have a reusable block of code to easily expand your ACO table as your app grows.

12 Appendices

Appendices contain information regarding the new features introduced in 1.3 and migration path from 1.2 to 1.3

12.1 Migrating from CakePHP 1.2 to 1.3

This guide summarizes many of the changes necessary when migrating from a 1.2 to 1.3 Cake core. Each section contains relevant information for the modifications made to existing methods as well as any methods that have been removed/renamed.

App File Replacements (important)

- webroot/index.php: Must be replaced due to changes in bootstrapping process.
- config/core.php: Additional settings have been put in place which are required for PHP 5.3.
- webroot/test.php: Replace if you want to run unit tests.

 # Removed Constants

The following constants have been removed from CakePHP. If your application depends on them you must define them in `app/config/bootstrap.php`

- `CIPHER_SEED` - It has been replaced with Configure class var `Security.cipherSeed` which should be changed in `app/config/core.php`
- `PEAR`
- `INFLECTIONS`
- `VALID_NOT_EMPTY`
- `VALID_EMAIL`
- `VALID_NUMBER`
- `VALID_YEAR`

 # Configuration and application bootstrapping

Bootstrapping Additional Paths.

In your app/config/bootstrap.php you may have variables like `$pluginPaths` or `$controllerPaths`. There is a new way to add those paths. As of 1.3 RC1 the `$pluginPaths` variables will no longer work. You must use `App::build()` to modify paths.

```
App::build(array(
    'plugins' => array('/full/path/to/plugins/', '/next/full/path/to/plugins/'),
    'models' =>  array('/full/path/to/models/', '/next/full/path/to/models/'),
    'views' => array('/full/path/to/views/', '/next/full/path/to/views/'),
    'controllers' => array('/full/path/to/controllers/', '/next/full/path/to/controllers/'),
    'datasources' => array('/full/path/to/datasources/', '/next/full/path/to/datasources/'),
    'behaviors' => array('/full/path/to/behaviors/', '/next/full/path/to/behaviors/'),
    'components' => array('/full/path/to/components/', '/next/full/path/to/components/'),
    'helpers' => array('/full/path/to/helpers/', '/next/full/path/to/helpers/'),
    'vendors' => array('/full/path/to/vendors/', '/next/full/path/to/vendors/'),
    'shells' => array('/full/path/to/shells/', '/next/full/path/to/shells/'),
    'locales' => array('/full/path/to/locale/', '/next/full/path/to/locale/'),
    'libs' => array('/full/path/to/libs/', '/next/full/path/to/libs/')
));
```

Also changed is the order in which bootstrapping occurs. In the past `app/config/core.php` was loaded **after** `app/config/bootstrap.php`. This caused any `App::import()` in an application bootstrap to be un-cached and considerably slower than a cached include. In 1.3 core.php is loaded and the core cache configs are created **before** bootstrap.php is loaded.

Loading custom inflections

`inflections.php` has been removed, it was an unnecessary file hit, and the related features have been refactored into a method to increase their flexibility. You now use `Inflector::rules()` to load custom inflections.

```
Inflector::rules('singular', array(
    'rules' => array('/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' => '\1ta'),
    'uninflected' => array('singulars'),
    'irregular' => array('spins' => 'spinor')
));
```

Will merge the supplied rules into the infection sets, with the added rules taking precedence over the core rules.

File renames and internal changes

Library Renames

Core libraries of libs/session.php, libs/socket.php, libs/model/schema.php and libs/model/behavior.php have been renamed so that there is a better mapping between filenames and main classes contained within (as well as dealing with some name-spacing issues):

- session.php ⇒ cake_session.php
    - App::import('Core', 'Session') ⇒ App::import('Core', 'CakeSession')
- socket.php ⇒ cake_socket.php
    - App::import('Core', 'Socket') ⇒ App::import('Core', 'CakeSocket')
- schema.php ⇒ cake_schema.php
    - App::import('Model', 'Schema') ⇒ App::import('Model', 'CakeSchema')
- behavior.php ⇒ model_behavior.php
    - App::import('Core', 'Behavior') ⇒ App::import('Core', 'ModelBehavior')

In most cases, the above renaming will not affect userland code.

Inheritance from Object

The following classes no longer extend Object:

- Router
- Set
- Inflector
- Cache
- CacheEngine

If you were using Object methods from these classes, you will need to not use those methods.

Controller & Components

Controller

- `Controller::set()` no longer changes variables from `$var_name` to `$varName`. Variables always appear in the view exactly as you set them.
- `Controller::set('title', $var)` no longer sets `$title_for_layout` when rendering the layout. `$title_for_layout` is still populated by default. But if you want to customize it, use `$this->set('title_for_layout', $var)`.
- `Controller::$pageTitle` has been removed. Use `$this->set('title_for_layout', $var);` instead.
- Controller has two new methods `startupProcess` and `shutdownProcess`. These methods are responsible for handling the controller startup and shutdown processes.

Component

- `Component::triggerCallback` has been added. It is a generic hook into the component callback process. It supplants `Component::startup()`, `Component::shutdown()` and `Component::beforeRender()` as the preferred way to trigger callbacks.

CookieComponent

- `del` is deprecated use `delete`

AclComponent + DbAcl

Node reference checks done with paths are now less greedy and will no longer consume intermediary nodes when doing searches. In the past given the structure:

```
1.        ROOT/
2.            Users/
3.                  Users/
4.                        edit
```

The path `ROOT/Users` would match the last Users node instead of the first. In 1.3, if you were expecting to get the last node you would need to use the path `ROOT/Users/Users`

RequestHandlerComponent

- `getReferrer` is deprecated use `getReferer`

SessionComponent & SessionHelper

- `del` is deprecated use `delete`

`SessionComponent::setFlash()` second param used to be used for setting the layout and accordingly rendered a layout file. This has been modifed to use an element. If you specified custom session flash layouts in your applications you will need to make the following changes.

1. Move the required layout files into app/views/elements
2. Rename the $content_for_layout variable to $message
3. Make sure you have `echo $session->flash();` in your layout

`SessionComponent` and `SessionHelper` are not automatically loaded.

Both `SessionComponent` and `SessionHelper` are no longer automatically included without you asking for them. SessionHelper and SessionComponent now act like every other component and must be declared like any other helper/component. You should update `AppController::$components` and `AppController::$helpers` to include these classes to retain existing behavior.

```
1.      var $components = array('Session', 'Auth', ...);
2.      var $helpers = array('Session', 'Html', 'Form' ...);
```

These change were done to make CakePHP more explicit and declarative in what classes you the application developer want to use. In the past there was no way to avoid loading the Session classes without modifying core files. Which is something we want you to be able to avoid. In addition Session classes were the only magical component and helper. This change helps unify and normalize behavior amongst all classes.

Library Classes

CakeSession

- `del` is deprecated use `delete`

SessionComponent

- `SessionComponent::setFlash()` now uses an *element* instead of a *layout* as its second parameter. Be sure to move any flash layouts from app/views/layouts to app/views/elements and change instances of $content_for_layout to $message.

Folder

- `mkdir` is deprecated use `create`
- `mv` is deprecated use `move`
- `ls` is deprecated use `read`
- `cp` is deprecated use `copy`
- `rm` is deprecated use `delete`

Set

- `isEqual` is deprecated. Use == or ===.

String

- `getInstance` is deprecated, call String methods statically.

Router

`Routing.admin` is deprecated. It provided an inconsistent behavior with other prefix style routes in that it was treated differently. Instead you should use `Routing.prefixes`. Prefix routes in 1.3 do not require additional routes to be declared manually. All prefix routes will be generated automatically. To update simply change your core.php.

```
//from:
Configure::write('Routing.admin', 'admin');
//to:
Configure::write('Routing.prefixes', array('admin'));
```

See the New features guide for more information on using prefix routes. A small change has also been done to routing params. Routed params should now only consist of alphanumeric chars, - and _ or `/[A-Z0-9-_+]+/`.

```
Router::connect('/:$%@#param/:action/*', array(...)); // BAD
Router::connect('/:can/:anybody/:see/:m-3/*', array(...)); //Acceptable
```

For 1.3 the internals of the Router were heavily refactored to increase performance and reduce code clutter. The side effect of this is two seldom used features were removed, as they were problematic and buggy even with the existing code base. First path segments using full regular expressions was removed. You can no longer create routes like

```
Router::connect('/([0-9]+)-p-(.*)/', array('controller' => 'products', 'action' => 'show'));
```

These routes complicated route compilation and impossible to reverse route. If you need routes like this, it is recommended that you use route parameters with capture patterns. Next mid-route greedy star support has been removed. It was previously possible to use a greedy star in the middle of a route.

```
Router::connect(
    '/pages/*/:event',
    array('controller' => 'pages', 'action' => 'display'),
    array('event' => '[a-z0-9_-]+')
);
```

This is no longer supported as mid-route greedy stars behaved erratically, and complicated route compiling. Outside of these two edge-case features and the above changes the router behaves exactly as it did in 1.2

Also, people using the 'id' key in array-form URLs will notice that Router::url() now treats this as a named parameter. If you previously used this approach for passing the ID parameter to actions, you will need to rewrite all your $html->link() and $this->redirect() calls to reflect this change.

```
// old format:
$url = array('controller' => 'posts', 'action' => 'view', 'id' => $id);
// use cases:
Router::url($url);
$html->link($url);
$this->redirect($url);
// 1.2 result:
/posts/view/123
// 1.3 result:
/posts/view/id:123
// correct format:
$url = array('controller' => 'posts', 'action' => 'view', $id);
```

Dispatcher

`Dispatcher` is no longer capable of setting a controller's layout/viewPath with request parameters. Control of these properties should be handled by the Controller, not the Dispatcher. This feature was also undocumented, and untested.

Debugger

- `Debugger::checkSessionKey()` has been renamed to `Debugger::checkSecurityKeys()`
- Calling `Debugger::output("text")` no longer works. Use `Debugger::output("txt")`.

Object

- `Object::$_log` has been removed. `CakeLog::write` is now called statically. See New Logging features for more information on changes made to logging.

Sanitize

- `Sanitize::html()` now actually always returns escaped strings. In the past using the `$remove` parameter would skip entity encoding, returning possibly dangerous content.
- `Sanitize::clean()` now has a `remove_html` option. This will trigger the `strip_tags` feature of `Sanitize::html()`, and must be used in conjunction with the `encode` parameter.

Configure and App

- Configure::listObjects() replaced by App::objects()
- Configure::corePaths() replaced by App::core()
- Configure::buildPaths() replaced by App::build()
- Configure no longer manages paths.
- Configure::write('modelPaths', array...) replaced by App::build(array('models' => array...))
- Configure::read('modelPaths') replaced by App::path('models')
- There is no longer a debug = 3. The controller dumps generated by this setting often caused memory consumption issues making it an impractical and unusable setting. The `$cakeDebug` variable has also been removed from `View::renderLayout` You should remove this variable reference to avoid errors.
- `Configure::load()` can now load configuration files from plugins. Use `Configure::load('plugin.file');` to load configuration files from plugins. Any configuration files in your application that use `.` in the name should be updated to use `_`

Cache

In addition to being able to load CacheEngines from app/libs or plugins, Cache underwent some refactoring for CakePHP1.3. These refactorings focused around reducing the number and frequency of method calls. The end result was a significant performance improvement with only a few minor API changes which are detailed below.

The changes in Cache removed the singletons used for each Engine type, and instead an engine instance is made for each unique key created with `Cache::config()`. Since engines are not singletons anymore, `Cache::engine()` was not needed and was removed. In addition

`Cache::isInitialized()` now checks cache *configuration names*, not cache *engine names*. You can still use `Cache::set()` or `Cache::engine()` to modify cache configurations. Also checkout the New features guide for more information on the additional methods added to `Cache`.

It should be noted that using an app/libs or plugin cache engine for the default cache config can cause performance issues as the import that loads these classes will always be uncached. It is recommended that you either use one of the core cache engines for your `default` configuration, or manually include the cache engine class before configuring it. Furthermore any non-core cache engine configurations should be done in `app/config/bootstrap.php` for the same reasons detailed above.

Model Databases and Datasources

Model

- `Model::del()` and `Model::remove()` have been removed in favor of `Model::delete()`, which is now the canonical delete method.
- `Model::findAll`, findCount, findNeighbours, removed.
- Dynamic calling of setTablePrefix() has been removed. tableprefix should be with the `$tablePrefix` property, and any other custom construction behavior should be done in an overridden `Model::__construct()`.
- `DboSource::query()` now throws warnings for un-handled model methods, instead of trying to run them as queries. This means, people starting transactions improperly via the `$this->Model->begin()` syntax will need to update their code so that it accesses the model's DataSource object directly.
- Missing validation methods will now trigger errors in development mode.
- Missing behaviors will now trigger a cakeError.
- `Model::find(first)` will no longer use the id property for default conditions if no conditions are supplied and id is not empty. Instead no conditions will be used
- For Model::saveAll() the default value for option 'validate' is now 'first' instead of true

Datasources

- DataSource::exists() has been refactored to be more consistent with non-database backed datasources. Previously, if you set `var $useTable = false; var $useDbConfig = 'custom';`, it was impossible for `Model::exists()` to return anything but false. This prevented custom datasources from using `create()` or `update()` correctly without some ugly hacks. If you have custom datasources that implement `create()`,

`update()`, and `read()` (since `Model::exists()` will make a call to `Model::find('count')`, which is passed to `DataSource::read()`), make sure to re-run your unit tests on 1.3.

Databases

Most database configurations no longer support the 'connect' key (which has been deprecated since pre-1.2). Instead, set `'persistent' => true` or false to determine whether or not a persistent database connection should be used

SQL log dumping

A commonly asked question is how can one disable or remove the SQL log dump at the bottom of the page?. In previous versions the HTML SQL log generation was buried inside DboSource. For 1.3 there is a new core element called `sql_dump`. `DboSource` no longer automatically outputs SQL logs. If you want to output SQL logs in 1.3, do the following:

```
1.  <?php echo $this->element('sql_dump'); ?>
```

You can place this element anywhere in your layout or view. The `sql_dump` element will only generate output when `Configure::read('debug')` is equal to 2. You can of course customize or override this element in your app by creating `app/views/elements/sql_dump.ctp`.

View and Helpers

View

- `View::renderElement` removed. Use `View::element()` instead.
- Automagic support for `.thtml` view file extension has been removed either declare `$this->ext = '.thtml';` in your controllers, or rename your views to use `.ctp`
- `View::set('title', $var)` no longer sets `$title_for_layout` when rendering the layout. `$title_for_layout` is still populated by default. But if you want to customize it, use `$this->set('title_for_layout', $var)`.
- `View::$pageTitle` has been removed. Use `$this->set('title_for_layout', $var);` instead.

- The `$cakeDebug` layout variable associated with debug = 3 has been removed. Remove it from your layouts as it will cause errors. Also see the notes related to SQL log dumping and Configure for more information.

All core helpers no longer use `Helper::output()`. The method was inconsistently used and caused output issues with many of FormHelper's methods. If you previously overrode `AppHelper::output()` to force helpers to auto-echo you will need to update your view files to manually echo helper output.

TextHelper

- `TextHelper::trim()` is deprecated, used `truncate()` instead.
- `TextHelper::highlight()` no longer has:
- an `$highlighter` parameter. Use `$options['format']` instead.
- an `$considerHtml`parameter. Use `$options['html']` instead.
- `TextHelper::truncate()` no longer has:
- an `$ending` parameter. Use `$options['ending']` instead.
- an `$exact` parameter. Use `$options['exact']` instead.
- an `$considerHtml`parameter. Use `$options['html']` instead.

PaginatorHelper

PaginatorHelper has had a number of enhancements applied to make styling easier. `prev()`, `next()`, `first()` and `last()`

The disabled state of these methods now defaults to `<span>` tags instead of `<div>` tags.

passedArgs are now auto merged with url options in paginator.

`sort()`, `prev()`, `next()` now add additional class names to the generated html. `prev()` adds a class of prev. `next()` adds a class of next. `sort()` will add the direction currently being sorted, either asc or desc.

FormHelper

- `FormHelper::dateTime()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::year()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::month()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::day()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::minute()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::meridian()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- `FormHelper::select()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
- Default urls generated by form helper no longer contain 'id' parameter. This makes default urls more consistent with documented userland routes. Also enables reverse routing to work in a more intuitive fashion with default FormHelper urls.
- `FormHelper::submit()` Can now create other types of inputs other than type=submit. Use the type option to control the type of input generated.
- `FormHelper::button()` Now creates `<button>` elements instead of reset or clear inputs. If you want to generate those types of inputs use `FormHelper::submit()` with a `'type' => 'reset'` option for example.
- `FormHelper::secure()` and `FormHelper::create()` no longer create hidden fieldset elements. Instead they create hidden div elements. This improves validation with HTML4.

Also be sure to check the Form helper improvements for additional changes and new features in the FormHelper.

HtmlHelper

- `HtmlHelper::meta()` no longer has an `$inline` parameter. It has been merged with the `$options` array.
- `HtmlHelper::link()` no longer has an `$escapeTitle` parameter. Use `$options['escape']` instead.
- `HtmlHelper::para()` no longer has an `$escape` parameter. Use `$options['escape']` instead.
- `HtmlHelper::div()` no longer has an `$escape` parameter. Use `$options['escape']` instead.
- `HtmlHelper::tag()` no longer has an `$escape` parameter. Use `$options['escape']` instead.
- `HtmlHelper::css()` no longer has an `$inline` parameter. Use `$options['inline']` instead.

SessionHelper

- `flash()` no longer auto echos. You must add an `echo $session->flash();` to your session->flash() calls. flash() was the only helper method that auto outputted, and was changed to create consistency in helper methods.

CacheHelper

CacheHelper's interactions with `Controller::$cacheAction` has changed slightly. In the past if you used an array for `$cacheAction` you were required to use the routed url as the keys, this caused caching to break whenever routes were changed. You also could set different cache durations for different passed argument values, but not different named parameters or query string parameters. Both of these limitations/inconsistencies have been removed. You now use the controller's action names as the keys for `$cacheAction`. This makes configuring `$cacheAction` easier as its no longer coupled to the routing, and allows cacheAction to work with all custom routing. If you need to have custom cache durations for specific argument sets you will need to detect and update cacheAction in your controller.

TimeHelper

TimeHelper has been refactored to make it more i18n friendly. Internally almost all calls to date() have been replaced by strftime(). The new method TimeHelper::i18nFormat() has been added and will take localization data from a LC_TIME locale definition file in app/locale following the POSIX standard. These are the changes made in the TimeHelper API:

- TimeHelper::format() can now take a time string as first parameter and a format string as the second one, the format must be using the strftime() style. When called with this parameter order it will try to automatically convert the date format into the preferred one for the current locale. It will also take parameters as in 1.2.x version to be backwards compatible, but in this case format string must be compatible with date().
- TimeHelper::i18nFormat() has been added

Deprecated Helpers

Both the JavascriptHelper and the AjaxHelper are deprecated, and the JsHelper + HtmlHelper should be used in their place.

You should replace

- `$javascript->link()` with `$html->script()`

- `$javascript->codeBlock()` with `$html->scriptBlock()` or `$html->scriptStart()` and `$html->scriptEnd()` depending on your usage.

Console and shells

Shell

`Shell::getAdmin()` has been moved up to `ProjectTask::getAdmin()`

Schema shell

- `cake schema run create` has been renamed to `cake schema create`
- `cake schema run update` has been renamed to `cake schema update`

Console Error Handling

The shell dispatcher has been modified to exit with a `1` status code if the method called on the shell explicitly returns `false`. Returning anything else results in a `0` status code. Before the value returned from the method was used directly as the status code for exiting the shell.

Shell methods which are returning `1` to indicate an error should be updated to return `false` instead.

`Shell::error()` has been modified to exit with status code 1 after printing the error message which now uses a slightly different formatting.

```
1.      $this->error('Invalid Foo', 'Please provide bar.');
2.      // outputs:
3.      Error: Invalid Foo
4.      Please provide bar.
5.      // exits with status code 1
```

`ShellDispatcher::stderr()` has been modified to not prepend Error: to the message anymore. It's signature is now similar to `Shell::stdout()`.

ShellDispatcher::shiftArgs()

The method has been modified to return the shifted argument. Before if no arguments were available the method was returning false, it now returns null. Before if arguments were available the method was returning true, it now returns the shifted argument instead.

Vendors, Test Suite & schema

vendors/css, vendors/js, and vendors/img

Support for these three directories, both in `app/vendors` as well as `plugin/vendors` has been removed. They have been replaced with plugin and theme webroot directories.

Test Suite and Unit Tests

Group tests should now extend TestSuite instead of the deprecated GroupTest class. If your Group tests do not run, you will need to update the base class.

Vendor, plugin and theme assets

Vendor asset serving has been removed in 1.3 in favour of plugin and theme webroot directories.

Schema files used with the SchemaShell have been moved to `app/config/schema` instead of `app/config/sql` Although `config/sql` will continue to work in 1.3, it will not in future versions, it is recommend that the new path is used.

12.2 New features in CakePHP 1.3

CakePHP 1.3 introduced a number of new features. This guide attempts to summarize those changes and point to expanded documentation where necessary.

Components

SecurityComponent

The various requireXX methods like `requireGet` and `requirePost` now accept a single array as their argument as well as a collection of string names.

```
1.      $this->Security->requirePost(array('edit', 'update'));
```

Component settings

Component settings for all core components can now be set from the `$components` array. Much like behaviors, you can declare settings for components when you declare the component.

```
1.      var $components = array(
2.          'Cookie' => array(
3.              'name' => 'MyCookie'
4.          ),
5.          'Auth' => array(
6.              'userModel' => 'MyUser',
7.              'loginAction' => array('controller' => 'users', 'action' => 'login')
8.          )
9.      );
```

This should reduce clutter in your Controller's `beforeFilter()` methods.

EmailComponent

- You can now retrieve the rendered contents of sent Email messages, by reading `$this->Email->htmlMessage` and `$this->Email->textMessage`. These properties will contain the rendered email content matching their name.
- Many of EmailComponent's private methods have been made protected for easier extension.
- EmailComponent::$to can now be an array. Allowing easier setting of multiple recipients, and consistency with other properties.
- `EmailComponent::$messageId` has been added, it allows control over the Message-ID header for email messages.

 # View & Helpers

Helpers can now be addressed at `$this->Helper->func()` in addition to `$helper->func()`. This allows view variables and helpers to share names and not create collisions.

New JsHelper and new features in HtmlHelper

See JsHelper documentation for more information

Pagination Helper

Pagination helper provides additional css classes for styling and you can set the default sort() direction. `PaginatorHelper::next()` and `PaginatorHelper::prev()` now generate span tags by default, instead of divs.

Helper

`Helper::assetTimestamp()` has been added. It will add timestamps to any asset under WWW_ROOT. It works with `Configure::read('Asset.timestamp');` just as before, but the functionality used in Html and Javascript helpers has been made available to all helpers. Assuming `Asset.timestamp == force`

```
1.      $path = 'css/cake.generic.css'
2.      $stamped = $this->Html->assetTimestamp($path);
3.      //$stamped contains 'css/cake.generic.css?5632934892'
```

The appended timestamp contains the last modification time of the file. Since this method is defined in `Helper` it is available to all subclasses.

TextHelper

highlight() now accepts an array of words to highlight.

NumberHelper

A new method `addFormat()` has been added. This method allows you to set currency parameter sets, so you don't have to retype them.

```
1.      $this->Number->addFormat('NOK', array('before' => 'Kr. '));
2.      $formatted = $this->Number->currency(1000, 'NOK');
```

FormHelper

The form helper has had a number of improvements and API modifications, see Form Helper improvements for more information.

 # Logging

Logging and `CakeLog` have been enhanced considerably, both in features and flexibility. See New Logging features for more information.

 # Caching

Cache engines have been made more flexible in 1.3. You can now provide custom `Cache` adapters in `app/libs` as well as in plugins using `$plugin/libs`. App/plugin cache engines can also override the core engines. Cache adapters must be in a cache directory. If you had a cache engine named `MyCustomCacheEngine` it would be placed in either `app/libs/cache/my_custom_cache.php` as an app/libs. Or in `$plugin/libs/cache/my_custom_cache.php` as part of a plugin. Cache configs from plugins need to use the plugin dot syntax.

```
1.      Cache::config('custom', array(
2.          'engine' => 'CachePack.MyCustomCache',
3.          ...
4.      ));
```

App and Plugin cache engines should be configured in `app/bootstrap.php`. If you try to configure them in core.php they will not work correctly.

New Cache methods

Cache has a few new methods for 1.3 which make introspection and testing teardown easier.

- `Cache::configured()` returns an array of configured Cache engine keys.
- `Cache::drop($config)` drops a configured Cache engine. Once dropped cache engines are no longer readable or writeable.

- `Cache::increment()` Perform an atomic increment on a numeric value. This is not implemented in FileEngine.
- `Cache::decrement()` Perform an atomic decrement on a numeric value. This is not implemented in FileEngine.

Models, Behaviors and Datasource

App::import(), datasources & datasources from plugins

Datasources can now be included loaded with `App::import()` and be included in plugins! To include a datasource in your plugin you put it in `my_plugin/models/datasources/your_datasource.php`. To import a Datasource from a plugin use `App::import('Datasource', 'MyPlugin.YourDatasource');`

Using plugin datasources in your database.php

You can use plugin datasources by setting the datasource key with the plugin name. For example if you had a WebservicePack plugin with a LastFm datasource (plugin/webservice_pack/models/datasources/last_fm.php), you could do:

```
1.      var $lastFm = array(
2.          'datasource' => 'WebservicePack.LastFm'
3.          ...
```

Model

- Missing Validation methods now trigger errors, making debugging why validation isn't working easier.
- Models now support virtual fields

Behaviors

Using behaviors that do not exist, now triggers a `cakeError` making missing behaviors easier to find and fix.

CakeSchema

CakeSchema can now locate, read and write schema files to plugins. The SchemaShell also exposes this functionality, see below for changes to SchemaShell. CakeSchema also supports `tableParameters`. Table Parameters are non column specific table information such as collation, charset, comments, and table engine type. Each Dbo implements the tableParameters they support.

tableParameters in MySQL

MySQL supports the greatest number of tableParameters; You can use tableParameters to set a variety of MySQL specific settings.

- `engine` Control the storage engine used for your tables.
- `charset` Control the character set used for tables.
- `encoding` Control the encoding used for tables.

In addition to tableParameters MySQL dbo's implement `fieldParameters`. fieldParameters allow you to control MySQL specific settings per column.

- `charset` Set the character set used for a column
- `encoding` Set the encoding used for a column

See below for examples on how to use table and field parameters in your schema files.

tableParameters in Postgres

....

tableParameters in SQLite

....

Using tableParameters in schema files

You use `tableParameters` just as you would any other key in a schema file. Much like `indexes`:

```
        var $comments => array(
            'id' => array('type' => 'integer', 'null' => false, 'default' => 0, 'key' => 'primary'),
            'post_id' => array('type' => 'integer', 'null' => false, 'default' => 0),
            'comment' => array('type' => 'text'),
            'indexes' => array(
                'PRIMARY' => array('column' => 'id', 'unique' => true),
                'post_id' => array('column' => 'post_id'),
            ),
            'tableParameters' => array(
                'engine' => 'InnoDB',
                'charset' => 'latin1',
                'collate' => 'latin1_general_ci'
            )
        );
```

is an example of a table using `tableParameters` to set some database specific settings. If you use a schema file that contains options and features your database does not implement, those options will be ignored. For example if you imported the above schema to a PostgreSQL server, all of the tableParameters would be ignore as PostgreSQL does not support any of the included options.

Console

Bake

Bake has had a number of significant changes made to it. Those changes are detailed in the bake updates section

Subclassing

The ShellDispatcher has been modified to not require shells and tasks to have *Shell* as their immediate parent anymore.

Output

`Shell::nl()` has been added. It returns a single or multiple linefeed sequences. `Shell::out()`, `err()` and `hr()` now accept a `$newlines` parameter which is passed to `nl()` and allows for controlling how newlines are appended to the output.

`Shell::out()` and `Shell::err()` have been modified, allowing a parameterless usage. This is especially useful if you're often using `$this->out('')` for outputting just a single newline.

Acl Shell

All AclShell commands now take `node` parameters. `node` parameters can be either an alias path like `controllers/Posts/view` or Model.foreign_key ie. `User.1`. You no longer need to know or use the aco/aro id for commands.

The Acl shell `dataSource` switch has been removed. Use the Configure settings instead.

SchemaShell

The Schema shell can now read and write Schema files and SQL dumps to plugins. It expects and will create schema files in `$plugin/config/schema`

....

 # Router and Dispatcher

Router

Generating urls with new style prefixes works exactly the same as admin routing did in 1.2. They use the same syntax and persist/behave in the same way. Assuming you have `Configure::write('Routing.prefixes', array('admin', 'member'));` in your core.php you will be able to do the following from a non-prefixed url:

```
1.      $this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'member' => true));
2.      $this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'admin' => true));
```

Likewise, if you are in a prefixed url and want to go to a non-prefixed url, do the following:

```
1.      $this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'member' => false));
2.      $this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'admin' => false));
```

Route classes

For 1.3 the router has been internally rebuilt, and a new class `CakeRoute` has been created. This class handles the parsing and reverse matching of an individual connected route. Also new in 1.3 is the ability to create and use your own Route classes. You can implement any special routing features that may be needed in application routing classes. Developer route classes must extend `CakeRoute`, if they do not an error will be triggered. Commonly a custom route class will override the `parse()` and/or `match()` methods found in `CakeRoute` to provide custom handling.

Dispatcher

- Accessing filtered asset paths, while having no defined asset filter will create 404 status code responses.

 # Library classes

Inflector

You can now globally customize the default transliteration map used in Inflector::slug using Inflector::rules. eg. `Inflector::rules('transliteration', array('/å/' => 'aa', '/ø/' => 'oe'))`

The Inflector now also internally caches all data passed to it for inflection (except slug method).

Set

Set has a new method `Set::apply()`, which allows you to apply callbacks to the results of `Set::extract` and do so in either a map or reduce fashion.

```
1.      Set::apply('/Movie/rating', $data, 'array_sum');
```

Would return the sum of all Movie ratings in `$data`.

L10N

All languages in the catalog now have a direction key. This can be used to determine/define the text direction of the locale being used.

File

- File now has a copy() method. It copies the file represented by the file instance, to a new location.

Configure

- `Configure::load()` can now load configuration files from plugins. Use `Configure::load('plugin.file');` to load configuration files from plugins. Any configuration files in your application that use `.` in the name should be updated to used _

App/libs

In addition to `app/vendors` a new `app/libs` directory has been added. This directory can also be part of plugins, located at `$plugin/libs`. Libs directories are intended to contain 1st party libraries that do not come from 3rd parties or external vendors. This allows you to separate your organization's internal libraries from vendor libraries. `App::import()` has also been updated to import from libs directories.

```
1.      App::import('Lib', 'ImageManipulation'); //imports app/libs/image_manipulation.php
```

You can also import libs files from plugins

```
1.      App::import('Lib', 'Geocoding.Geocode'); //imports app/plugins/geocoding/libs/geocode.php
```

The remainder of lib importing syntax is identical to vendor files. So if you know how to import vendor files with unique names, you know how to import libs files with unique names.

Configuration

- The default `Security.level` in 1.3 is **medium** instead of **high**

- There is a new configuration value `Security.cipherSeed` this value should be customized to ensure more secure encrypted cookies, and a warning will be generated in development mode when the value matches its default value.

i18n

Now you can use locale definition files for the LC_TIME category to retrieve date and time preferences for a specific language. Just use any POSIX compliant locale definition file and store it at app/locale/*language*/ (do not create a folder for the category LC_TIME, just put the file in there).

For example, if you have access to a machine running debian or ubuntu you can find a french locale file at: /usr/share/i18n/locales/fr_FR. Copy the part corresponding to LC_TIME into app/locale/fr_fr/LC_TIME file. You can then access the time preferences for French language this way:

```
1.      Configure::write('Config.language','fr-fr'); // set the current language
2.      $monthNames = __c('mon',LC_TIME,true); // returns an array with the month names in French
3.      $dateFormat = __c('d_fmt',LC_TIME,true); // return the preferred dates format for France
```

You can read a complete guide of possible values in LC_TIME definition file in this page

 # Miscellaneous

Error Handling

Subclasses of ErrorHandler can more easily implement additional error methods. In the past you would need to override `__construct()` and work around ErrorHandler's desire to convert all error methods into `error404` when debug = 0. In 1.3, error methods that are declared in subclasses are not converted to `error404`. If you want your error methods converted into error404, then you will need to do it manually.

Scaffolding

With the addition of `Routing.prefixes` scaffolding has been updated to allow the scaffolding of any one prefix.

```
1.      Configure::write('Routing.prefixes', array('admin', 'member'));
2.      class PostsController extends AppController {
```

```
3.            var $scaffold = 'member';
4.        }
```

Would use scaffolding for member prefixed urls.

Validation

After 1.2 was released, there were numerous requests to add additional localizations to the `phone()` and `postal()` methods. Instead of trying to add every locale to Validation itself, which would result in large bloated ugly methods, and still not afford the flexibility needed for all cases, an alternate path was taken. In 1.3, `phone()` and `postal()` will pass off any country prefix it does not know how to handle to another class with the appropriate name. For example if you lived in the Netherlands you would create a class like

```
1.        class NlValidation {
2.            function phone($check) {
3.                ...
4.            }
5.            function postal($check) {
6.                ...
7.            }
8.        }
```

This file could be placed anywhere in your application, but must be imported before attempting to use it. In your model validation you could use your NlValidation class by doing the following.

```
1.        var $validate = array(
2.            'phone_no' => array('rule' => array('phone', null, 'nl')),
3.            'postal_code' => array('rule' => array('postal', null, 'nl'))
4.        );
```

When your model data is validated, Validation will see that it cannot handle the 'nl' locale and will attempt to delegate out to `NlValidation::postal()` and the return of that method will be used as the pass/fail for the validation. This approach allows you to create classes that handle a subset or group of locales, something that a large switch would not have. The usage of the individual validation methods has not changed, the ability to pass off to another validator has been added.

IP Address Validation

Validation of IP Addresses has been extended to allow strict validation of a specific IP Version. It will also make use of PHP native validation mechanisms if available.

```
1.  Validation::ip($someAddress);          // Validates both IPv4 and IPv6
2.  Validation::ip($someAddress, 'IPv4'); // Validates IPv4 Addresses only
3.  Validation::ip($someAddress, 'IPv6'); // Validates IPv6 Addresses only
```

Validation::uuid()

A uuid() pattern validation has been added to the `Validation` class. It will check that a given string matches a uuid by pattern only. It does not ensure uniqueness of the given uuid.