Practical CakePHP Projects

Copyright © 2009 by Kai Chan and John Omokore with Richard K. Miller

ISBN-13 (pbk): 978-1-4302-1578-3

ISBN-13 (electronic): 978-1-4302-1579-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: David Golding
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Candace English
Compositor: Patrick Cunningham
Proofreader: Martha Whitt
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Cake Fundamentals

Using a framework of some sort has now become the order of the day for building large-scale web applications. Organizations have found that using an in-house framework for web projects enhances code reuse, scalability, quick project turnarounds, and security.

New and evolving frameworks provide rapid application development tools to promote the adoption of particular programming languages. Many frameworks derived from PHP have been popular with programmers in the open source community. CakePHP—Cake for short—is currently one of the fastest-growing rapid application development frameworks. When you are developing large web applications or creating components that you will reuse in many applications, you'll find Cake to be a great help.

In this chapter, we'll highlight some of the concepts, technologies, and tools that Cake relies on, including the PHP scripting language, the Model-View-Controller design pattern, and object-oriented programming techniques. We will also outline the default folder structures and naming conventions and introduce some Cake best practices. And, of course, we'll demonstrate how to write some Cake code.

This chapter will serve as a quick reference that will provide you with a solid foundation on which to build your knowledge of the framework throughout the rest of the book.

Cake Features

Why should you use Cake when there are so many other frameworks in town? There is a number of good reasons for the popularity of Cake PHP. It has a short learning curve in comparison to other frameworks, because Cake is easy to use and understand. Also, because there are so many PHP programmers, Cake has a large community. New users can find many projects to refer to and use.

Here are some features of Cake that make web application development with it easy and fast:

- It uses the Model-View-Controller (MVC) framework for PHP.
- Its database connectivity support includes MySQL and PostgreSQL, as well as many other database platforms.
- Cake is easy to install on most platforms, including Unix and Windows.
- Its MIT license is more flexible than other licenses.
- It uses easy and flexible templating (which allows PHP syntax, with helpers).

- Cake has view helpers to assist in the insertion of often-repeated snippets of HTML and forms code, Ajax, JavaScript, and so on.
- It has components for handling e-mail, authentication, access control, localization, security, sessions, and request handling.
- Cake provides utility classes to manipulate resources such as sets, files, folders, XML, and many others.
- Your URLs are optimized for search engines.

■**Note** For a complete and up-to-date list of Cake features; see the official web site at `http://cakephp.org`. You can also find many discussions regarding how Cake compares with other frameworks, such as Ruby on Rails, symfony, Zend Framework, and CodeIgniter. For a comparison of Cake with the aforementioned frameworks, check `http://frinity.blogspot.com/2008/06/why-choose-cakephp-over-other.html`.

The Ingredients of Cake

In this section, we'll delve into the core concepts and technologies employed by Cake, starting with the MVC design pattern.

The Model-View-Controller Design Pattern

Cake supports the MVC design pattern, which aims to modularize an application into three parts:

- The *model* represents the data for the application.
- The *view* represents the presentation.
- The *controller* ties the model and view together and deals with user input.

Familiarity with the MVC pattern is a plus, but this book does not assume you have any prior knowledge of MVC. This chapter covers how Cake employs the MVC concept.

Rapid Application Development

Along with MVC, Cake took on the philosophy of rapid application development (RAD), sometimes also known as rapid prototyping. RAD is basically a method of decreasing the time taken to design software systems by using many prebuilt skeleton structures. This provides developers with many advantages, including easier maintenance, code reuse, more efficient teamwork, and quick project turnaround. RAD also provides the ability to make rapid changes based on client feedback, decreasing the dangers of feature creep.

Additionally, you can find a lot of off-the-shelf open source code, which you can easily plug into your Cake applications. A great place to start is `http://bakery.cakephp.org`.

PHP 4+

PHP 4+ refers to PHP version 4 and above. PHP has become one of the most important server-side scripting languages on the Web. It is currently a predominant language for the development of web applications. It provides web developers the functionalities to quickly create dynamic web applications. PHP has come a long way since PHP 3 was first introduced more than a decade ago.

The adoption of the Cake framework assumes knowledge of PHP 4. The official PHP manual, at `http://www.php.net`, provides a complete reference on PHP.

■ **Note** A common problem faced in life with a new adventure is where to go for the right information in order to avoid the mistakes of predecessors. If you are just starting out with PHP, you can refer to the many online PHP forums and repositories, such as the popular PEAR library and the ever-growing `http://www.phpclasses.org` web site.

Object-Oriented Programming

Object-oriented programming (OOP) can be described as a method of implementation in which the parts of a program are organized as a collection of objects, each of which represents an instance of a class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. For example, a `Dog` object `says() 'woof woof'`, while a `Cat` object `says() 'meow meow'`, and they both inherit `says()` from the `Pets` class.

The Cake framework supports the three key principles of object-oriented development: encapsulation, inheritance, and polymorphism.

For the simple magic called *encapsulation*, Cake's implementation of one object is protected, or hidden away, from another object to eliminate interference. However, there must be some interaction with other objects in the application, or the object is useless. As in most OOP applications, an object in the Cake framework provides an *interface* to another object to enable this interaction. Listing 1-1 shows the default database configuration class, called `DATABASE_CONFIG`, which encapsulates `$default` and `$shop` database connection arrays.

Listing 1-1. *The Cake Database Configuration Class*

```
class DATABASE_CONFIG {
    var $default = array(
            'driver' => 'mysql',
            'persistent' => 'true',
            'host' => 'localhost',
            'login' => 'admin',
            'password' => 'superadmin',
            'database' => 'userdb',
            'prefix' => ''
    var $shop = array(
            'driver' => 'mysql',
            'persistent' => 'true',
            'host' => 'localhost',
            'login' => 'user',
            'password' => 'userme',
            'database' => 'shopdb',
            'prefix' => 'sp'
);
```

By default, Cake internally interfaces with the `$default` connection database. It uses its array parameters for its default database connection unless you explicitly specify a different database connection by assigning the `$useDbConfig = '$shop'` property in a model class. This explicit interface will enable some interaction with the tables in the `shop` database.

Cake's support for *inheritance* cannot be overemphasized. It wraps a lot of database manipulation and other utility functions in its default classes in a manner that enables an object to take on the functions of another object and extend or tailor those functions so you don't repeat the same code. We consider this act of charity as one of the greatest benefits to developers, as it undoubtedly ensures fast application development. Therefore, you need to spend some time sharpening your knives by reading a Cake cheat map or its online API (`http://api.cakephp.org`) to understand what your objects will inherit.

In a controller genealogy, user-defined controller objects inherit from the `AppController` object. The `AppController` inherits from `Controller` object, which extends the `Object` class. A controller class can be derived from the `AppController` class, as shown in Listing 1-2. (Controllers are discussed in more detail in the upcoming sections about Cake models, views, and controllers.)

Listing 1-2. *The Application Controller Class*

```
class ProductsController extends AppController {
    function beforeFilter() {
    }
}
```

This default class contains the `beforeFilter()` method, which can be overridden in any class that extends the `AppController` class, such as a user-defined controller class. In Listing 1-2, `ProductsController` extends the `AppController` class.

And lastly, Cake implements *polymorphism* and ensures that functions within an object can behave differently depending on the input. It basically creates the ability to respond to the same function call in many different ways.

The Cake framework creates many reusable objects. You can use these objects without knowing their internal workings. This is one of the key benefits of using Cake.

■Note For more information about OOP in relation to PHP, refer to the PHP manual at `http://www.php.net/oop5`.

Dissecting Cake

Before you start baking a Cake application, you will need to download the Cake framework from `cakephp.org` and install it on your computer. Remember that Cake is based on the PHP scripting language, so you need to have PHP up and running first. If you will be using information stored in a database, you will need to install the database engine. All our examples assume the MySQL database.

Cake's Directory Structure

When you unpack Cake, you will find the following main folder structures:

- `app`: Contains files and folders for your application. The `app` folder is your development folder, where your application-specific folders and files reside.
- `cake`: Contains core Cake libraries. The `cake` folder contains the core libraries for CakePHP. You should not touch these libraries unless you really know what you are doing.
- `docs`: Contains Cake document files such as the read me, copyright, and change log text files. You can store your own documentation in this folder.
- `vendors`: Contains third-party code. The `vendors` folder can contain third-party libraries, such as the Swift Mailer package for sending e-mail messages.

Separating the default Cake core library folder from the application folder makes it possible for you to have many different applications sharing a single Cake installation. With this folder structure, you can easily upgrade your existing version of Cake without affecting any applications you have written. Table 1-1 details Cake's default folder structure.

Table 1-1. *The Cake Default Folder Structure*

Directory			Description
`app/`			The parent folder for your application
	`config/`		Contains configuration files for global structures such database connections, security, and access control
	`controllers/`		Contains your application controllers files (e.g., `user_controller.php`)
		`components/`	Contains your user-defined component files
	`/index.php`		Allows you to deploy Cake with `/app` as the `DocumentRoot`
	`locale/`		Contains locale files that deal with internationalization
	`models/`		Contains the model files
	`plugins/`		Contains the plugin files
	`tests/`		Contains the test folders and files
	`tmp/`		Used for caches and logs
	`vendors/`		Contains third-party libraries

Continued

Table 1-1. *Continued*

Directory			Description
	`views/`		Contains view folders and files for presentation (e.g., the `.ctp` files)
		`elements/`	Elements, which are bits of views, go here
		`errors/`	Custom error pages
		`helpers/`	Helpers
		`layouts/`	Application layout files
	`webroot/`		The `DocumentRoot` for the application
		`css/`	Contains the application style sheet files
		`files/`	Contains any files
		`img/`	Contains graphics
		`js/`	Contains JavaScript files
`cake/`			Contains Cake core libraries
`vendors/`			Contains third-party libraries for all applications

The Cake Naming Conventions

Like similar frameworks, Cake employs naming conventions instead of configuration files for many of its workings, such as for its MVC structure. It is good practice to understand and employ the Cake conventions. You can override some of the rules later, when you become a proficient Cake baker.

Cake has naming conventions for the four core objects: controllers, models, views, and tables. It also provides global constants and functions.

Controller Naming

Controller class names must be plural and must have `Controller` appended, as in `ProductsController`. If the object has more than one word, the second word must also begin with an uppercase letter, as in `OnlineProductsController`. Do not use underscores to separate words.

File names must be plural, with `_controller` appended and the `.php` extension, as in `products_controller.php`. If the object has more than one word, the subsequent words must be delimited with underscores, as in `online_products_controller.php`.

Model Naming

Model class names are singular, as in `Product`. If the object has more than one word, the second word must also begin with an uppercase letter (camel case), as in `OnlineProduct`.

File names are singular, with the `.php` extension, as in `product.php`. If the object has more than one word, the subsequent words are delimited with underscores, as in `online_product.php`.

View Naming

View file names take on the action name in the controller. For example, if the object has a method `ProductsController::upgrade()`, the path is `app/views/products/upgrade.ctp`.

Table Naming

Database table names should be plural, with words delimited with underscores, as in `country_codes`. You can override this naming convention by setting the `$useTable` property to your preferred table name. For example, you could set the following:

```
$useTable = 'mytable';
```

where `'mytable'` is the name of a table in a database.

Global Constants

The global constants are categorized into three major parts:

- *Core defines*: For example, `CAKE _SESSION_STRING` defines the Cake application session value.
- *Web root configurable paths*: For example, `WEBROOT_DIR` defines the web root folder where resources such as image and CSS files are stored.
- *Paths*: For example, `VIEWS` defines the parent folder for the presentation files (views).

For example, you can have the following code snippet to read the `example.ctp` page as an array from the `VIEWS` folder:

```
$page = VIEWS . 'example.ctp';
$file = file($page);
```

Global Functions

The global functions serve as wrappers for some utility functions. For example, the following function code snippet performs a simple search and replace operation to add style to `$text`.

```
$word = "sweet";  // search string - The needle!
$text = "This Cake is as sweet as honey"; // The haystack
// r(...) is the global function
r('$word', "<div class='red'>$word</div>", 'text');
```

The controller should contain most of the business logic, like this:

```
$shopping_basket_balance = $net_price + $tax;
```

Quite often in real applications, the business logic is split into separate parts. However, in Cake, the business logic is often separated into components or vendors, as discussed later in this chapter.

■**Note** It is advisable to familiarize yourself with the global constants and functions to avoid reinventing the wheel. To see a complete list of Cake's various classes and functions, visit `http://api.cakephp.org`.

Models

The model is the first of the MVC concepts. Communicating with data stores such as tables, iCal events, structured files, LDAP records, and so on is an inevitable aspect of any large-scale web application, especially when it involves a large number of users. The actions of manipulating data stored in a data store are best done within a model. The model should be involved with just fetching and saving data from data stores. For example, table queries should be placed in the model.

Model Creation

Models are declared using the keyword `class`, followed by the name you wish to give to the model. Just like any PHP class, a user-defined model may contain some properties and methods specific to the implementation of that model as determined by the business requirement.

A user-defined model class should follow the Cake naming convention and predefined rules; for example, a `Product` model class should extend the `AppModel` class. The `AppModel` class extends the `Model` class, which defines Cake's model functionality. For example, the `Product` model class in Listing 1-3 invariably inherits all of the `Model` class properties and methods.

Listing 1-3. *A Sample Model Class*

```
class Product extends AppModel {
}
```

Though the `Product` class in Listing 1-3 appears empty, it is heavily loaded with some of Cake's properties and methods. We will bring some of these useful properties and methods into the limelight throughout this book.

The `AppModel` class is originally defined in the `cake/` directory. To create your own, place it in `app/app_model.php`. This allows methods to be shared among the models. The `Model` class, which `AppModel` extends, is a standard Cake library defined in `cake/libs/model.php`. Model default methods such as the `find()` method are defined in the `Model` class stored in `cake/libs/model/model.php`. Don't start tweaking Cake's default `Model` class until you become an expert baker.

■**Note** Refer to the cheat sheet in at `http://cakephp.org/files/cakesheet.pdf` before writing a query method in your `Model` class definition. Alternatively, check the Cake API at `http://api.cakephp.org`. This effort will save you from rewriting existing functionalities and enhance rapid application development. For example, Cake provides a `find('all')` query to retrieve some or all information from a database table.

Let's use an example to demonstrate the four well-known types of operations that you will normally perform on a database (collectively known as CRUD):

- Create a record and insert it into a database table.
- Retrieve records from one or more database tables.
- Update tables.
- Delete records.

First, we will create a table named `departments` and insert some sample data with the SQL shown in Listing 1-4.

Listing 1-4. *The Table Schema for departments*

```
CREATE TABLE IF NOT EXISTS `departments` (
 `id` int(10) unsigned NOT NULL auto_increment,
`name` varchar(255) default NULL,
`region` varchar(255) default NULL,
  PRIMARY KEY (`id`)
) ;

INSERT INTO `departments` (`id`, `name`,`region`) VALUES
(1, 'Customer Services','UK'),
(2, 'Sales','UK'),
(3, 'Press Office','UK'),
(4, 'Investor Relations','US'),
(5, 'Human Resources',NG),
(6, 'Partnership Opportunities','US'),
(7, 'Marketing','UK'),
(8, 'Online Marketing','US');
```

Listing 1-4 contains some records about the name and region of the departments. This table is simple and self-explanatory.

Now that we have a database, we'll perform the first of the CRUD operations. We'll create some records, by using the default `save()` method provided in a model class. Using this method comes at the price of ensuring that the format of the data to be passed to it as a parameter must adhere to the Cake preformatted array structure. Let's take a look at a sample data structure in Listing 1-5.

Listing 1-5. *Cake's Expected $this->data Format*

```
Array
(
    [0] => Array
        (
            [Department] => Array
                (
                    [id] => 9
                    [name] => Warranties
                    [region] => Russia
                )

        )

    [1] => Array
        (
            [Department] => Array
                (
                    [id] => 10
                    [name] => Website
                    [region] => UK
                )
        )
)
```

In Listing 1-5, we've preformatted two additional records to be added to our `departments` database table. This structure, stored in a PHP variable such as `$data` or `$this-data`, will save its values to matching fields in the `departments` database table. To commit the data in this structure into this table, the `save()` method is at your service, but the format of `$this->data` argument is crucial to the success of the operation.

Now that we've created the expected data structure, let's define the `Department` model class to use this preformatted data and commit the two additional records into the `departments` table. The `Department` model class is shown in Listing 1-6.

Listing 1-6. *The Department Model Class*

```
<?php
class Department extends AppModel {

    var $name = 'Department';
    var $useTable = 'departments';

    function saveMessage($data) {
        if ($this->save($data)) {
            return true;
        } else {
            return false;
        }
    }
?>
```

In Listing 1-6, first we declare the `Department` class that extends `AppModel`. Next are the properties, starting with the `$name` property assigned the value `Department`. This property is necessary if you are running on anything less than PHP 5. The `$useTable` property specifies the name of the table required for data access or manipulation in the model. Although it isn't required, if omitted, Cake will use a table with the name of the model. For example, if the model name is `department`, Cake will use the `departments` table for the model by default. It is important to explicitly specify which database table you are using, especially if Cake's table naming convention is not followed.

The `saveMessage()` model function call should be done in a controller class. We'll discuss the controller in more detail later in this chapter. In our imaginary controller class, to invoke the `saveMessage()` method defined in Listing 1-6, we need the following statement:

```
$this->Department->saveMessage($data);
```

This method accepts as a parameter the preformatted array information called `$data`, as defined in Listing 1-5. Using an `If` statement, if the `$data` passed to Cake's `save()` model function is committed to the `departments` database table, a Boolean `true` value is returned; if not, `false` is returned. You can also save data into a database table in this manner by using Cake's `create()` model function.

When submitting an HTML form created using the `$form` object in a view, Cake automatically structures the form fields data submitted to a controller in a format that is similar to that shown in Listing 1-5.

Next, let's delve into the retrieve part of CRUD operations, or data access. To be meaningful, most data-access operations are filtered using some criteria. We're going to add a `getDepartment()` method to the `Department` model class, as shown in Listing 1-7.

Listing 1-7. *Retrieving Records Using $region='US' Criteria with the find() Method*

```
function getDepartment($region=null) {

    return $this->find('all',array('conditions'=>array('region'=>$region)));
}
```

In Listing 1-7, we define a `getDepartment()` method that accepts `$region` as its parameter. This method employs the service of the Cake's `find()` method to retrieve some department information based on `$region` as its parameter. To search for a department in the United States, we'll create the following in a controller class:

```
$this->Department->getDepartment('US');
```

This statement will retrieve and format all the departments in the US region, as shown in Listing 1-8.

Listing 1-8. *Structure of the Return Department Data Where Region Equals US*

```
Array
(
    [0] => Array
        (
            [Department] => Array
                (
                    [id] => 4
                    [name] => Investor Relations
                    [region] => US
                )

        )

    [1] => Array
        (
            [Department] => Array
                (
                    [id] => 6
                    [name] => Partnership Opportunities
                    [region] => US
                )
        )
    [2] => Array
        (
            [Department] => Array
                (
                    [id] => 8
                    [name] => Online Marketing
                    [region] => US
                )
        )

)
```

■ **Note** The formatted array data in Listing 1-8 might appear completely different when there are associations between the `Department` model class and other model classes that are connected to database tables. In case of associations, the array will include array data from tables of associated models. You will come across preformatted associated data in Chapter 3.

The find() function is one of the most useful Cake functions for data access. It has the following format:

```
find( conditions[array], fields[array], order[string], recursive[int] )
```

This method also accepts the parameters, in the order listed in Table 1-2.

Table 1-2. *The find() Function Parameters*

Name	Description	Default Value
type	Can be set to all, first, count, neighbors, or list to determine what type of data-access operation to carry out	first
conditions	An array of conditions specified as key and value	null
fields	An array of fields of key and value to retrieve	null
order	To specify whether to order fields in ascending (ASC) or descending (DESC) order	null (no SQL ORDER BY clause will be used if no order field is specified)
page	To determine the page number	null
limit	To limit the page result	null
offset	The SQL offset value	null
recursive	Whether to include the associated model	1

You can use many other Cake predefined model methods, such as the query() method or the read() method, which returns a list of fields from the database and sets the current model data (Model::$data) with the record found. Or you can create your own user-defined methods to manipulate data specific to the table a model object uses.

Listing 1-9 shows an alternative way of retrieving information about the departments. This listing will return exactly the same records from the departments table as the one shown in Listing 1-8 when used in our imaginary controller. The difference is that Listing 1-8 uses the find() method, while Listing 1-9 uses the query() method to access data.

Listing 1-9. *Retrieving Records with the query() Method*

```
function getDepartment($region=null) {

    $sql = "SELECT * FROM `departments` WHERE `region` = $region";
    return $this->query($sql);
}
```

One advantage of using the query() method is that you can put an already defined SQL statement, from a legacy system, into this method without going through the trouble of dividing the query parameters into parts, as you would need to do to use the find() method.

Data Validation

Data validation is an essential part of ensuring integrity and accuracy of data submitted by the user, such as via a web form. Cake has built-in validation mechanisms. You specify the validation rules in a model, and Cake automatically applies the rules when a web form is connected to that model. These rules can also be applied to XML data.

First, let's add a simple validation rule to our `Department` model using the `$validate` array, as shown in Listing 1-10. The validation rule array is basically an associative array. The keys are the names of the form fields to validate, and the corresponding values represent the rules attached to the form fields. We'll make use of this rule later, in the "Views" section.

Listing 1-10. *The Validation Rule for the Department Model*

```
var $validate = array( 'region' => array(
                                   'alphaNumeric' => array(
                                       'rule'=>'alphaNumeric',
                                       'required'=>true,
                                       'message'=>'Enter a region.'
                                       )
                                   )
                    );
```

A field can have multiple validation rules. The `$validate` array in Listing 1-10 defines a rule for the `region` field in our `Department` model. If a user does not submit a valid `region` field, the model will return an error to the controller and quit committing the data to the `departments` database table. The `message` key deals with the error messages during validation. To display the error message on a form, use the form helper's `error` function:

```
<?php echo $form->error('region');?>
```

This will display the error message "Enter a region" if a user enters a nonalphanumeric value in the `region` input field.

Apart from the rules employed in Listing 1-10, Cake provides a number of built-in validation rules to check the validity of form inputs and ensure the integrity of information you want to store. Table 1-3 lists a few of the built-in rules provided by Cake.

Table 1-3. *Some of Cake's Built-in Validation Rules*

Rule	Description	Example
`cc`	Checks for a valid credit card number	`'rule'=>array('cc','fast')`
`date`	Checks for a valid date	`'rule'=>'date'`
`email`	Checks for a valid e-mail address	`'rule'=>'email'`
`ip`	Checks for a valid IP address	`'rule'=>'ip'`
`phone`	Checks for a valid phone number	`'rule'=>array('phone', null, 'uk')`

For a complete list of the validation constants in your Cake build, see the predefined rules in the `cake/libs/validation.php` file.

You can extend the list of the rules by adding your own user-defined rules. In Listing 1-11, we define a simple custom rule to check if a value is a string.

Listing 1-11. *A Custom Rule Called String*

```
function string($check) {

    return is_string($check);
}
```

Before you apply a custom rule, such as the `string` rule shown in Listing 1-11, add it to the `cake/libs/validation.php` file, and then simply add the rule to your model `$validate` array:

```
var $validate = array('name'=>'string');
```

This will ensure that the `name` field is a valid string. In upcoming chapters, you will come across more validation rules.

The model object is robust and provides a lot of functionality for database manipulation. However, part of the data retrieved by a model is required for web surfers' consumption. When a user makes a URL request, some response is expected to be displayed in a view, which we'll look at in the following section.

Views

Now that we have some validation rules, let's build an HTML form to ask users to enter department information. The task of building a web form is done in a view.

Views are presentation pages. The HTML or XML documents on the Web are views to the users. However, views can be anything, especially if Cake is used to output other formats like RSS, PDF, and so on (which is certainly possible with the `RequestHandler` component and parsing extensions in the router). Views render information to the users. Views are composed of a mixture of HTML and PHP code.

By default, a view should be stored under the controller name folder. For example, the view for the `add()` method in the `DepartmentsController` is stored as `app/views/departments/add.php`.

Data from the controller is passed to the view by using the `set()` method in a controller.

■ **Note** Views should be involved only with displaying output. For example, this is where you will see HTML tags and XML tags. Business logic, such as `$shopping_basket_balance = $net_price + $tax;`, should not be in the view. However, the following is OK in a view: `If ($shopping_basket_balance > 1000 ) { echo 'You are eligible for a discount'; }`.

Let's build the view to add information to our `departments` database table. We are going to use another utility provided by Cake to build forms: the `$form` object. Listing 1-12 shows the add view.

Listing 1-12. *The Add View for the departments Table*

```
<h1>Add Department</h1>
<?=$form->create('department',array('action'=>'add'));?>
<p>Name:
<?=$form->input('Department.name',array('size'=>'120'));?>
<?=$form->error('name');?></p>
<p>Region:
<?=$form->input('Department.region',array('size'=>'40'));?>
<?=$form->error('region');?></p>
<?=$form->end('Save')?>
```

The view code created using the `$form` object is stored in `/app/views/department/add.ctp`. Remember that the validation rule for this form is created at Listing 1-10.

To display the add view to a user, we need a controller object with a function called `add()`, which tries to do exactly what it says: add the form data to our `departments` database table.

Listing 1-13 shows the action `add()` method of the `DepartmentsController`.

Listing 1-13. *The add() Action in the Departments Controller*

```
<?php
class DepartmentsController extends AppController {

   function add() {

        if (!empty($this->data)) {

            $this->Department->create();
            if ($this->Department->save($this->data)) {
                $this->Session->setFlash(➥
__('The Department data has been saved', true));
                $this->redirect(array('action'=>'add'));
            } else {
                $this->Session->setFlash(➥
__('The Department data could not be saved. Please, try again.', true));
            }
        }
    }
?>
```

Now that we've built a web form using the HTML helper and created a controller to handle the `add()` action, let's demonstrate how data is passed to a view. We'll start by creating a show action in our `Departments` controller class. The action method is shown in Listing 1-14.

Listing 1-14. *The show() action in the Departments Controller*

```
function show($region) {

    $this->set('data', $this->Department->getDepartment($region));
}
```

In Listing 1-14, the `show()` method accepts the `$region` data as a parameter, and then retrieves departmental data and uses the `set()` method to prepare the data for the view in Listing 1-15. A view is always named after an action. For example, the `show()` action in the `DepartmentsController` of Listing 1-14 will have a view file stored in `app/views/department/show.ctp`.

Listing 1-15. *A View for the Action show($region) of DepartmentsController*

```
<h1 class="first">

    Our Department: -

    <?php

        foreach($data as $department) {
            echo $department ['Department'] ['name'];
        }
    ?>

</h1>
<p>Our Departments are led by Dr. Cake .</p>
```

The view in Listing 1-15 will display the name of the department (for example, Sales) in the header section of the web page presented to the user.

Tip If you have a number of data items to display in a view, such as `$title = "Practical Cake Projects"; $department = "Sales"; and $region = "UK";`, using `$this->set(compact('title ','department','region'));` will enable you to use only one `set()` function in your controller class to pass all the information to your view. You can then access the individual variable in your view; for example, `<?php pr($title); ?>`.

An essential part of any framework is the part that handles requests. In the MVC structure, this is handled by the controller.

Controllers

As you've seen, a controller is a class with methods called *actions*. These actions or methods contain most of the logic that responds to user requests in an application. For example, if a user wants to know the number of departments in a particular region, the user needs to access the `show()` method of `DepartmentsController` defined in Listing 1-10, by typing the following URL in a browser address bar:

```
localhost/departments/show/US
```

The default structure for accessing a Cake URL is to first specify the controller and then the action. In the preceding URL, `departments` is the controller, `show` is the action, and `US` is a parameter.

By convention, a Cake request should be structured in the following manner:

```
http://[mydomain.com]/[Application]/[Controller]/[Action]/[Param1]/…[ParamN]
```

■ **Note** The `index()`method is the default access point to a controller when a method is not explicitly specified in a user's request. For example, you can load the `index()` method with codes that will invoke the welcome page of your application. However, do not forget to create a view, or you will get a warning from Cake stating that you should create a view for the action.

Your application's controller classes are expected to extend the `AppController` class, which in turn extends a core `Controller` class, which is a standard Cake library. The `AppController` class is defined in `/app/app_controller.php`, and it should contain methods that are shared between two or more controllers.

These controllers can include any number of actions. The `AppController` serves as a global class that can contain properties and methods common to all the user-defined controllers in an application. For example, you can have a method to detect and extract the IP address of a user, and then use the value of this address to determine the flow of the application. Earlier, in Listing 1-2, we used the default controller method called `beforeFilter()` in our controller class to reference the method defined in the `AppController` class stored in `app/app_controller.php`. Another simple example is to set a default page title for an application, as in this example:

```
<?php
class AppController extends Controller {
    var $pageTitle = 'Chapter 1 - A Bakery Application';
```

Since our user-defined controller extends the `AppController`, using the statement `$this->PageTitle` within our controller gives us access to the string `'Chapter 1 - A Bakery Application'` assigned to the `pageTitle` property in the `AppController` class.

The `$uses` property is an important property within the controller. It works similarly to the `require_once` statement in PHP. Basically, once you have created a model (such as `Department`) and you want to use the model in a controller, you need to include it in the `$uses` array. For example, where `Department` and `Trade` are existing models, you can have the following statement in your controller:

```
$uses = array('Department', 'Trade');.
```

Cake Components

Components are classes defined to carry out specific application tasks to support the controller. Cake comes with many built-in components, such as `Acl` (for user and group access control), `Auth` (for user and group authentication), `Email`, `Session`, and `RequestHandler`. Components can also be user-defined. In fact, in large web applications, you will most likely need to build some of your own components to be used by several controllers. All the components that you develop should be stored in the folder `app/controllers/components`. Components follow the same Cake conventions as controllers.

From a programmer's point of view, components enable you to extend the functionality of Cake. If you find that your component is quite useful and you possess the free open source spirit, you can and should post it on the Cake web site, where there is a public repository of components.

To demonstrate, we'll dive straight in and create our own simple component—a utility to convert an array to an object. Listing 1-16 shows the code to create this component.

Listing 1-16. *A Component to Convert an Array to an Object*

```
<?php
class ArrayToObjectComponent extends Object{

    function startup(&$controller) {
        $this->Controller = $controller;
    }

    function convert($array, &$obj) {
        foreach ($array as $key => $value) {
            if (is_array($value)) {
                $obj->$key = new stdClass();
                $this->array_to_obj($value, $obj->$key);
            } else {
                $obj->$key = $value;
            }
        }

        return $obj;
    }
}
?>
```

The `ArrayToObjectComponent` class in Listing 1-16 contains two basic functions, which are stored in `app/controllers/components/array_to_object.php`:

- The first function, called `startup()`, is used to instantiate the controller object. This enables all other functions within the component to access information contained in the parent controller. It's basically a callback method used to bring the controller object into the component.
- The second function, `convert()`, is our user-defined function. It does the work of accepting an array of data and returning the array as an object. You can use this component whenever you want to convert an array to an object.

Everything inside a component should be generic. Do not put controller-specific code, such as a database table name, into components.

You can use components within controllers or other components. To use a component—whether it is a built-in one or one you have created—you need to first declare the component within the `$components` array in a user-defined controller, another component, or in the `AppController` class. For example, to use the component in Listing 1-16, include the following statement:

```
var $components = array('ArrayToObject')
```

In Listing 1-17, we make references to the built-in `Session` component and our `ArrayToObject` component in the `DepartmentsController` class.

Listing 1-17. *Using Components in DepartmentsController*

```
<?php
class DepartmentsController extends AppController
{
    var $uses = array( 'Department');
    var $components = array( 'Session', 'ArrayToObject');

    function display() {
        $arrData = array();
        $arrData  =  $this->find('all');
        pr($this->ArrayToObject->convert(arrData, &$obj));
    }
}
?>
```

In Listing 1-17, we convert the result of the data retrieved from our `departments` database table from an array to an object. First, we use the `$uses` array to reference the `Department` model. We then use the `$component` array to reference the `Session` component, which is a built-in Cake component, and the `ArrayToObject` component, which is our user-defined component. Next, we create a `display()` function that contains a declaration of an array variable called `$arrData`. We retrieve the department data using the default `find()` function, store the result in the array, and then pass the array to the `convert()` method of the `ArrayToObject` component. Finally, we use the Cake `pr()` global function to print the resulting object.

Helpers

Cake helpers are classes that help to decrease development time by providing shortcuts to generate presentational elements. Earlier, we used the Cake form helper, which helps with form element creation and data handling. Helper files should be stored in the `app/views/helpers` folder. Table 1-4 briefly describes some of Cake's built-in helpers. For full documentation of the Cake helpers, check the Cake API at `http://api.cakephp.org`.

Table 1-4. *Some of Cake's Built-in Helpers*

Helper	Description
HTML	Helps automate creating HTML elements and also enables dynamic generation of HTML tags by accepting and parsing variables. This helper is called in a view by using the `$html` object. To include a reference to HTML helper, use the variable `$helpers = array('Html');`. HTML helper functions output HTML elements such as `charset`, `css`, `div`, `docType`, `image`, `link`, `meta`, `nestedList`, `para`, `style`, `tableHeaders`, `tableCells`, and so on.
Form	Helps in form creation and processing. Use the `$form` object together with its functions to create form elements. For example, to create a form input element, use the `$form->input()` function. To start the form tag, use the `$form->create()` function. Other form input element functions include `label`, `checkbox`, `dateTime`, `hidden`, `radio`, `textarea`, and so on. There are many options that can be used in form element functions, such as `maxLength`, to set the maximum length of an HTML attribute.
Ajax	Helps to simplify Ajax tasks. It requires the statement `$javascript->link(array('prototype'))` in the view, which references the Prototype JavaScript framework, in order to work properly in a view. You can get a copy of Prototype from `http://www.prototypejs.org/download`.
JavaScript	Helps to simplify JavaScript tasks, such as to create a JavaScript Object Notation (JSON) object from an array, using `$javascript->object()`. To attach an event to an element, use `$javascript->event()`.
Paginator	Helps to format data into multiple pages or to sort data based on some parameters. For example, to create a link to the next set of paginated results, use the `$paginator->next()` function.
Session	Provides functions to deal with session management. For example, to render messages, use `$session->flash()`. To read all values stored in a given session, use `$session->read()`.
Text	Provides functions to deal with text or string handling. For example, to remove whitespace from the beginning and/or end of text, use the `$text->trim()` function.
Time	Helps manage dates and times. For example, to check if a given date/time string is today, use the `$time->isToday()` function.
XML	Helps with XML manipulation. To create XML elements, use the `$xml->elem()` function. This helper can also be used to convert a result set into XML.

To reference the common helpers that you need in your application, you can specify the following statement in your `AppController` class:

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

This will ensure that the `$javascript->link()` function in a layout works properly.

You may need to create your own helper or tweak an existing helper class to provide additional functionality that is not yet supplied by Cake. As an example, Listing 1-18 creates a simple helper called `/app/views/helpers/break.php`. This helper will print a variable and insert a new break after printing.

Listing 1-18. *A Sample Custom Break Helper*

```
class BreakHelper extends AppHelper {
    function newline($val) {
        return $this->output("$val<br />");
    }
}
```

We can use this helper in our `Departments` controller object to insert a break whenever we use the controller's `print()` function, as shown in Listing 1-19.

Listing 1-19. *Using the Sample Break Helper*

```
<?php
class DepartmentsController extends AppController {
    var $name = 'Departments';
    var $helpers = array('Break');

    function print($val) {
        return $this->Break->newline($val);
    }
}
?>
```

First, in Listing 1-19, we reference the break helper by declaring `var $helpers = array('Break');`. Next, we define a `print()` function that accepts a `$val` parameter. This function contains the statement that invokes the break helper's `newline($val)` method, and consequently returns the result with a newline after it.

Plugins

With Cake, you can create a complete MVC package called a *plugin*, which you can integrate into other Cake applications. A plugin is a mini-application with its own controllers, models, views, and other Cake resources. Cake does not have any built-in plugins. You can use third-party plugins, or better still, build your own.

Here, we will create a basic feedback plugin that will provide mailing facility. It will have the following directory structure:

```
/app
    /plugins
        /feedback
            /controllers
            /models
            /views
            /feedback_app_controller.php
            /feedback_app_model.php
```

where

- `/controllers` contains plugin controllers.
- `/models` contains plugin models.
- `/views` contains plugin views.
- `/feedback_app_controller.php` is the plugin's `AppController`, named after the plugin.
- `/feedback_app_model.php` is the plugin's `AppModel`, named after the plugin.

■**Note** You must create both an `AppController` and an `AppModel` for a plugin to work properly. If you forget to define the `FeedbackAppController` class and the `FeedbackAppModel`, Cake will throw a "Missing Controller" error.

The feedback plugin's `AppController` is stored in `app/plugins/feedback_app_controller.php`, and its corresponding `AppModel` class is stored in `app/plugins/feedback_app_model.php`, as shown in Listing 1-20.

Listing 1-20. *Feedback App Classes for the Feedback Plugin*

```
<?php
class FeedbackAppController extends AppController {
    //..
}
?>

<?php
class FeedbackAppModel extends AppModel {
    / /..
}
?>
```

Now, let's create the FeedbackSendController for our feedback plugin. The code in Listing 1-21 is stored in app/plugins/feedback/controllers/feedback_send_controller.php.

Listing 1-21. *The FeedbackSendController to Invoke the send() Method*

```
<?php
class FeedbackSendController extends FeedbackAppController {
    var $name = 'Feedback';
    var $uses = array( 'Feedback');

    function send($toEmail) {

        $this->set( "result", false );

        if ( $this->Feedback->sendEmail($toEmail) ) {
            $this->set( "result", true );
        }
    }
}
?>
```

Next, we'll create and store the FeedbackSendModel class in the app/plugins/feedback/models/feedback_send_model.php file, as shown in Listing 1-22.

Listing 1-22. *The FeedbackSendModel That Uses the PHP mail() Function to Send a Message*

```
<?php
class FeedbackSendModel extends FeedbackAppModel
{
    function sendEmail($recipient) {
        // Here's where we try to send an email message.
        if( mail($recipient, 'Hi', 'What is baking', 'From: sugar@Cake.com') ) {
            return true;
        }
        return false;
    }
}
?>
```

Next, let's create a simple feedback plugin view stored in the app/plugins/feedback/views/feedback_send/send.ctp file, as shown in Listing 1-23.

Listing 1-23. *The Feedback Email View*

```
<h1>Feedback Email</h1>
<?php
if ( $result ) {
    echo 'Thank you for your feedback.';
} else {
    echo 'Sorry! Our system is down at the moment. Please try again later.';
}
?>
}
?>
```

Now that we have installed the feedback plugin, we can use it. To access the plugin within a Cake application, you can add name of the plugin, then the action, then the parameter to the URL, as follows:

`http://localhost/feedback/feedbackSend/feedbackSend/send/[`*`emailparam`*`]`

You can have a default controller with the name of your plugin. If you do that, you can access it via `/[plugin]/action`. For example, a plugin named `users` with a controller named `UsersController` can be accessed at `http://[your domain]/users/add` if there is no plugin called `AddController` in your `[plugin]/controllers` folder.

Plugins will use the layouts from the `app/views/layouts` folder by default. You will see how to override layouts in Chapter 3.

You can access a plugin within controllers in your Cake application by using the `requestAction()` function:

```
$sent = $this->requestAction(array('controller'=>'FeedbackSend','action'=>'send'));
```

Vendors

Many modern frameworks adopt the Don't Repeat Yourself (DRY) principle. Lazy (or maybe efficient) programmers don't like to reinvent the wheel! To allow us to sleep in a little longer, Cake has provided a `vendors` folder. This is where we store third-party applications that don't have any relationship with Cake, such as the phpBB message board application and the Swift Mailer mailing application. This comes in handy, considering the number of utility scripts and programs available in various PHP repositories such as `http://www.phpclasses.org`.

Cake's technique of including external scripts is as simple as using the following function:

```
 App::import('Vendor','file',array('file'=>'fileName.php'));
```

The third parameter of the function accepts an array of file names. Usually, `fileName.php` is the startup file of the third-party application.

The `App::import()` function can be used in controllers, models, and views of Cake applications. However, it is important that the call is made before any class definition. The `vendors` folder provides a standard way to include third-party applications.

As an example, let's create a `spillout.php` script, as shown in Listing 1-24. This script will serve as our third-party script that we'll use in our `ScreenController` later. This script should be stored in the `app/vendors` folder.

Listing 1-24. *A Script to Display the Content of a File on the Screen*

```
<?php
class ReadFile {
    protected $file;

    public function __construct($fileName) {
        $this->file = $fileName;
    }

    private function spill() {
        return readfile($this->file);
    }
}
?>
```

Listing 1-24 will simply read the content of the `welcome.html` file and send it to the screen for a user's consumption. We can import this script into our `ScreenController` as shown in Listing 1-25.

Listing 1-25. *The ScreenController to Use a Script as a Vendor*

```
<?php
    App::import('Vendor','file',array('file'=> 'spillout.php'));
    class ScreenController extends AppController {

    function index() {
        $output = new ReadFile('welcome.html');
        $output->spill();
    }
}
?>
```

In Listing 1-25, we use the `import()` function to load the content of the `spillout.php` file, and then we create an instance of the `ReadFile` class using the `welcome.html` file as its parameter and store it in `$output`. Finally, we send the content of the file to the screen using `$output->spill();`.

Summary

In this chapter, we briefly introduced you to the main features of Cake. We explained how it is a RAD platform, with the MVC design pattern forming the base foundation. We covered how the Cake MVC structure works, with business logic stored in controllers and components, data access in models, and presentational markup in the view. Additionally, we showed how Cake reduces development time with helpers, plugins, and vendors.

After reading this chapter, you should have an overview of how Cake structures a web application. You should feel confident that learning Cake is one of the best decisions for anyone interested in PHP programming and with a need to write rapid web applications. But do note, sometimes it may be better to write basic methods like “Hello World” in a simple PHP script, rather than using Cake, so that you don’t end up killing an ant with a sledge hammer.

In the following chapters, we’ll present full-fledged Cake applications, beginning with a simple blogging application.