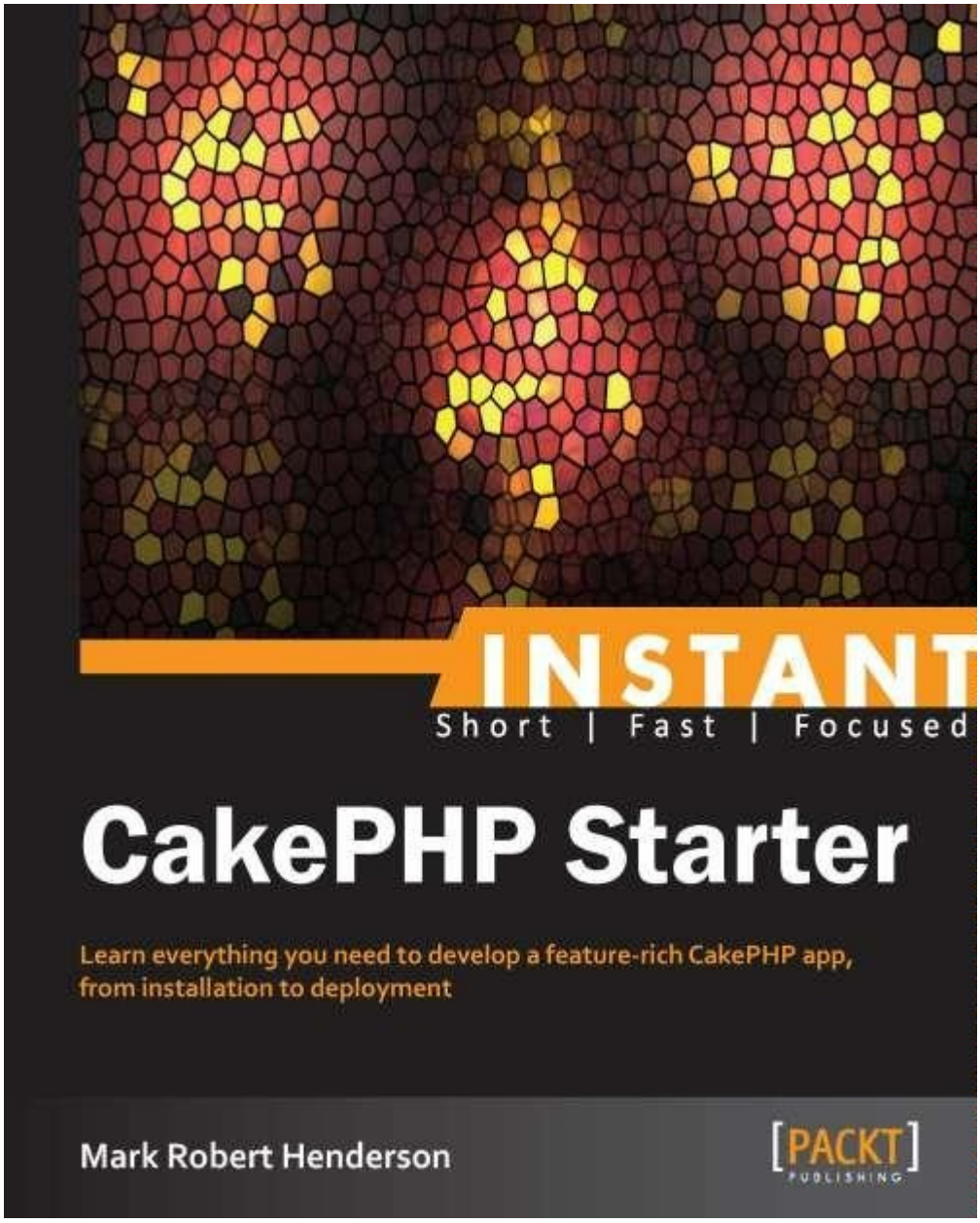

INSTANT
Short | Fast | Focused
CakePHP Starter
Learn everything you need to develop a feature-rich CakePHP app, from installation to deployment
Mark Robert Henderson
[PACKT]
PUBLISHING

Table of Contents

Instant CakePHP Starter

Instant CakePHP Starter

First published: April 2013

Production Reference: 1180413

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-260-5

www.packtpub.com

Credits

Author

Mark Robert Henderson

Reviewers

Jorge M. González Martín

Sherwin D. Robles

Acquisition Editor

James Jones

Commissioning Editors

Maria D'souza

Poonam Jain

Technical Editor

Nitee Shetty

Project Coordinator

Amigya Khurana

Proofreader

Dirk Manuel

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

Cover Image

Aditi Gajjar

About the author

Mark Robert Henderson has been understanding, developing, modifying, and deploying PHP applications since 2006, and while he has been through several love affairs with different technologies over the years, he tries to remain as unopinionated as possible about specific technologies, mostly to avoid arguments with other developers.

However, sometimes a technology comes along that just makes an engineer smile at its simplicity and elegance. CakePHP is one of those, and Mark is happy to share some of the cool things that he's learned in the course of creating applications with it.

I would like to thank Amigya and Maria at Packt Publishing for giving me the opportunity to write this book, and Jesse, Jordyn, Dave, and Sequoia for developing countless CakePHP applications with me between 2007 and 2010.

About the reviewers

Jorge M. González Martín has a degree in Computer Science, as well as extensive experience in different I.T. positions, from development, to consulting, and management. He started using CakePHP many years ago, and has never turned back. He has applied the framework to several different projects and domains, especially the hospitality and tourism sector, in which he's been working for over 7 years. In 2012 he joined the **Cake Development Corporation** (**CakeDC**), the company established by Larry Masters, the founder of CakePHP, as a developer and project manager.

Sherwin D. Robles is a web developer from Philippines. He graduated from Central Colleges of the Philippines with a bachelor's degree in Computer Science and is currently working at Druid Solutions Inc.

His expertise is rooted in research and development endeavors on how to achieve improved levels of dependability from Internet and computing systems.

I would like to thank my family, my girlfriend, and friends for all the love and support they have provided me over the years.

www.packtpub.com

Support files, eBooks, discount offers, and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt Publishing offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt Publishing books and eBooks.

Free Access for Packt Publishing account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant CakePHP Starter

Welcome to *Instant CakePHP Starter*. This book is designed to get a CakePHP app "out the door" in a fast, efficient, and agile manner. We're literally going to go from installation to production deployment in just a few pages.

Strap in!

This book contains the following sections:

So, what is CakePHP? is a technological look at Cake and the MVC architecture.

Installation will soon have you running a Cake app in a production environment. Don't worry; it's not as scary as it sounds.

Quickstart – building a web application uses an iterative approach where we will easily craft our application's core functionality using some fundamental features of Cake, and send our updates to the production server.

Top 6 features you need to know about gets us playing with the full power of our stack, and Cake can become what it was born to be!

People and places you should get to know lets you determine your own level of involvement! This section facilitates this with a peek into the CakePHP community: blogs, Twitter feeds, documentation, and more. In short, it covers the entire CakePHP-centric Internet.

So, what is CakePHP?

CakePHP is like the little buddy that lives on your shoulder and helps you out while you're developing a site. It helps you do it quickly, and organizes your code in a way that allows you and other developers to quickly understand what's going on.

CakePHP – the short story

Here's the definition of CakePHP we'll be using for the purposes of this book:

CakePHP is a **Model-View-Controller** framework implemented in the PHP programming language. Its aim is to take the repetitive tasks out of your way, provide structure to your code by focusing on convention instead of

configuration, and to "harden" your application with an important layer of security.

Why CakePHP – making the short story long

This story is about you, either now or some time in the past. You have some experience of writing HTML/CSS code and maybe a bit of JavaScript. You've marked up some static web pages and now you want to delve into the world of web application development.

You set forth with an idea for an app, and some time later you've pieced it all together. To your delight, it does exactly what you wanted it to do—nothing more, nothing less. You release it to the public and, harnessing the momentum and exhilaration, you decide to take a break from e-mails so you can start working on your next project.

As you begin to write your next app, you find yourself saying "Wait a minute, I already wrote a login system..." but when you look at your login code you find that it's so specific to your previous application that you can barely reuse any of it. Writing that system was fun then, but it is tedious now. Now you just want to get to the meat of your app and not worry about any of this!

Having had the wind sucked out of your sails, you decide to get back to your e-mail. Your heart sinks further—bug reports have started coming in; not just small formatting bugs but "big" bugs involving security. Nobody told you to sanitize your input! And what is a cross-site scripting attack?

You don't have time for this. You want to work on your next app, not spend time fixing this one! You decide to bring another developer on to help, but your code base is so specific to both you and the application that you spend as much time answering his questions and explaining the choices that you made in the code as you would have spent just fixing it yourself.

Finally, the bugs all get fixed, but because there was no testing framework or official definition of the functionality, the bug "fixes" broke other parts of the site, and caused more bugs!

You're not sure what happened, or how this all could have been avoided.

Getting the point?

CakePHP to the rescue!

The benefit of using a framework like CakePHP is that a lot of this work is already done, and a lot of these problems already solved. There are many

frameworks like CakePHP, but CakePHP's focus on rapid development and simplicity puts it in a class of it's own. Also, a strong community is important around any technology, and I can say without hesitation or irony that CakePHP's community is unmatched.

Out of the box, Cake provides:

- User session management (via cookies or sessions)
- Request handling (via GET and POST)
- E-mail support
- A well-tested security layer that includes CSRF protection and data sanitation
- Multiple caching mechanisms
- Integrated CRUD (Create, Retrieve, Update, Delete) for your datastore
- Model validation (required fields, valid e-mail, and so on)
- Data sanitation
- Code generation (both PHP and HTML)
- A unit testing framework complete with code coverage support for both your app and the core itself
- A standard coding convention, project structure, and suggested code standard
- A robust view-helper library that handles AJAX, pagination, forms generation, static file inclusion, and more

In addition to that, there is an active community built around the framework to which you can always go for support and inspiration. Check out http://community.cakephp.org.

Model-View-Controller – what is that all about?

An entire book could be written on this subject alone, so we'll only broach the subject here.

Model-View-Controller is what is known as a design pattern, and is based around separating components of code based on their responsibilities. Models refer to the data layer, views refer to the presentation layer, and controllers contain your application or business logic.

This concept was invented (among many, many other things) in the 70's at Xerox Parc when they were building the first GUIs. It is one of the (if not the) oldest pattern in GUI programming.

For more information about the MVC design pattern, please see the section titled *Top 6 features you need to know about*.

Installation

This section is divided into our three main areas of concern: our development environment, our application, and our deployment environment.

We are going to intentionally obfuscate the development environment in these examples, because everybody's development environment is going to be different. Yes, there are typically two or three main environments (PC, Mac, Linux), but everybody's own style and conventions are different. Also, CakePHP is designed to run on any platform, so Windows should be running it just as well as any *nix-based system such as Linux or Mac.

> **Tip**
>
> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Step 1 – preparing the development environment

Before we begin, let's ensure that a few things are in place: an HTTP server, a database layer, the PHP scripting language, and source control.

As far as our operating system goes, we are going to stay as agnostic as possible. The file paths are Windows-based because we used Windows to write and create the examples in this book, but Unix file paths are perfectly acceptable as well. Do not panic.

The HTTP server (Apache)

The web server software should be set up and running. You should be able to visit `http://localhost` or `http://127.0.0.1/` in your web browser and see an Apache test page.

I prefer to use Apache in my day-to-day work. You can download the Apache HTTP Server from http://httpd.apache.org/download.cgi.

Alternatively, CakePHP can use IIS, LightHTTPD, nginx, or any other web server that you're familiar with. Whatever you do, you'll need to make sure that `mod_rewrite` or any other type of URL rewriting module is enabled. If you're looking for a quick and easy way to get up and running, you can look at the XAMPP project at http://www.apachefriends.org/en/xampp.html.

The database layer (MySQL)

Although CakePHP does not require a database layer, this book does. You should have the ability to create, connect to, read from, and write to a database.

MySQL is one of the tried and true technologies that is well supported by the PHP APIs. You can grab anything above Version 5.1 at http://www.mysql.com/downloads/mysql/.

CakePHP is also compatible with PostgreSQL, MS SQL Server, or SQLite.

PHP

This one is mandatory, as it literally puts the "PHP" in "CakePHP". The older versions of Cake can run with PHP Version 4, but we don't want to do that. To save yourself any issues in the near-to-medium future, grab at least Version 5.3 from http://php.net/downloads.php.

Again, if your head is spinning with all of this you can visit http://www.apachefriends.org/en/xampp.html.

Source control

CakePHP will work just fine without a source control system in place, and this book doesn't explicitly cover source control with its step-by-step instructions, but consider its inclusion as a major heading in this section a strong recommendation that you use it.

The CakePHP developers maintain an active repository on GitHub, so using git for your project makes a lot of sense. You can download it at http://git-scm.com/.

Step 2 – getting and installing the CakePHP application

The base CakePHP application is a collection of PHP files that work together to provide you with a solid foundation on which to do most things in a safe and structured way.

Base install

Let's get and install these files. We will use the following steps:

1. Create the `cake-starter` folder in your server's document root.

   When you visit `http://localhost` in your browser, this maps to a folder on the computer you're using. If you don't know which folder this is, you can view your web server's configuration files to find out. Search for files such as `httpd.conf`, `apache2.conf`, `nginx.conf`, or `lighthttpd.conf`.

   Typically, If you're using LAMP in Ubuntu or other Linux, your document root will probably be in `/var/www/`. If you're using XAMPP in Windows it would be in `C:\xampp\htdocs`. If you're using MAMP in MAC, it is located in the MAMP Application directory `/Application/MAMP/htdocs`.

2. Download the latest `.zip` archive of CakePHP.

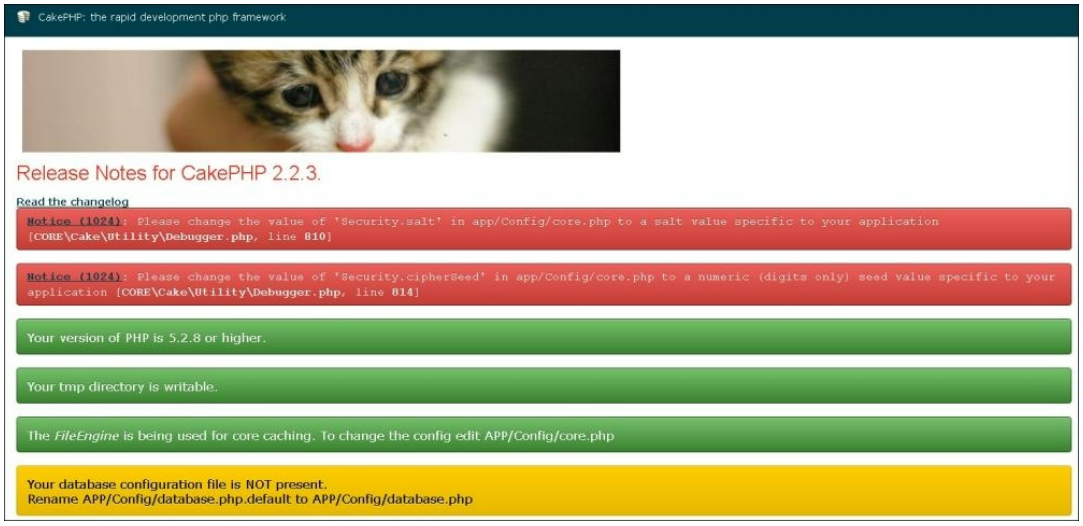
   This one's easy. Click the large **Download** emblem at http://cakephp.org.

3. Populate the `cake-starter` folder with the contents of the ZIP file.

   Unzip the file that you downloaded, and place the contents into the `cake-starter` folder.

4. Visit `http://localhost/cake-starter` in your browser.

If all goes well, you should see a page that looks like the following screenshot. If not, you may have to follow the instructions at http://book.cakephp.org/2.0/en/installation/advanced-installation.html#apache-and-mod-rewrite-and-htaccess, assuming that you're using Apache.



Cleaning up the errors on the home page

This home page serves a few purposes. The fact that it even renders properly tells us two things:

- The application is installed correctly
- The URL rewriting rules are functioning properly

CakePHP then does a number of configuration checks. It checks:

- Security settings
- PHP versions
- Cache settings
- Database settings

As you can see, we still have a little bit of work to do in order to get all the configuration settings to be green. Let's take care of that now. It's time to fire up your favorite text editor!

1. Edit the two security settings in `app/config/core.php`.

   Using your text editor, open the file contained in `app/config/core.php`, which lives in your `cake-starter` folder. Use the `find` function to find `Security.salt` and change it to some sort of unique value. Any random string will do for now. Repeat the steps to find `Security.cipherSeed` and change it to a random integer value. The longer the better, in both instances.

2. Rename `app/Config/database.php.default` to `app/Config/database.php` and edit the values.

   In the Development Environment section that we saw earlier, you should have a database set up with a user and password. First, rename `app/Config/database.php.default` to

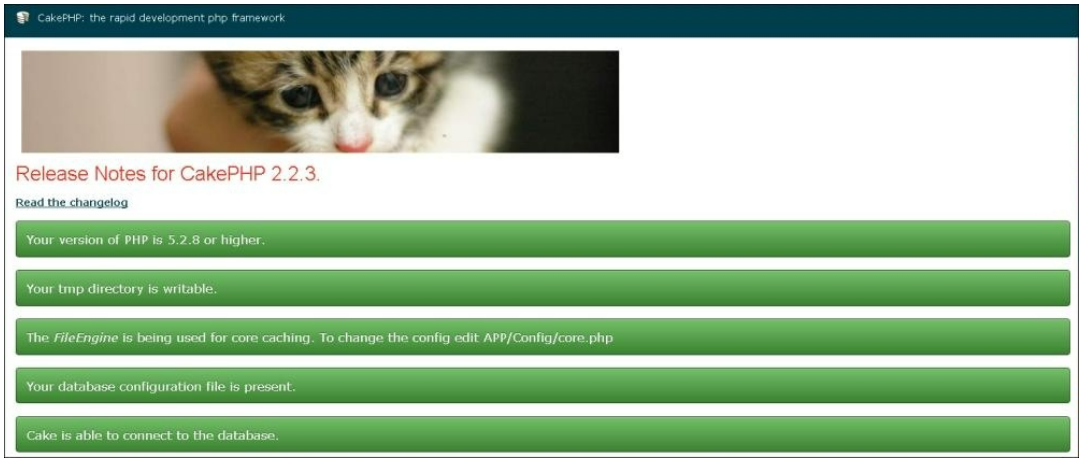
`app/Config/database.php`. Then, inside `app/Config/database.php` edit the array called `$default`. In the following example, I'm connecting to a local MySQL database called `cake_starter` with a user/password combination of `cakestarter`/`cake`. Obviously you'll want a more secure password in production, but for now this will be ok.

```
 * Database configuration class.
class DATABASE_CONFIG {

  public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'login' => 'cakestarter',
    'password' => 'cake',
    'database' => 'cake_starter',
    'prefix' => '',
    //'encoding' => 'utf8',
  );

  public $test = array(
}
```

There. Now all of the warnings are gone, and CakePHP has been successfully installed. Congratulations!



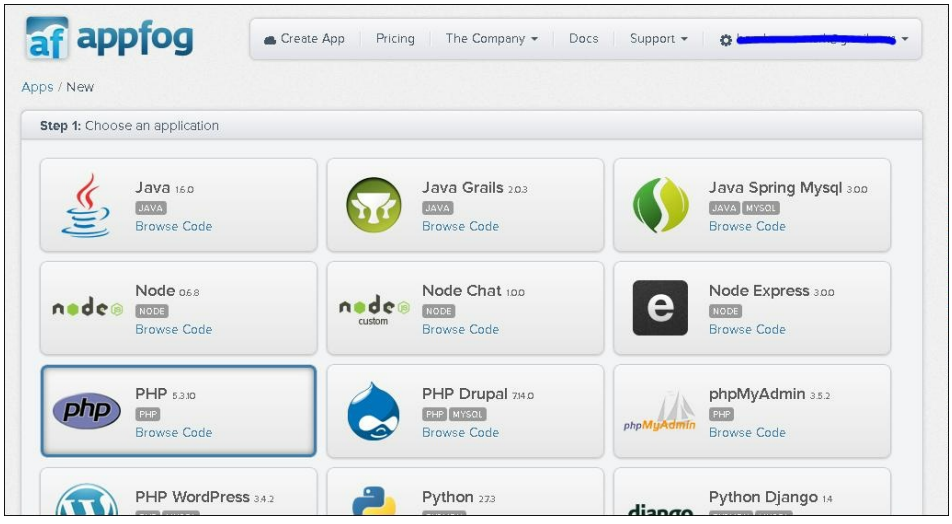
Step 3 – setting up the production deployment

Setting up a public-facing, production-ready environment can be a complex, time consuming process. AppFog, and each of the other services like it (Amazon AWS, Heroku, EngineYard, and so on), takes care of many aspects of this for you, and works to buffer you from this process, allowing you to concentrate on application development. This type of thing is referred to as **IaaS**, short for **Infrastructure as a Service**.

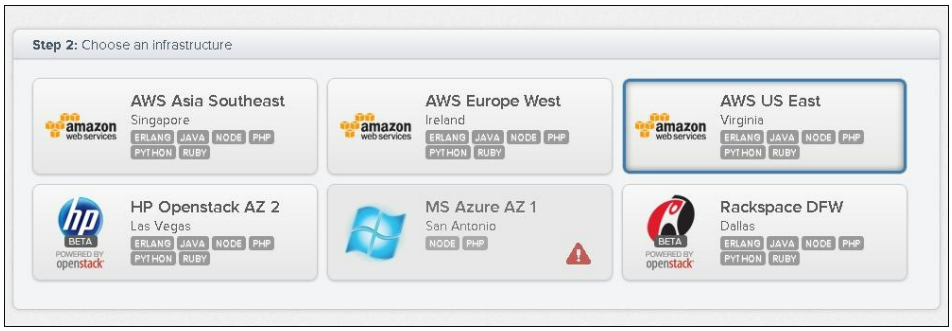
> ## Note
>
> This section is bit more advanced. However, having your full vertical stack—from development to production—up and running on day one is crucial. The more you do this type of thing, the more you will understand. Read carefully, and pay attention while following the steps, and you should be okay with registering on AppFog.com.

We've chosen AppFog as the deployment environment for this book. We have no vested interest in AppFog's success but the fact that it is free for small applications, relatively secure, supports PHP, and is relatively easy to set up is appreciated. We can use the following steps for setting up AppFog:

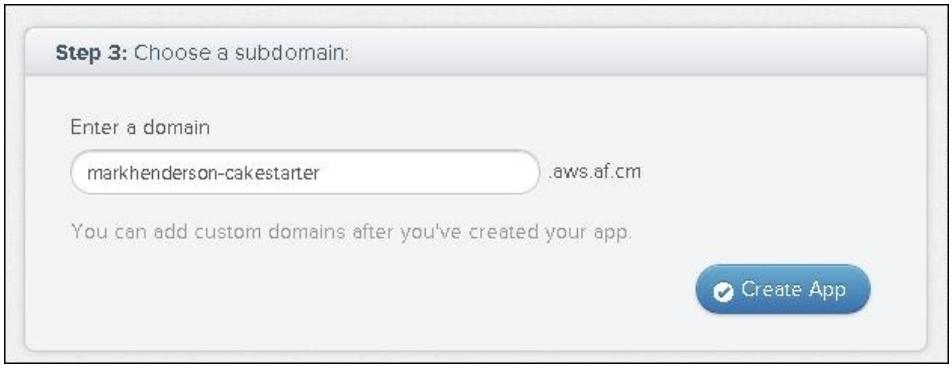
1. Visit https://console.appfog.com/signup and complete the registration.
2. On the next screen, select the application simply labeled **PHP**, as seen in the following screenshot:



3. Then, select the closest of the top row of AWS infrastructures, depending on your location:



4. For your subdomain, enter your last name, then a dash, then `cakestarter`:



5. Finally, click on **Create App**.

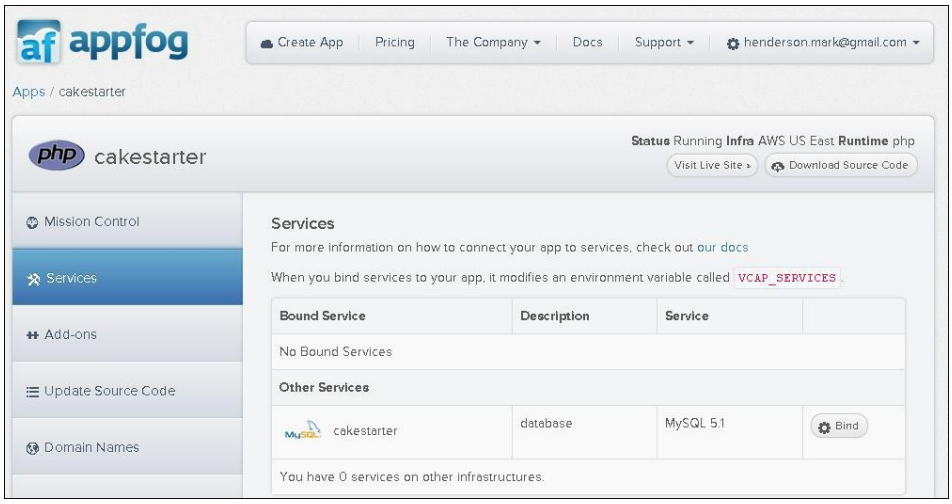
Binding your MySQL database to your application

You will then be brought to the **Mission Control** panel of your application. From here on, use the following steps:

1. Click on the **Services** tab on the left.

   The screen should show with a MySQL 5.1 service already provisioned.

2. Click on the **Bind** button. Your application will restart with the service bound.



Changing the file for deployment

What we need to do is set up our `database.php` file such that it reads from our local development database when we're working locally, but reads from AppFog's provisioned database when the code is in production. Luckily, CakePHP makes this easy for us; we only have to change one file.

After the large block comment at the beginning of the file, edit your file at `app/Config/database.php` so that it looks like this:

```
class DATABASE_CONFIG {
  public $default = null;

  public $dev = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'login' => 'cakestarter',
    'password' => 'cake',
    'database' => 'cake_starter',
    'prefix' => '',
    //'encoding' => 'utf8',
  );

  /* This database is used for the testing
framework. No important data should be stored
here. */
  public $test = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'login' => 'user',
    'password' => 'password',
    'database' => 'test_database_name',
    'prefix' => '',
```

```
    //'encoding' => 'utf8',
  );

  /* Constructor - this is called every time the
class is instantiated.*/
  function __construct() {

    // Read in the VCAP_SERVICES environment
variable, parse //the json, and check to see if
it exists.//If it does, use the settings for our
default// database connection.
    $vcap_services =
json_decode(getenv("VCAP_SERVICES"), true);
    if(!empty($vcap_services)) {
      $this->default = array(
        'datasource' => 'Database/Mysql',
        'persistent' => false,
        'host' =>
$vcap_services['mysql-5.1'][0]['credentials']
['host'],
        'login' =>
$vcap_services['mysql-5.1'][0]['credentials']
['username'],
        'password' =>
$vcap_services['mysql-5.1'][0]['credentials']
['password'],
        'database' =>
$vcap_services['mysql-5.1'][0]['credentials']
['name'],
        'port' =>
$vcap_services['mysql-5.1'][0]['credentials']
['port'],
      );
    } else {
      //If not, use our development settings.
      $this->default = $this->dev;
    }
  }
}
```

For security, AppFog uses an environment variable called `VCAP_SERVICES` to store database connection information. This code uses an `if` statement to determine whether that environment variable is populated or not. If it is, the database configuration is set to `AppFogs`; if not, it uses our local development settings from the previous section.

Downloading the af command-line tool

AppFog provides a simple command-line interface to upload, download, and sync your codebase between your development and production environments. It is packaged as what's called a "Ruby Gem" which is a

small program written in Ruby that you can obtain through the rubygems package manager.

- You can install Ruby from http://www.ruby-lang.org/en/downloads/
- You can install RubyGems from http://rubygems.org/pages/download

On your command line, test your installation by running the following command from the command line:

```
$ gem install af
```

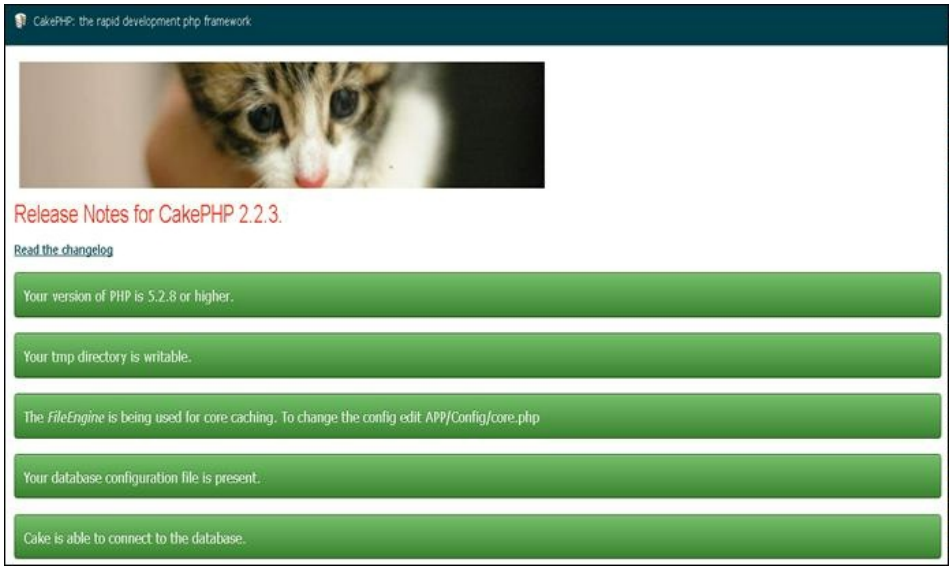
Pushing your code

Now we push our code to AppFog, which is made easy by the `af` tool. We can use the following steps:

1. Log in to `af` by typing `af login`.
2. Follow the instructions on the screen; you will be asked for your `username` and `password` for the AppFog website.
3. Then, update your code by typing the following command:

```
af update markhenderson-cakestarter
```

4. You will need to change `markhenderson-cakestarter` to the subdomain you defined at the beginning of this section. Remember, it will be your name, and then a dash, and then `cakestarter`.
5. If all goes well, you should see something much like your local homepage, as shown in the following screenshot:



And that's it

If this works, congratulations! You've successfully installed CakePHP in a complete vertical stack. You should be proud already

Quickstart – building a web application

The application idea in this book is inspired by my interest in indigenous languages, but the concept is generic enough to have many applications as well as being thorough enough that we can explore the vast majority of CakePHP's rich feature set.

Here's the back story: I'm working with somebody at the Oglala Lakota College in South Dakota. They have a great deal of content written in Lakota spread out across multiple sites, but the act of cross-referencing the words from the sites manually can be time-consuming for the students. If they were able to see the highlighted words on the websites and easily access their definitions from a simple interface, even as simple as an HTML`<abbr>` tag's default behavior, it would be much easier for them.

What they asked for is a simple glossary database of terms, definitions, and categories, packaged with some JavaScript that highlights the words on each page, and then displays the definitions in a popup or a tooltip. The JavaScript should be usable on any domain, using AJAX calls to "phone home" to the central glossary API and insert the words.

Step 1 – architecture

I cannot stress how important it is to take a deep breath and actually plan what you want to build. The more you do this at the outset, the less you will have to change your plan as you develop. Context switching in mid-development can often prove disastrous.

At the end of this section, we will have three things in place: the models, the controllers, and the views. This is no coincidence—the MVC structure exists for a reason, and a happy consequence of setting up the full MVC stack is that it gets you about 80 percent of the way towards getting a finished application in place. You'll be amazed at how much we've covered by the end of this section.

The models

In the Model-View-Controller architecture, the models contain "domain-specific logic". In layman's terms, that means that they encapsulate any data or functionality based around specific object types (vehicles, birds, or categories) or specific concept types (virtualization, translation, and so on). In CakePHP, our models are PHP class definitions, and they provide a

standardized API that we can use to access and act upon the data in our application code.

For this application, we only require two models: one for our glossary terms, and another for the categorization of the terms. Because Cake likes its model names to be in camel-cased with initial caps, let's call them GlossaryTerms and Tags.

GlossaryTerms will need to contain the following information:

- **ID**: This represents a unique identifier for each GlossaryTerm. Cake and MySQL work in concert to provide this to you automatically.
- **Term**: This will be a string that contains the GlossaryTerm. This should be unique. For example, "waste" (pronounced wash-tay) in Lakota is the term.
- **Definition**: This will also be stored in the database as a string. For example, "good, ok, fine, positive" would be the definition. This does not have to be unique.

The categories will contain the following information:

- **Id**: This is a unique identifier for categories, automatically created.
- **Name**: This is the title of the tag, to be displayed to the user. It should be stored as a string and must be unique.
- **Definition**: This is another user-facing descriptor for the category. It should be stored as a string and is not unique.

In order for these two tables work together, we will need a third table called a lookup table, which matches the Tag IDs with the Term IDs. This is slightly complex, but CakePHP can do this work for us without us having to explicitly define another model. All that is visible to us are the more tangible aspects of our application.

You may notice that these two tables are very similar. It may be tempting to try to combine them but it's important that we separate our concerns, because as the application grows, the similarity between the two tables will shrink.

The controllers

The controller is the middle layer between the data represented in the models and their display to the end user in the views. Data is organized and sometimes filtered in this layer, and structured in a way that a view (and therefore a user) can understand. Also note that the names of our controllers and resultant actions affect the URLs that our application presents. This, of course, can be configured, but this is default behavior you need to be aware of.

Once you start working within MVC and interacting with other members of the MVC community, you will hear a concept called "fat models, thin controllers". This is a helpful way of framing your code—you're going to want as much domain-specific logic in your models as possible (lots of lines of code) and very little in the controllers, besides rearranging the data to be displayed in the view (much less code than the model).

What we need to begin (and what 99 percent of web applications tend to need to begin) is what's referred to as a **CRUD** interface. This stands for **Create**, **Read**, **Update**, **Delete**, which comprises the core actions that a user will take against a database. For now, we will use CakePHP's built-in scaffolding feature to make our lives easier.

The views

In an MVC structure, the views are what are displayed to the user. These are represented in `.ctp` files, which are executed as PHP. Each of the items in CRUD will be represented in our views; we will create interfaces to add, edit, delete, and view each of the items represented by our models.

Step 2 – scaffolding with the command-line tool

CakePHP comes packaged with the Cake command-line tool, which provides a number of code generation tools for creating models, controllers, views, data fixtures, and more, all on the fly. For the remainder of this section, we will use the command-line tool to generate our desired CRUD interface in a matter of minutes.

Please note that this is great for prototyping, but is non-ideal for a production environment.

On your command line, from the `cake-starter` folder, type the following:

```
cd app
Console\cake bake
```

You will see something similar to the following:

```
> Console/cake bake
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/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)
>
```

As you can see, there's a lot to be done with this tool. Note that there are other commands beside `bake`, such as `schema`, which we are about to use.

Creating the schema definition

Inside of your `app/Config/Schema` folder, create a file called `glossary.php`. Insert the following code into this file:

```
<?php

/**
 * This schema provides the definitions for the
core tables in the glossary app.
 *
 * @var $glossary_terms - The main
terms/definition table for the app
 * @var $categories - The categories table
 * @var $terms_categories - The lookup table, no
model will be created.
 *
 * @author mhenderson
 *
 */
class GlossarySchema extends CakeSchema {
  public $glossaryterms = array(
      'id' => array('type' => 'integer', 'null' =>
false, 'key' => 'primary'),
      'title' => array('type' => 'string', 'null' =>
false, 'length' => 100),
      'definition' => array('type' => 'string', 'null'
=> false, 'length' => 512)
  );

  public $categories = array(
      'id' => array('type' => 'integer', 'null' =>
false, 'key' => 'primary'),
```

```
        'name' => array('type' => 'string', 'null' =>
false, 'length' => 100),
        'definition' => array('type' => 'string', 'null'
=> false, 'length' => 512)
    );

    public $glossaryterms_categories = array(
        'id' => array('type' => 'integer', 'null' =>
false, 'key' => 'primary'),
        'glossaryterm_id' => array('type' => 'integer',
'null' => false),
        'category_id' => array('type' => 'string', 'null'
=> false)
    );
}
```

This class definition represents three tables: `glossaryterms`, `categories`, and a `lookup` table to facilitate the relationship between the two tables. Each variable in the class represents a table, and the array keys inside of the variable represent the fields in the table. As you can see, the first two tables match up with our earlier architecture description.

Creating the database schema

On the command line, assuming you haven't moved to any other folders, type the following command:

```
Console/cake schema create glossary
```

You should then see the following responses. When prompted, type `y` once to drop the tables, and again to create them.

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/app
---------------------------------------------------
-------------
Cake Schema Shell
---------------------------------------------------
-------------

The following table(s) will be dropped.
glossaryterms
categories
glossaryterms_categories
Are you sure you want to drop the table(s)? (y/n)
[n] > y
```

```
Dropping table(s).
glossaryterms updated.
categories updated.
glossaryterms_categories updated.

The following table(s) will be created.
glossaryterms
categories
glossaryterms_categories
Are you sure you want to create the table(s)?
(y/n)
[y] > y

Creating table(s).
glossaryterms updated.
categories updated.
glossaryterms_categories updated.
End create.
```

If you look at your database now, you will notice that the three tables have been created. In the section *Top 6 features you'll want to know about*, we will make modifications to the `glossary.php` file and run the `cake schema` command again to update it.

If you want to try something a little more daring, you can use the migrations plugin found at https://github.com/CakeDC/migrations. This plugin allows you to save "snapshots" of your schema to be recalled later, and also allows you to write custom scripts to migrate "up" to a certain snapshot version, or migrate "down" in the event of an emergency or a mistake.

Creating the models

On the command line, run the following commands to create a Category model. Note that when you're done, you'll need to repeat this step again to create a second model for `glossaryterms`. All of these `bake` steps will need to be carried out twice. I just don't feel the need to waste any space in this book by listing them twice.

Please note that if you do not have PHPUnit installed, then you may see some errors being generated during execution of the coming steps.

```
Console/cake bake model
```

You should see the following output:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
```

```
App : app
Path: /path/to/app
-------------------------------------------------
-------------
-------------------------------------------------
-------------
Bake Model
Path: /path/to/app/Model
-------------------------------------------------
-------------
Use Database Config: (default/dev/test)
[default] >
Possible Models based on your current database:
1. Category
2. Glossaryterm
3. GlossarytermsCategory
Enter a number from the list above,
type in the name of another model, or 'q' to exit
[q] > 1
```

The defaults in the `bake` app are very logical and incredibly helpful. In fact, from this point on you can just hit *Enter* each time you're prompted, and get the result that you want. Not only does this tool generate code, it's also very smart about reading your code to figure out exactly what it is you're trying to do.

That being said, a couple steps worth taking note of are the following:

```
Would you like to supply validation criteriafor
the fields in your model? (y/n)
[y] >
```

Validation is the concept of examining user input and deciding whether or not it should actually be stored in the database, usually based around the content or format of the input. Out of the box, CakePHP provides a number of useful validators, such as not blank, numeric, e-mail address, but it also allows you to create and use your own. Some examples of custom validators could be the names of states in the U.S., phone numbers with a certain area code, bad language filtering, and so on.

The second step worth noting is:

```
Would you like to define model associations
(hasMany, hasOne, belongsTo, etc.)? (y/n)
[y] >
```

CakePHP can keep track of how your database tables relate to each other. For example, if you're representing a tree in the code, then you may have a relationship like this:

- A tree has one trunk
- A trunk has many branches
- Each branch belongs to the trunk

Again, CakePHP keeps track of this for you and gets out of your way so you can focus on building the parts of the app that you want to build.

Once you've complete this step, bake will generate three files as follows:

- `app/Model/Category.php`: This file is the model itself, which is a collection of code that provides an API to your data
- `app/Test/Case/Model/CategoryTest.php`: This file is a test case to be used in our automated testing
- `app/Test/Fixture/CategoryFixture.php`: This file is a collection of data to be used as initial or testing data

After you complete this step, run `Console/cake bake model` again and select `2`, initially, to create a `glossaryterm` model. You don't have to create a model for `glossarytermscategory`.

Creating the controllers

Much like models, the `cake` tool provides a number of options for creating controllers. We're going to use the tool now to create some basic actions. Namely, `index`, `add`, `view`, and `edit`.

On the command line, enter the following:

```
Console/cake bake controller
```

You should receive the following output:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/app
---------------------------------------------------
-------------
---------------------------------------------------
-------------
Bake Controller
```

```
Path: /path/to/app/Controller
---------------------------------------------------
-------------
Use Database Config: (default/dev/test)
[default] >
Possible Controllers based on your current
database:
---------------------------------------------------
-------------
 1. Categories
 2. Glossaryterms
 3. GlossarytermsCategories
Enter a number from the list above,
type in the name of another controller, or 'q' to
exit
[q] > 1
```

You'll need to repeat this again for `glossaryterms` as well, but not for `glossarytermscateories`. Here's the rest of the output. You can't just hit *Enter* through this! See the following for the correct sequence of answers that you'll need.

```
---------------------------------------------------
-------------
Baking CategoriesController
---------------------------------------------------
-------------
Would you like to build your controller
interactively?
Warning: Choosing no will overwrite the
CategoriesController. (y/n)
[y] > y
Would you like to use dynamic scaffolding? (y/n)
[n] > n
Would you like to create some basic class methods
(index(), add(), view(), edit())? (y/n)
[n] > y
Would you like to create the basic class methods
for admin routing? (y/n)
[n] > n
Would you like this controller to use other
helpers
besides HtmlHelper and FormHelper? (y/n)
[n] > n
Would you like this controller to use any
components? (y/n)
[n] > n
Would you like to use Session flash messages?
(y/n)
[y] > y

---------------------------------------------------
-------------
```

```
The following controller will be created:
---------------------------------------------------------
-------------
Controller Name:
        Categories
---------------------------------------------------------
-------------
Look okay? (y/n)
[y] > y
```

You can also run:

```
Console/cake bake controller Categories
```

This will allow you to skip a few of the earlier steps and create the following two files:

- `app/Controller/CategoriesController.php`: This is our controller file for categories
- `app/Test/Case/Controller/CategoriesControllerTest.php`: This is our test case for the categories controller

Repeat this step again with identical options to create a `Glossaryterms` controller. You don't need to make a controller for `GlossarytermsCategories`.

Note that even though we're talking about "scaffolding" our app in this chapter, we are going to choose "n" when the `bake` tool asks us if we want to use dynamic scaffolding when we create our controller. Let's explore this a little further.

At first glance, whether or not you choose yes or no at this point makes no immediate difference. You are still presented with a CRUD interface to your data, using CakePHP's built-in frontend templates. There is no effective difference to the user between choosing dynamic scaffolding and using the auto-generated one.

However, we are not users. We are the developers of the application, and we have a lot more at stake than just being able to create, read, update, and delete data. We have to enhance the functionality of the app, and more importantly, maintain it.

That being said, let's take a look at the code and note the important differences between choosing dynamic scaffolding and not.

When you choose to scaffold a controller, you end up with a controller that looks like this:

```
<?php
App::uses('AppController', 'Controller');
/**
 * Categories Controller
 *
 * @property Category $Category
 */
class CategoriesController extends AppController {
/**
 * Scaffold
 *
 * @var mixed
 */
  public $scaffold;
}
```

That's it.

However, if you were to say `no` to dynamic scaffolding, and `yes` to basic class methods for `add`, `index`, `edit`, and `delete`, you'd get something like this:

```
<?php
App::uses('AppController', 'Controller');
/**
 * Categories Controller
 *
 * @property Category $Category
 */
class CategoriesController extends AppController {

/**
 * index method
 *
 * @return void
 */
  public function index() { ...}

/**
 * view method
 *
 * @throws NotFoundException
 * @param string $id
 * @return void
 */
  public function view($slug= null) { ...}

/**
 * add method
 *
 * @return void
 */
  public function add() { ... }
```

```
/**
 * edit method
 *
 * @throws NotFoundException
 * @param string $id
 * @return void
 */
  public function edit($id = null) { ... }

/**
 * delete method
 *
 * @throws MethodNotAllowedException
 * @throws NotFoundException
 * @param string $id
 * @return void
 */
  public function delete($id = null) { ...
}
```

Take a look at this code closely; this does everything that our scaffold does, but exposes a lot more of it to us as developers. We can see:

- The calls to our models (they look like `$this->Category->*`). Inside of a controller, associated and declared models are attached to the `$this` variable for use in the controller code.
- Messages that are set to display to the user, also known as the **Flash** messages (they look like `$this->Session->setFlash()`). The `$this->Session` is a component, something that lives with the controller to expose specific functionality for you.
- Error handling, anywhere you see a line that starts with `throw new`.
- Some very basic security, in lines such as `if(!$this->request->is('post'))` which only allows `HTTP POST` requests through.

There's nothing wrong with dynamic scaffolding to start with, but once you want to edit anything, you're a bit out of luck. This doesn't just apply to controller code either. Skipping the scaffolding step and going the "long way" provides you with the ability to also edit your view markup, as you're about to see.

Creating the views

Last step! We will create preliminary views for each of the actions that we created inside of our controllers, namely, `index`, `add`, `view`, and `edit`.

```
Console/cake bake view
```

Answer `y` and `n` as you seen in the following output, and repeat for `Glossaryterms`.

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/app
---------------------------------------------------
-------------
---------------------------------------------------
-------------
Bake View
Path: C/path/to/app/View
---------------------------------------------------
-------------
Use Database Config: (default/dev/test)
[default] >
Possible Controllers based on your current
database:
---------------------------------------------------
-------------
 1. Categories
 2. Glossaryterms
 3. GlossarytermsCategories
Enter a number from the list above,
type in the name of another controller, or 'q' to
exit
[q] > 1
Would you like bake to build your views
interactively?
Warning: Choosing no will overwrite Categories
views if it exist. (y/n)
[n] > y
Would you like to create some CRUD views
(index, add, view, edit) for this controller?
NOTE: Before doing so, you'll need to create your
controller
and model classes (including associated models).
(y/n)
[y] > y
Would you like to create the views for admin
routing? (y/n)
[n] > n
Baking `index` view file...
Baking `view` view file...
Baking `add` view file...
Baking `edit` view file...
---------------------------------------------------
-------------

View Scaffolding Complete.
```

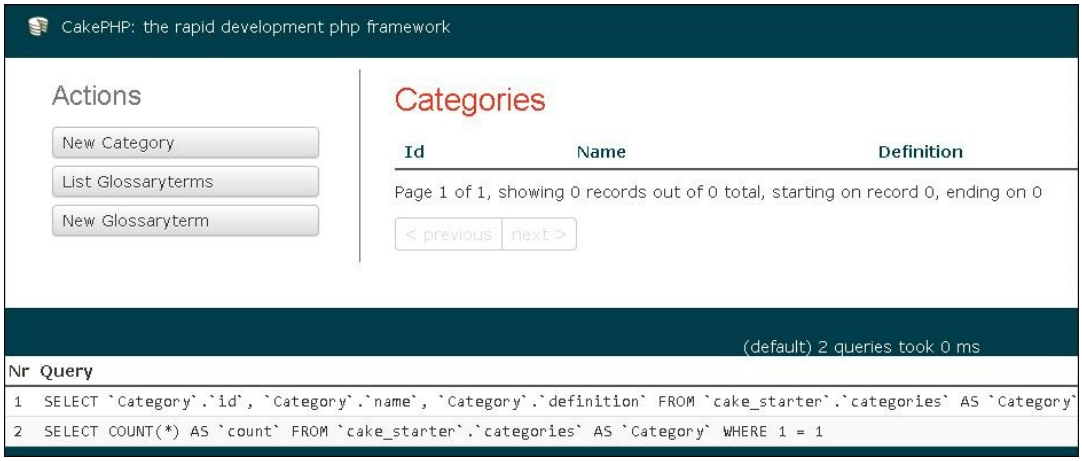
This created four files:

- `app/View/Categories/index.ctp`: A listing of all categories.
- `app/View/Categories/edit.ctp`: An edit form for a single category that already exists.
- `app/View/Categories/add.ctp`: A blank form for creating a new category.
- `app/View/Categories/view.ctp`: This is the "detailed view" of our category. This view will be called upon whenever we see a single category by itself.

> **Note**
>
> The defaults for baking a view are so logical that you can type "V" at the initial `bake` prompt, then hit *Enter* to give the default answer of "n" when it asks if you want to build your view interactively. With that, you will get fully functional views for your model/controller pair with no configuration whatsoever!

Step 3 – taking a look at our scaffolded app

If you visit `http://localhost/cake-starter` you won't see much of a difference. However, visit `http://localhost/cake-starter/categories`. What you should see is an exciting, surprisingly rich interface that contains a lot of information and functionality. If you receive any file permission errors, you can likely fix them by running the `chmod` command.



On the left are your actions. You can create a new category or glossary term, or view a listing of glossary terms. Once you enter some data, you'll be able to view each individual item, and delete items as necessary.

On the bottom is a list of the database queries that have been called, and some profiling data about each call. As the application grows, this information will become invaluable in debugging the database layer and performance.

Another tool that could potentially be useful is called **debugkit**, and is available at: https://github.com/cakephp/debug_kit.

Take some time to click around. A lot has been created in very little time. In the next section, *Top 6 features you'll want to know about*, we will begin to carve, craft, and sculpt, as this simple, yet fully functional app becomes our own.

Step 4 – what else does "bake" do for us?

I want to wait until the next section, *Top 6 features you'll want to know about*, to get down and dirty with writing custom code, so for now, let's take a look at what else the `bake` tool can do for us.

API documentation

An oft-overlooked feature of bake is the ability to look up documentation about CakePHP's core APIs from the command line. It accomplishes this by parsing out what's known as **doc block** documentation. You can see an example of this in any of your baked models or controllers, for example, at the top your `CategoriesController`:

```
/**
 * Categories Controller
 *
 * @property Category $Category
 */
```

Obviously, the documentation in Cake's core code is a lot more verbose and helpful than this. Let's have a look. Go to your command line and run the following command:

```
Console/cake api
```

This will give you a basic Help message that details the usage. It's always nice to run the base command in this fashion to get an idea of what it's expecting. In fact, this is a good practice for any command-line tool. I won't post the output here since it's subject to change, but you get the idea.

What I will post is the output of the following command:

```
Console/cake api helper HtmlHelper
```

The output will be as follows:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: path/to/app
---------------------------------------------------
-------------
HtmlHelper
---------------------------------------------------
-------------
1. addCrumb($name, $link = NULL, $options = NULL)
2. charset($charset = NULL)
3. css($path, $rel = NULL, $options = array ())
4. div($class = NULL, $text = NULL, $options =
array ())
5. docType($type = 'html5')
6. getCrumbList($options = array (), $startText =
false)
7. getCrumbs($separator = '&raquo;', $startText =
false)
8. image($path, $options = array ())
9. link($title, $url = NULL, $options = array (),
$confirmMessage = false)
10. loadConfig($configFile, $path = NULL)
11. media($path, $options = array ())
12. meta($type, $url = NULL, $options = array ())
13. nestedList($list, $options = array (),
$itemOptions = array (),$tag = 'ul')
14. para($class, $text, $options = array ())
15. script($url, $options = array ())
16. scriptBlock($script, $options = array ())
17. scriptEnd()
18. scriptStart($options = array ())
19. style($data, $oneline = true)
20. tableCells($data, $oddTrOptions = NULL,
$evenTrOptions = NULL, $useCount = false,
$continueOddEven = true)
21. tableHeaders($names, $trOptions = NULL,
$thOptions = NULL)
22. tag($name, $text = NULL, $options = array ())
23. useTag($tag)
Select a number to see the more information about
a specific method. q to quit.
l to list.
[q] > 4
---------------------------------------------------
-------------
```

```
HtmlHelper::div($class = NULL, $text = NULL,
$options = array ())
-------------------------------------------------
-------------

  Returns a formatted DIV tag for HTML FORMs.

  ### Options

  - `escape` Whether or not the contents should
be html_entity escaped.

  @param string $class CSS class name of the div
element.
  @param string $text String content that will
appear inside the div element.
    If null, only a start tag will be printed
  @param array $options Additional HTML attributes
of the DIV tag
  @return string The formatted DIV element
  @link http://book.cakephp.org/2.0/en/core-
  libraries/helpers/html.html#HtmlHelper::div

Select a number to see the more information about
a specific method. q to quit.
l to list.
[q] >
```

If you don't appreciate how awesome this is, let me explain a bit. You'll be using this `HtmlHelper` extensively throughout the development of your application's frontend; you'll be adding CSS files, images, breadcrumbs, and more. You're going to have questions.

You can hop online to *CakePHP Cookbook 2.x* to look up what methods `HtmlHelper` exposes to you, and what arguments each method is expecting, but maybe the command line is easier for you, or maybe an Internet connection isn't available to you and you just need a quick answer. The `bake` tool has all of the documentation built-in, simply by being able to parse the `doc` block comments from the code. Pretty cool right!

Internationalization, or i18n

In this day and age, it's insane to think that your application will, or should, be consumed in only one language. If you want to really expand out into the world and have as many people as possible consume your application, then getting i18n working in your app is effort that you're going to want to expend.

Luckily, again, CakePHP makes this relatively easy for you, and the `bake` tool has functionality that assists you with this. We're not going to be

translating anything just yet, but we do want to lay the groundwork for this functionality just as we we would any other part of our application.

Generating .pot files

Before I explain what a `.pot` file is, I should explain `.po` files. There is a Unix program called `gettext` that reads "portable object" files (or `.po` files), which contain multiple translated versions of an application's text and displays the proper translation based on the user's desired and selected language. CakePHP utilizes `gettext` under the hood, by providing an interface for it that looks like the `__()` function.

This will take care of all of the static text for you. Any dynamic text will have to be handled separately, at the controller level, inside your business logic code.

Any time `__()` appears with a string passed in as a variable, it is telling our system "whatever string is inside this function, return the translated version of that string". Cake's bake tool gives us a great utility in that it searches our application code for calls to `__()`, and then collects them all into a portable object template file, or `.pot`.

So let's do this. From our command line, we run:

```
Console/cake i18n
```

We will get the following output:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/app
---------------------------------------------------
-------------
I18n Shell
---------------------------------------------------
-------------
[E]xtract POT file from sources
[I]nitialize i18n database table
[H]elp
[Q]uit
What would you like to do? (E/I/H/Q)
What would you like to do? (E/I/H/Q)
> E
Current paths: None
What is the path you would like to extract?
[Q]uit [D]one
```

```
[C:\Users\Mark\Documents\webroot\cakestarter\app
\] > [enter]

Current paths:
C:\Users\Mark\Documents\webroot\cakestarter\app
\
What is the path you would like to extract?
[Q]uit [D]one
[D] > D

Would you like to extract the messages from the
CakePHP core? (y/n)
[n] > n
What is the path you would like to output?
[Q]uit
[C:\Users\Mark\Documents\webroot\cakestarter\app
\Locale] > [enter]

Would you like to merge all domains strings into
the default.pot file? (y/n)
[n] > n
```

You'll see it spin through a number of your files in `Config`, `Console`, `Model`, `Test`, `View`, `Webroot`—basically all of your app folders—looking for strings in your PHP that are wrapped in a function that looks like `__()`.

Bake will collect all of these, and then put them in a file called `default.pot` in your `app/Locale` folder. Let's take a quick glance at what this looks like:

```
#: View\Categories\add.ctp:14
#: View\Categories\edit.ctp:15
#: View\Categories\index.ctp:8;40
#: View\Categories\view.ctp:22;40
#: View\Glossaryterms\add.ctp:14
#: View\Glossaryterms\edit.ctp:15
#: View\Glossaryterms\index.ctp:8;40
#: View\Glossaryterms\view.ctp:22;40
#: View\Themed\Bootstrap\Categories\index.ctp:14
#: View\Themed\Bootstrap\Categories\view.ctp:23
#:
View\Themed\Bootstrap\Glossaryterms
\index.ctp:15
#: View\Themed\Bootstrap\Glossaryterms\view.ctp:24
msgid "Actions"
msgstr ""

#: View\Categories\add.ctp:17
#: View\Categories\edit.ctp:19
#: View\Categories\view.ctp:26
#: View\Glossaryterms\add.ctp:18
#: View\Glossaryterms\edit.ctp:20
#: View\Glossaryterms\index.ctp:43
```

```
#: View\Glossaryterms\view.ctp:28
#: View\Themed\Bootstrap\Glossaryterms\view.ctp:9
msgid "List Categories"
msgstr ""
```

The syntax of this file is simple, but you might not be used to it. The lines starting with `#` are comments, and simply tell you in which files it found this particular string . `msgid` refers to the string that it found, that is, `__('Actions')` for the first definition. `Msgstr` is a placeholder—it is meant to contain the translated string for "Actions" in whatever language you'll be translating to.

Later, we'll translate some of these strings to Lakota by creating a sub folder in our `app\Locale` folder.

Step 5 – deploying to production

Now that we have something working locally, we can take the time to update our production environment.

Pushing the code

On your command line, navigate back to your `cake-starter` folder, and use the `af` tool to update the code on the server:

```
cd /path/to/cake-
starter
af update [yourname]-cakestarter
```

You should see a success message, and your code will be updated. However, if you visit your production URL you will see an error message. This is because we need to update the database.

Updating the database

To update the database, start by creating a SQL dump of your existing database. You can do this by using the export feature of your database GUI, or by using the following command line:

```
mysqldump -u cakestarter -p cake_starter >
dump.sql
```

The syntax for this command is `mysqldump -u [user] -p [database]`. These should map to the development database variables that we have defined in `app/Config/database.php`.

Next, use the `af` command-line tool to access the database on your AppFog instance. The results will vary from installation to installation,, but make note of the `Service connection info` section. You will need those values in the next step.

```
af tunnel
1: exampleapp1-mysql
Which service to tunnel to?: 1
Password: ********
Getting tunnel connection info: OK

Service connection info:
username : uaLDy9EhhvMLq
password : p5Odjf6E5O7uW
name : dc1aaa897343f4eb1aed047ec7c86f19f

Starting tunnel to exampleapp1-mysql on port
10000.
1: none
2: mysql
Which client would you like to start?: 1
```

Then, you can import your data by running the following command:

```
mysql --protocol=TCP --host=localhost
--
port=10000 --user=[username]
--password=[password] [name] < dump.sql
```

Replace the bracketed variables with the information from the `Service connection info` section of the output. If all goes well, your deployment environment should mirror your development environment. The data will be blank, but everything else should be identical.

Step 6 – congratulations!

You've done a lot of work in a short amount of time, and you should be proud. Not a lot of people can build web applications this quickly. Alright, maybe that's not true—anybody that downloads the bake tool can do what we've done, but don't you feel like you have some sort of secret power now?

Think about it. In one section of this book we have:

- Created models that represent your data in code and give you a remarkable amount of power over the data they contain

- Created controllers that allow you to utilize the models and their methods to get the data into a state that you want
- Created views that will allow users to create, read, update, and delete the application's data
- Found API documentation to suit your needs, online or offline
- Created the foundation for internationalization, or i18n
- Deployed your app to a public-facing production server, ready to display to a client or other stakeholder

And we have done all of this without writing more than what, 20 lines of code? I don't get evangelical about software often but these people have done a pretty great thing with this bake tool.

If you don't feel you have a secret power, maybe you will at the end of the next section where we crack the code open and make our application even more awesome. Read on.

Top 6 features you'll want to know about

Now that we've got our application's scaffolding in place, we're going to want to make it the best it can be using the tools we have available. Although a large part of this work will utilize our experience as web developers, and incorporate technologies such as HTML5, CSS3, and JavaScript, these will not be the focus of this section.

This section starts with a discussion on naming conventions, and then it is organized by practical matters: small but powerful ideas and techniques that we can use to make our application better, specifically by using CakePHP's features and conventions.

In the course of doing so, you will learn the following:

- Models, views, and controllers (of course)
- Themes
- Routing
- Behaviors
- Fixtures and tests

What do we know at this point?

Just a few minutes with the command-line tool has taught us the following:

- Data objects are known as models, user interfaces are known as views, and the logic that binds them together are called controllers.
- Models, views, and controllers can be easily created by using the command line bake tool, which generates PHP files that define our MVC structure in code. Models, controllers, and views are placed in `app/Model`, `app/Controller`, and `app/View` respectively.
- Visiting `http://localhost/categories` and `http://localhost/glossaryterms` will show us the completed pages that bake has generated for us.
- If and when we internationalize our application, the files necessary to do so are in place inside of `app/Locale`.
- If we need help, we can look at either http://book.cakephp.org or the built in API documentation that the cake tool provides.

The command-line tool really is an impressive feature of CakePHP. Not only has it scaffolded out a functional app, it has also given us an innate sense of how everything is tied together. However, we don't want to leave

too much to instinct, so let's talk through how this all works before we move any further.

Routing – naming matters!

The way that files and folders are named in your application is critical to how the underpinnings of CakePHP work. The system expects certain files to exist based on things such as PHP function names and the names of other files. This is most readily apparent in the relationship between your controllers, your views, and the URLs that you use to access them.

Without any modifications to your base application, the core routing concept is:

- URLs map to controllers and the methods (also known as actions) defined within them.
- Controller names and actions map to folders within `app/Views` and the specific `.ctp` files contained within these folders.

This automatic mapping of URLs to controllers can remain simple and automatically-generated or it can become as complex as you want, by using Cake's routing API.

A bit more detail

When a URL is requested in a CakePHP app, the system will try to parse the request as follows:

```
http://example.url/[controller]/[action]/
[arguments*]
```

The arguments portion of the URL is optional, and the action will default to the index action if left blank. From there, it will parse the `Views` directory, in a similar way, to find the appropriate `.ctp` file to render.

In terms of our app, this means:

1. The `http://localhost/categories` URL is requested.
2. CakePHP looks in `app/Controller` for our `CategoriesController.php` file.
3. Because the action portion of the URL is blank, it calls the `index()` function of `CategoriesController`.
4. The system looks in `app/View` for the `Categories` folder, and renders `index.ctp`.

Note that `categories` in the URL is in lowercase, whereas `Categories` in the code is capitalized. This is correct. Let's try a more complicated example, to make sure, that we understand how these relationships work.

1. The `http://localhost/glossaryterms/add` URL is requested.
2. CakePHP looks in `app/Controller` for our `GlossarytermsController.php` file.
3. Because the action portion of the URL is not blank, it calls `GlossarytermsController::add()`.
4. The system looks in `app/View` for the `Categories` folder, and renders `add.ctp`.

Wait! Let's be thorough. One more example:

1. The `http://localhost/glossaryterms/edit/2` URL is requested.
2. CakePHP looks in `app/Controller` for our `GlossarytermsController.php` file.
3. Because the action portion of the URL is not blank, it calls `GlossarytermsController::edit()`.
4. Because the arguments portion of the URL is also not blank, it passes in the value of `2` as the argument into the `edit()` function of `GlossarytermsController`. The following code is what this looks like:

```
/**
 * edit method
 *
 * @throws NotFoundException
 * @param string $id
 * @return void
 */
  public function edit($id = null) {
    $this->Glossaryterm->id = $id;
    if (!$this->Glossaryterm->exists()) {
      throw new
NotFoundException(__('Invalid
glossaryterm'));
    }
    if ($this->request->is('post') ||
$this->request->is('put')) {
      if
($this->Glossaryterm->save($this->request-
>data)) {
        $this->Session->setFlash(__('The
glossaryterm has been saved'));
        $this->redirect(array('action' =>
'index'));
      } else {
```

```
            $this->Session->setFlash(__('The
glossaryterm could not be saved. Please,
try again.'));
        }
    } else {
        $this->request->data =
$this->Glossaryterm->read(null, $id);
    }
    $categories =
$this->Glossaryterm->Category-
>find('list');
    $this->set(compact('categories'));
  }
```

5. The system looks in `app/View` for the `Glossaryterms` folder, and renders `edit.ctp`.

Starting to make sense? If not, try a few more on your own and meet us in the next section.

Views and themes

One of the first things you're going to want to do is to change the appearance of your application. Most of the time, you will write your own `.ctp` files, and edit the ones that already exist. In fact, most of your work in Cake will be doing just that. The blog tutorial covers this, so we're going to quickly discuss some of the folders inside of `app/Views`, and then have some fun with the themes.

Views summary

Let's start, then, by examining the structure of your `app/Views` folder:

```
app/View/
  Categories/
  Elements/
  Glossaryterms/
  Layouts/
  Pages/
  Themed/
```

All of these folders contain `.ctp` files, which are essentially snippets of PHP and HTML that render in combination with one another to compose an entire web page. These are your views.

Notice that this folder structure already has some logic to it. Looking at this list, we can already glean an understanding of what some of these folders contain. Let's familiarize ourselves with a few, in detail:

- `Categories/` and `Glossaryterms/` contain the custom application views that you generated in the *Quickstart – building a web application* section.
- `Pages/` seems like a special directory, but it actually follows the same convention as the two folders mentioned in the previous bullet point. If you look in your `Controllers` directory you will see `PagesController` alongside the ones we generated in the last section. This is used for static pages, such as an "about us" page or a "terms of service page".
- `Layouts/` contains a series of outer "frames" for your views: the header and footer elements that contain your inner views. This is used for your `doctype`, your `<head>` element, and scripts that you want to include site-wide. If you have multiple layouts in here, you can easily switch between them for your views.
- `Elements/` contains what I like to think of as sub-or micro-views. These are smallish bits of PHP and HTML that you can include inside your other views. This is useful for list elements and other repeated bits of markup that you don't want to write over and over again.
- `Themed/` is a special directory. Instead of containing `.ctp` files directly, it contains folders that contain a new structure of views inside of them. The application can switch between these folders.

Views – a quick overview

Going from our earlier example, where `http://localhost/glossaryterms/add` renders `Glossaryterms/add.ctp`, we can notice that `add.ctp` doesn't contain all the markups that you see rendered to the page. What's actually happening is that `Layouts/default.ctp` loads first, as the "frame," and then `add.ctp` is inserted inside it.

Here's the `default.ctp` file:

```
$cakeDescription = __d('cake_dev', 'CakePHP: the
rapid development php framework');
?>
<!DOCTYPE html>
<html>
<head>
  <?php echo $this->Html->charset(); ?>
  <title>
    <?php echo $cakeDescription ?>:
    <?php echo $title_for_layout; ?>
  </title>
  <?php
    echo $this->Html->meta('icon');
```

```
    echo $this->Html->css('cake.generic');

    echo $this->fetch('meta');
    echo $this->fetch('css');
    echo $this->fetch('script');
  ?>
</head>
<body>
  <div id="container">
    <div id="header">
      <h1><?php echo
$this->Html-
>link($cakeDescription,
'http://cakephp.org'); ?></h1>
    </div>
    <div id="content">

      <?php echo $this->Session->flash(); ?>

      <?php echo $this->fetch('content'); ?>
    </div>
    <div id="footer">
      <?php echo $this->Html->link(
          $this->Html->image('cake.power.gif',
array('alt' => $cakeDescription, 'border' =>
'0')),
          'http://www.cakephp.org/',
          array('target' => '_blank', 'escape' =>
false)
        );
      ?>
    </div>
  </div>
  <?php echo $this->element('sql_dump'); ?>
</body>
</html>
```

Look for the lines that contain `$this->fetch`. There are four of them, but we only want to pay attention to three:

- `$this->fetch('css')`: This loads your CSS files. There are a couple of files that are loaded by default, and if you want to add more then you first add a file to the `app/webroot/css` directory, and then add a line like this to your view (in this case `add.ctp`): `$this->Html->css('my_file')`; where `'my-file'` is replaced by the name of your file without the extension.
- `$this->fetch('scripts')`: As in the earlier case, add a file to `app/webroot/js` and then add `$this->Html->scripts('my_file')` to your view; where `'my-file'` is replaced by the name of your file without the extension.
- `$this->fetch('content')`: This is where your `add.ctp` file is loaded. As you can see, the other markup in `default.ctp` acts as

the "frame" into which your actual views are loaded. CakePHP will replace this line with the content of the relevant `.ctp` file from your views.

Installing the Cakestarter-Bootstrap theme

We could go through how to make a nice looking view step-by-step, but this book isn't about frontend development; this book is about getting up and running quickly and efficiently. One of the best ways to do this is to reuse the work that other people have done. CakePHP's theming feature allows us to do this.

The folks over at Drawr Design (http://drawrdesign.com) made a CakePHP theme based on Twitter's Bootstrap. I've forked this theme and modified it for our application. Installing it is relatively simple:

1. Download the zipped version of the repository at https://github.com/aphelionz/Cakestarter-Bootstrap and save it into a folder called `Bootstrap`.
2. Visit `https://github.com/aphelionz/Cakestarter-Bootstrap` and click on the **ZIP** button towards the center of the page.
3. Unzip the file, and place its contents in your `app/Views/Themed` folder.
4. Modify the class in `app/Controller/AppController.php` so that it looks like the following code:

```
class AppController extends Controller {

  public $theme = "Cakestarter-Bootstrap";

  public function index() {
  }

  public function components() {

  }

  public function base_css() {

  }

  public function javascript() {
  }

}
```

5. Refresh your app, and enjoy your new theme!

Now, look inside of the `Bootstrap` folder. You will notice that there is a directory structure that mirrors your views structure. These folders act as

overrides to their mirrored counterparts. Whenever the `$theme` variable is set in a controller, the corresponding theme will override your base theme.

You may be unfamiliar with how overrides work, so I will explain them. When a theme is installed, the application first looks in the theme's folder for the view that it wants, which will be something like `/path/to/theme/Categories/view.ctp`. If it doesn't find it there, then it goes to the regular old view that exists in `app/Categories/view.ctp`.

Because your custom `Controller` classes extend `AppController`, setting the `$theme` variable there changes the theme site-wide.

To deploy your code to production, on your command line, run the following command:

```
af update [name]-cakestarter
```

Better URLs

We will be improving upon our URLs so that they contain readable words and not numbers. `/categories/view/animals` is better than `/categories/view/4` for accessibility, SEO, and so much more. To accomplish this, we will need:

- An extra field in our database tables to contain the slug
- Functionality that automatically generates our slugs based on titles whenever a model is saved
- Some controller logic that processes a slug from the URL instead of a number
- Finally, we will need to alter the views so that our links go to `/url/slug` instead of `url/123`

This is complex stuff, but again we are going to obtain a tremendous amount of help from both CakePHP's functionality and the community.

Altering the schema

Let's edit `app/Config/Schema/Glossary.php`. We're going to add one line to `$glossaryterms` and `$categories`, as shown in the following code:

```
  public $glossaryterms = array(
      'id' => array('type' => 'integer', 'null'
=> false, 'key' => 'primary'),
      'title' => array('type' => 'string', 'null'
=> false, 'length' => 100),
      'definition' => array('type' => 'string',
'null' => false, 'length' => 512),
```

```
        'slug' => array('type' => 'string', 'null'
=> false, 'length' => 100)
    );

    public $categories = array(
        'id' => array('type' => 'integer', 'null'
=> false, 'key' => 'primary'),
        'name' => array('type' => 'string', 'null'
=> false, 'length' => 100),
        'slug' => array('type' => 'string', 'null'
=> false, 'length' => 100)
    );
```

The additions are the lines that start with `'slug'`. To update the database itself, go to the app/ directory on the command line and run:

```
Console/cake schema update glossary
```

We will see the following output:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------
-------------
App : app
Path: /path/to/app
---------------------------------------------------
-------------
Cake Schema Shell
---------------------------------------------------
-------------
Comparing Database to Schema...

The following statements will run.
ALTER TABLE `cake_starter`.`glossaryterms`
        ADD `slug` varchar(100) NOT NULL AFTER
`definition`;
ALTER TABLE `cake_starter`.`categories`
        ADD `slug` varchar(100) NOT NULL AFTER
`definition`;
ALTER TABLE
`cake_starter`.`glossaryterms_categories`
        ;
Are you sure you want to alter the tables? (y/n)
[n] > y

Updating Database...
glossaryterms updated.
categories updated.
glossaryterms_categories updated.
End update.
```

Your database has been updated to contain the new fields. You can kick the tires on the app if you want; nothing should have broken.

Installing the sluggable behavior

Behaviors are wonderful, pluggable little bits of functionality that hook into your models. They save, create, and delete events in order to add extra features. They are quick and easy ways to "mix in" functionalities to your app, and also to share functionality between applications and developers.

In our case, we want our models to auto-generate URL-friendly strings of text, or "slugs" to our data.

In other words, whenever a model is saved or created, it will create a slug based on the `title_field` setting, name in `Category`, and title in `GlossaryTerm`. For example, if your term is "Hello World", it will create a slug like "hello_world".

To enable sluggable behavior, we use the following steps:

1. Visit https://raw.github.com/vduglued/CakePHP-Sluggable-Behavior/master/sluggable.php, copy the whole file, and paste it into a new file in your app, named `app/Models/Behavior/Sluggable.php`.
2. Update `app/Model/Category.php` by adding the following code:

```
app/Model/Category.php
  /**
   * Enable sluggable behavior
   *
   * @var string
   */
  var $actsAs = array(
      'Sluggable' => array(
          'title_field' => 'name'
      )
  );
```

3. Update `app/Model/GlossaryTerms.php` by using the following code:

```
/**
  * Enable sluggable behavior
  *
  * @var string
  */
 var $actsAs = array(
     'Sluggable' => array(
         'title_field' => 'title'
     )
```

```
);
```

This activates the downloaded behavior in your models. Because the behavior is set to generate a slug on creation or save, you can recreate the slugs in your existing records by saving each record that you have created, without changing anything on the edit page.

Making your controllers aware of the slugs

Let's say we have a glossary term called "Example Term". The auto-generated slug would be example_slug, and our URL would be `http://localhost/glossaryterms/view/example_slug`. However, when we view this URL we see the `Invalid Glossaryterm` error.

We need to add two lines to our controllers to make them aware of this change. We use the following steps to do so:

1. Inside `app/Controllers/GlossarytermsController.php` add the highlighted lines, and also change the argument from `$id` to `$slug`:

```
  public function view($slug = null) {
    $glossary_term =
$this->Glossaryterm->findBySlug($slug);
    $id =
$glossary_term['Glossaryterm']['id'];

    $this->Glossaryterm->id = $id;
    if (!$this->Glossaryterm->exists()) {
    throw new
NotFoundException(__('Invalid
glossaryterm'));
    }
    $this->set('glossaryterm',
$this->Glossaryterm->read(null, $id));
  }
```

2. Inside `app/Controllers/CategoriesController.php` add the highlighted lines, and also change the argument from `$id` to `$slug`:

```
  public function view($slug= null) {
    $category =
$this->Category->findBySlug($slug);
    $id = $category['Category']['id'];

    $this->Category->id = $id;
    if (!$this->Category->exists()) {
      throw new
NotFoundException(__('Invalid category'));
    }
```

```
        $this->set('category',
    $this->Category->read(null, $id));
      }
```

Everything else here was written when we originally generated these controllers. The two lines that we added simply intercept the assignment of the `$id` variable and then let the controller carry on its work.

Updating the views

Now, go through your views and look for instances of `$this->Html->link` that contain:

- `$category['Category']['id']`
- `$glossaryterm['Glossaryterm']['id']`
- `$category['id']`
- `$glossaryterm['id']`

Respectively, change them to:

- `$category['Category']['slug']`
- `$glossaryterm['Glossaryterm']['slug']`
- `$category['Category']['slug']`
- `$glossaryterm['Glossaryterm']['slug']`

I'm hesitant to include code samples, or even line numbers, here as you may have been playing with the HTML in the views from the previous steps, and the theme may have changed since this book was published. That being said, we can still explore how this works.

Begin in your controller. The `$this->set` function defines a variable to be used in the view:

```
$this-
>set('category', $this->Category->read(null,
$id));
```

In the view, this will be represented as `$category`. For example, you may see something like:

```
<td><?
php echo $category['Category']['id']; ?></td>
```

The `$category` variable was defined in the controller. Then, the first array index is the model name. In this instance it is `Category`. The next index is

the field name. This syntax will become very familiar to you as you continue to develop with CakePHP.

To deploy this to production, you will have to run `af update [yourname]-cakestarter` and run the database migration steps given at the end of the *Installation* section.

Adding the Install page

We've got a lot going on in our app right now. It is fully functional, and now has a much nicer look and feel to it. But we're not done yet. Remember, we needed a JavaScript component that will recognize these words on other pages, and highlight them. Luckily, I've included one in the theme you downloaded!

> **Note**
>
> The JavaScript file itself is included in this theme. Because this book is not meant to cover JavaScript, a thorough discussion of the script itself is not included. However, I have left the file commented, and I plan to post a discussion on my blog at http://markroberthenderson.com.

Using the built-in PagesController

CakePHP has a very convenient built-in controller called **PagesController**. This lives in `app/Controller`, happily mapping URL strings to the names of the `.ctp` files in `app/View/Pages`.

In our case, we need to create `http://localhost/cake-starter/pages/install`. Type that URL into your browser and note the error message that appears. Note that CakePHP's standard error messages are as helpful as they can be. Read it carefully and note that it tells you exactly what you need to do.

Create a file called `install.ctp` and insert the following code:

```
<section id="typography">
  <div class="page-header">
    <h1>Installation <small>Empowering Your Site
with the Lakota Language Glossary</small></h1>
  </div>

  <!-- Headings & Paragraph Copy -->
  <div class="row">
    <div class="span5">
      <div class="well">
        <h2>Only 2 Steps:</h2>
```

```
        <ol>
          <li>
            <h3>Add this HTML Tag</h3>
            <code>
              &lt;script
src="/path/to/file.js"&gt;&lt;/script&gt;
            </code>
            <p>
              Installation is simple. Just copy
and paste this one line of HTML
              and paste it just before the
closing</body> tag of your page:
            </p>
          </li>
          <li>
            <h3>Mark Up Your Page</h3>
            <code>
              &lt;span class="glossary"
id="23"&gt;hecinska&lt;/span&gt;
            </code>
            <p>
              In the HTML of your page, put
&lt;span&gt;tags around the word you want to
highlight.
              It needs to have a class of
'glossary' and arel attribute that contains a url
from the glossary.
            </p>
          </li>
        </ol>
      </div>
    </div>
    <div class="span7">
      <h2>Example</h2>
      <p>
        This paragraph contains a word that has
been given the HTML treatment described in the
<span class="glossary"
id="13">instructions</span>to the left. The word
has been highlighted and when you move your
cursor over the word, the definition and a link
to its page in the glossary is displayed.Follow
the instructions on the left toinstall it on your
own site!
      </p>
    </div>
  </div>
</section>
```

Now, refresh the URL. The error is gone, and you have a nice looking page with instructions on how to use the software.

There actually used to be a tutorial based solely around creating CakePHP apps by following and fixing error messages until a complete application

emerged. However, a quick Google search coming up empty makes me think that this is not the recommended way of creating an application.

Creating a JSON service for the JavaScript to consume

When you look at `http://localhost/cake-starter/install` and `http://localhost/cake-starter/example` and open up your console's JavaScript, you will see that the AJAX request is receiving a 404 error.

Open up `app/Controller/GlossarytermsController.php` and add the following function to the class definition:

```
  /**
  * Function to return json Category and
Glossaryterms objects from a comma-separated
string of ids
  *
  * @param string $ids
  */
  public function json($ids)
  {
    $this->layout = 'ajax';
    $glossaryterms = array();

    foreach(explode(',', $ids) as $id)
    {
      $glossaryterms[] =
$this->Glossaryterm->findById($id);
    }

    $this->set('glossaryterms', $glossaryterms);
  }
```

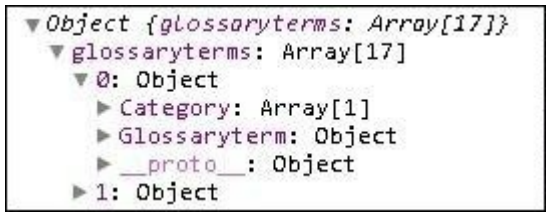
The most important line for our purposes is `$this->layout = 'ajax'` (which is highlighted preceding code). This tells the function to use the `ajax.ctp` file in `app/Views/Layout` instead of `default.ctp`. The `ajax.ctp` layout is a minimal layout made for XML or JSON output.

From there, this code reads in `Glossaryterm` object models from the query string, stores them all in an array, and passes them into a view.

Create a file at `app/Views/Glossaryterms/json.ctp`. The code for this file is very simple:

```
<?php
  echo json_encode($glossaryterms);
?>
```

Now visit those pages again and check out a few things that have changed. First of all, we have not only the `Glossaryterm` objects on the client side, but we also get their associated Categories for free! Of course you'll want to keep an eye on this to prevent bloat, but for now this is working out in our favor.



Second, and most importantly, the JavaScript should have worked in updating the UI, and the terms on the pages should be popup-enabled. Pretty cool!

Don't forget to run `af update` to push this code to your server!

Fixtures and automated tests

Now our scaffolding is turning into something that resembles a real application, we should take the necessary steps to make sure that it's going to work how we expect it to. This is accomplished with either the Cake command-line tool or the browser. We will cover both methods in this section.

What is automated testing?

Automated testing is the use of software to run a series, or a "battery" of tests, against other code that you or somebody else has written.

This is incredibly useful in assuring the quality of your code, and provides a sort of safety net that ensures your application code will still continue to function after a refactor. You can write code, write tests, refactor, and then run the tests again to ensure that what you expect to happen will still happen.

Automated testing is something that can take up not one, but several other books, so we'll just stick to CakePHP-specific things.

The initial setup

Before we dive into the initial set-up that we need to make sure a few initial things are in place, namely PHPUnit and a separate testing database for our automated tests.

Installing PHPUnit

This section requires PHPUnit, the de facto standard for PHP-based automated testing. You can download and install PHPUnit using PEAR, which is a PHP package management system. The *CakePHP Cookbook 2.x* has the following steps, once PEAR is installed:

```
pear upgrade PEAR
pear config-set auto_discover 1
pear install pear.phpunit.de/PHPUnit
```

> **Note**
>
> I ran into some issues of my own trying to install PHPUnit. Unfortunately it's not as easy as it should be. Make sure that the `PHPUnit` folder is in your install path and that pear is at the required version using `pear upgrade PEAR`. Also, make sure that you restart Apache or any other web server software you might be using.
>
> If you can't figure it out, try Googling it. I know that's a horrible thing to put in an instruction manual, but I'm afraid that if I put more detailed instructions here, they would be outdated by the time the book is published.

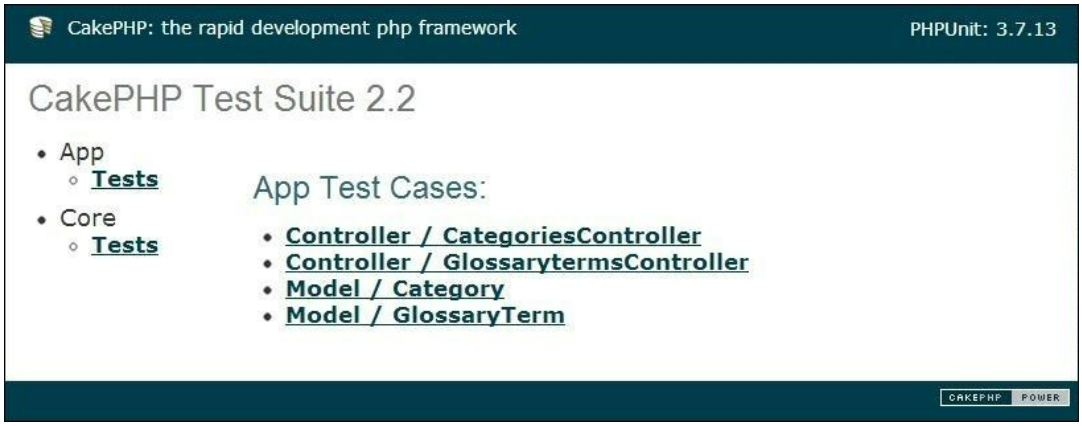
Creating your test database

The *Cake Cookbook 2.x* says it's a "good idea" to make your test database different to your actual database. I'm telling you here that this is NOT an option. Testing adds, deletes, and changes data and you simply cannot take that sort of risk with your data. Not even a little. Seriously.

Create a new database called something like `cakestarter_test`, and then update your `app/Config/database.php` file and copy the same settings that you have for your `$dev` database configuration to the `$test` configuration. However, change the name of the database to a value with `_test` appended to the end.

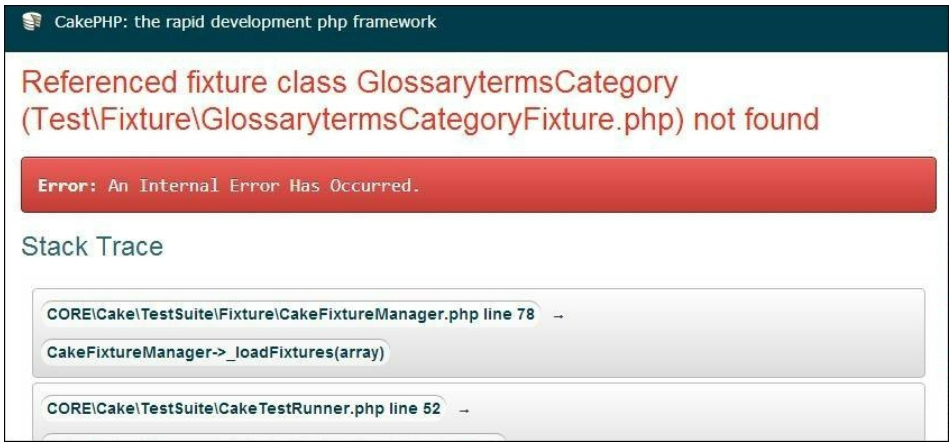
On to the good stuff!

Now, visit `http://localhost/cakestarter/test.php` in your browser. You should see the following:



CakePHP provides an excellent web-based interface for running your tests. Feel free to click around in here and see if you can glean anything useful. It probably won't take you long to find some errors in this interface. Some of them are our fault, and some of them aren't, but all of them are things that we will clean up.

1. First, click on **Controller** | **CategoriesController** and we will notice our first error:



   To clean this up, simply run the `bake` tool and create the missing fixture. See the command line trace as follows:

```
Console\cake bake
```

   We will get the following output:

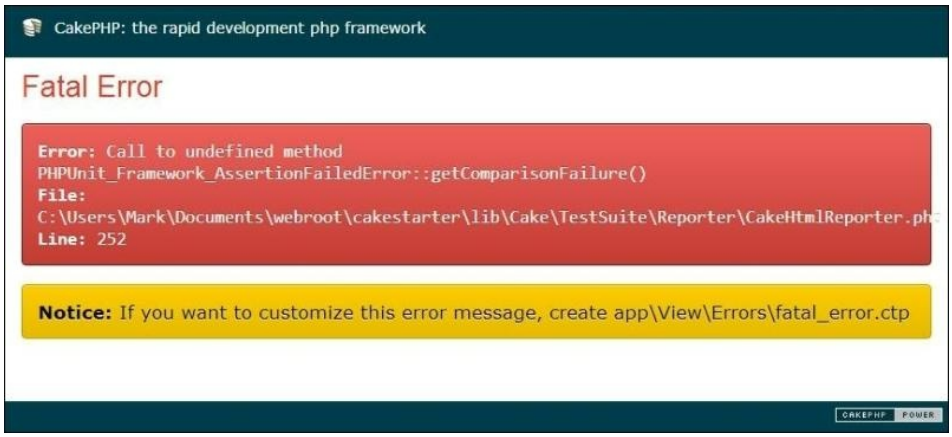
```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------------------------
App : app
Path: path/to/app
```

```
---------------------------------------------
--------------------
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)
> F
---------------------------------------------
--------------------
Bake Fixture
Path:
C:\Users\Mark\Documents\webroot\cakestarter
\app\Test\Fixture\
---------------------------------------------
--------------------
Use Database Config: (default/dev/test)
[default] >
Possible Models based on your current
database:
1. Category
2. Glossaryterm
3. GlossarytermsCategory
Enter a number from the list above,
type in the name of another model, or 'q'
to exit
[q] > 3
Would you like to import schema for this
fixture? (y/n)
[n] >
Would you like to use record importing for
this fixture? (y/n)
[n] >
Would you like to build this fixture with
data from GlossarytermsCategory's tabl
e? (y/n)
[n] >

Baking test fixture for
GlossarytermsCategory...
```

That should clear up that error, and the tests should run properly now. Let's move on.

2. We can see the next error by going back to `test.php` and clicking on **Controller** | **GlossarytermsController**.



This one isn't our fault; the `bake` tool has created test cases without any tests in them. Although the fatal error here doesn't mention this, we can easily remedy this by running the `bake` tool again.

```
Console\cake bake
```

We get the following output:

```
Welcome to CakePHP v2.2.3 Console
---------------------------------------------
---------------------
App : app
Path:
C:\Users\Mark\Documents\webroot\cakestarter
\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)
> T
---------------------------------------------
---------------------
Bake Tests
```

```
Path:
C:\Users\Mark\Documents\webroot\cakestarter
\app\Test\
---------------------------------------------
--------------------
---------------------------------------------
--------------------
Select an object type:
---------------------------------------------
--------------------
1. Model
2. Controller
3. Component
4. Behavior
5. Helper
Enter the type of object to bake a test
for or (q)uit (1/2/3/4/5/q)
[q] > 2
Choose a Controller class
1. AppController
2. CategoriesController
3. GlossarytermsController
4. PagesController
Choose an existing class, or enter the
name of a class that does not exist
> 3
Bake is detecting possible fixtures...

Baking test case for Glossaryterms
Controller ...
```

Next, open up `app/Test/Case/Controller/GlossarytermsControllerTest.php`. You should see that it has five functions: `testView`, `testAdd`, `testEdit`, `testDelete`, and `testJson`. They're empty right now, which means that they'll pass, no matter what. This is not very useful to us, so you're going to want to fill them in with code that runs against your controller.

Unfortunately, I have limited space and time in this book to talk about such things. For more information on how to do this, check out http://book.cakephp.org/2.0/en/development/testing.html.

You can also run tests from the command line by using `cake test`. The following commands are some examples.

- `Console\cake test app Controller/GlossarytermsController`
- `Console\cake test app Controller/CategoriesController`
- `Console\cake test app Model/GlossaryTerm`
- `Console\cake test app Model/Category`

The syntax here should be pretty self explanatory. In the third example, you're testing the `Glossaryterm` model of your app. Feel free to re-run bake as necessary if you get any errors related to missing tests.

And, of course, once you've written some good tests, run `af update` to push your code to production. Once your tests run successfully in production, you'll want to remove `test.php` from the production server. Bonus points if you make this part of your build or continuous integration process!

People and places you should get to know

Many of the things we've accomplished in the last section relied on not only on the features of CakePHP, but also on the community associated with the project. We found ourselves installing third-party plugins and building upon the work of others. This is one of the benefits of open source software: being able to build upon foundations that others have laid out.

The Bakery

The CakePHP Bakery, located at http://bakery.cakephp.org, is the site put forth by the CakePHP foundation to foster exactly this type of communication.

If you visit http://bakery.cakephp.org/categories, you can see that it is easy to find "recipes" both by concept and by specific class types: models, helpers, plugins, and so on. There is also http://plugins.cakephp.org that allows you to find, contribute, and browse plugins as well.

Getting support from the community

Often times, the answers you seek are not readily available in the documentation or even in the Bakery. You will have to delve into and eventually join the community to find what you seek. The best place to start is the official CakePHP community site, at http://community.cakephp.org.

IRC (Internet Relay Chat) channels

The IRC protocol still persists even though it is several decades old. The CakePHP developers maintain active channels on the popular open source network FreeNode. Once you've connected to irc.freenode.net with your favorite IRC client, you can join one of the following channels to get help:

- `#cakephp`: This is where you can go if you have a general question while you're trying to build a CakePHP application
- `#cakephp-docs`: Here, you can discuss, ask questions, or otherwise provide suggestions about the CakePHP documentation
- `#cakephp-bakery`: This is similar to the earlier channel, except it relates to the Bakery

The Google group, Google Plus page, and CakePHP questions

If IRC is unavailable to you for any reason, there are three more places to which you can go for support:

- http://groups.google.com/group/cake-php: This is an active Google group with a great archive of previously-answered questions
- https://plus.google.com/communities/108328920558088369819: This is the official Google Plus community page for CakePHP
- http://ask.cakephp.org/: This is a Stack Overflow-like interface for obtaining and selecting answers for your application development questions

CakePHP on social media

Another thing that you can do is follow the CakePHP developers on social media sites, such as Twitter or Facebook. This section mentions some of their accounts.

Twitter

Several of the CakePHP developers are active on Twitter and are worth following. You can start with `@CakePHP` and `@CakeDC`, but sometimes it's nice to delve a little deeper:

- `@phpnut`: Larry E. Masters, CakePHP founder/lead developer
- `@mark_story`: Mark Story, CakePHP core developer, often found on the FreeNode IRC channels
- `@ad7six`: Andy Dawson, CakePHP contributor
- `@jose_zap`: Jose Zap, CakePHP Core developer

Facebook

Although the community on Facebook isn't as strong as it is in other places on the Internet, you can still get help by visiting http://www.facebook.com/groups/cake.community/.

Conventions and meetups

Sometimes, the best thing to do to build and participate in a community is to get out into the real world and interface with real people in real time. CakePHP has a number of events where you can go and meet other users, developers, and enthusiasts.

CakeFest

Although other PHP conventions will have sessions devoted to PHP in them, CakeFest is a CakePHP specific convention that happens once a

year, in locations all over the world. You can find more information at http://cakefest.org/.

Meetups

Meetup groups blur the line between social media on the Internet, bringing together people with common interests. CakePHP has meetups in several cities around the world. Visit http://meetup.com and perform a global search for CakePHP and you will hopefully see one in your area.

Parting words

I hope you've enjoyed this short Starter book! Remember that the best thing you can do to learn how to build CakePHP apps is to...build CakePHP apps. Each application that build will provide a unique set of challenges that will build your experience, your skill set, and ultimately your career.

Remember to try and make your helpers, behaviors, models, components, and themes as reusable and as modular as possible. Because CakePHP's code is so structured, you WILL be able to use several parts of each app that you build in other apps, and maybe people will be downloading your code from the Bakery before long.

Good luck, and build great things for all of us to see!