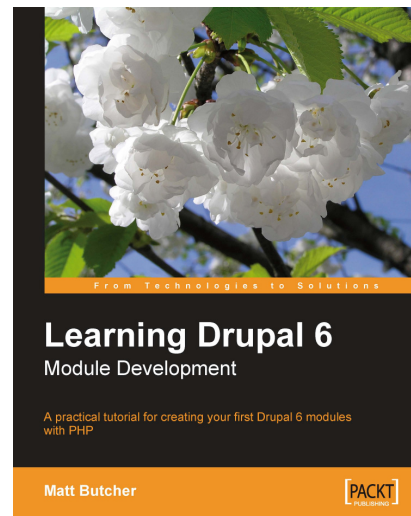




Learning Drupal 6 Module Development

Matt Butcher



Chapter No. 2 "Creating Our First Module"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Creating Our First Module"

A synopsis of the book's content

Information on where to buy this book

About the Author

Matt Butcher is the principal consultant for Aleph-Null, Inc. (<http://aleph-null.tv>), where he specializes in content management systems, Linux system integration, and Open Source technologies. He has been an active participant in open-source technologies for over a decade. Along with Learning Drupal 6, Matt has also written Mastering OpenLDAP, Managing and Customizing OpenCms 6, and Building Websites with OpenCms, all of which are published by Packt. When not pushing bits, Matt likes to explore Colorado with his wife and three daughters.

The Drupal community has not only been a boundless source of information, but also a positive environment. This community is to be commended for creating a successful habitat for growing a top-quality application. Writing this book has involved a veritable army of editors, technical reviewers, and proofreaders. This book has benefited tremendously from their hard work. I would like to thank Douglas Patterson not only for being my editor, but for getting me involved with Drupal in the first place. Thanks to Patricia Weir and Swapna V. Verlekar, who have worked tirelessly on the book. I owe a huge debt of gratitude to John Forsythe and Jason Flatt, whose meticulous reading and copious suggestions have had a profound influence on the final state of this book. I'd also like to thank: Edward Peters, David Norman, and Sherif for their invaluable suggestions. Thanks also to the many individuals at Drupalcon 2008 who provided input and who took the time to chat with me about Drupal. Finally, thanks to Angie, Katherine, Claire, and Annabelle for putting up with a few late nights and some occasional whining.

For More Information:

www.packtpub.com/drupal-6-module-development/book

Learning Drupal 6 Module Development

Drupal is a highly successful open-source Content Management System (CMS). It is well-respected for its robustness, its flexible and immaculate code, and its seemingly infinite capacity for extension and customization.

Drupal 6, released early in 2008, represents a significant evolution in this already mature CMS. In this book, we build extensions for Drupal 6, focusing on the important APIs and libraries. We also highlight the new features introduced in version 6, making this book appropriate not only for those new to Drupal, but also those who are transitioning from version 5.

This book provides a practical, hands-on approach to developing Drupal modules. We also take a developer-centered look at themes and installation profiles—two other facets of Drupal that the developer should be familiar with. Each chapter focuses on the creation of a custom extension. Using this approach we develop a handful of modules, a theme, and an installation profile. However, more importantly, we get a practical perspective on how to make the most of Drupal.

By the end of the book, you will have a solid understanding of how to build modules for Drupal. With the knowledge of foundational APIs and libraries, you will be able to develop production-quality code that fully exploits the power and potential of Drupal 6.

For More Information:

www.packtpub.com/drupal-6-module-development/book

What This Book Covers

This book focuses on developing modules for Drupal 6. Each chapter introduces new concepts, libraries and APIs, while building on material from previous chapters.

Chapter 1 is a developer's introduction to Drupal. We take a look at Drupal's architecture, focusing on modules and themes. After covering some of the important concepts and taking a high-level look at foundational APIs and libraries, we finish up with a look at some useful development tools.

Chapter 2 gets us working on our first module. In this chapter, we develop a Drupal module that takes data from an XML feed and displays it as a block on our Drupal site. In this chapter, you will learn about the basics of module development, including what files need to be created and where they go. Hooks, a major component of Drupal development, are also introduced here.

Chapter 3 switches gears from modules to themes. Learning the theming system is integral to being able to produce high-quality modules. In this chapter, we create a theme using CSS, HTML, and PHPTemplates. We also take a look at the theme system architecture, along with some of the APIs.

Chapter 4 builds on the introduction to theming. In this chapter, we develop a new module that deals with a custom content type, a quote. This module uses the theming subsystem to prepare quotes for display. Our focus here is using the theme system to enrich modules. The theme API covered in this chapter is used throughout the rest of the book.

Chapter 5 focuses on Drupal's JavaScript libraries. Starting with the module we built in Chapter 4, we use the jQuery library and a couple of Drupal hooks to implement an AJAX (Asynchronous JavaScript and XML) service. This chapter also introduces the Database API and the menu system.

Chapter 6 is focused on building an administration module. This module provides an interface for administrators to send email messages to users. However, the Mail API is not the only thing we will look at. The tremendously important Forms API is also introduced here. We also get our first look at Drupal's access control features.

Chapter 7 takes a closer look at Drupal nodes. In this chapter, we use the Schema API, the Database API, and the node system to build a content type that represents a biography. Module installation files are introduced, and the Forms API and access control mechanisms are revisited.

Chapter 8 discusses filters, actions, and hooks—three of the more advanced features of Drupal. We create a module for emailing a newsletter to our users. We implement filters to prepare content for the email message. Actions and triggers are used to automatically send our newsletter when it is ready. Also, to allow other modules to interact with this one, we define our own custom hook that other modules can implement.

Chapter 9 changes tracks, focusing on installation profiles. In this chapter, we build an installation profile that can install a custom version of Drupal preloaded with the modules and themes of our choice. Working with the installer, we get a glimpse into Drupal's inner workings. Along with learning how to write code in this minimalistic pre-installation environment, we also look at registering themes and defining triggers automatically.

For More Information:

www.packtpub.com/drupal-6-module-development/book

2

Creating Our First Module

In the last chapter, we looked at the basics of Drupal module development. Now we will dive in and create our first module. Our first module will make use of an existing web service to pull in some XML data, format it, and display it as a block in the site's layout.

We will cover the following topics in this chapter:

- Creating the `.info` and `.module` files
- Creating a new module
- Using basic hooks
- Installing and configuring the module
- Using important Drupal functions

Starting Out

Our first module is going to fetch XML data from Goodreads (<http://www.goodreads.com>), a free social networking site for avid readers. There, users track the books they are reading and have read, rate books and write reviews, and share their reading lists with friends.

Reading lists at Goodreads are stored in **bookshelves**. These bookshelves are accessible over a web-based XML/RSS API. We will use that API to display a reading list on the Philosopher Bios website we introduced in Chapter 1.

To integrate the Goodreads information in Drupal, we will create a small module. Since this is our first module, we will get into greater details, since they will be commonplace in the later chapters.

For More Information:

www.packtpub.com/drupal-6-module-development/book

A Place for the Module

In Drupal, every module is contained in its own directory. This simplifies organization; all of the module's files are located in one place.

To keep naming consistent throughout the module (a standard in Drupal), we will name our directory with the module name. Later, we will install this module in Drupal, but for development, the module directory can be wherever it is most convenient.

Once we have created a directory named `goodreads`, we can start creating files for our module. The first file we need to create is the `.info` (dot-info) file.

Creating a .info File

Before we start coding our new module, we need to create a simple text file that will hold some basic information about our module. Various Drupal components use the information in this file for module management.

The `.info` file is written as a PHP INI file, which is a simple configuration file format.



If you are interested in the details of INI file processing, you can visit <http://php.net/manual/en/function.parse-ini-file.php> for a description of this format and how it can be parsed in PHP.

Our `.info` file will only be five lines long, which is probably about average.

The `.info` file must follow the standard naming conventions for modules. It must be named `<modulename>.info`, where `<modulename>` is the same as the directory name. Our file, then, will be called `goodreads.info`.

Following are the contents of `goodreads.info`:

```
;$Id$
name = "Goodreads Bookshelf"
description = "Displays items from a Goodreads Bookshelf"
core = 6.x
php = 5.1
```

This file isn't particularly daunting. The first line of the file is, at first glance, the most cryptic. However, its function is mundane: it is a placeholder for Drupal's CVS server.

Drupal, along with its modules, is maintained on a central **CVS (Concurrent Version System)** server. CVS is a version control system. It tracks revisions to code over time. One of its features is its ability to dynamically insert version information into a file. However, it needs to know where to insert the information. The placeholder for this is the special string `Id`. But since this string isn't actually a directive in the `.info` file, it is commented out with the PHP INI comment character, `;` (semi-colon).



You can insert comments anywhere in your `.info` file by beginning a line with the `;` character.

The next four directives each provide module information to Drupal.

The `name` directive provides a human-readable display name for the module. In the last chapter, we briefly discussed the Drupal module installation and configuration interface. The names of the modules we saw there were extracted from the `name` directive in their corresponding `.info` files. Here's an example:

▼ Core - optional			
Enabled	Name	Version	Description
<input type="checkbox"/>	Aggregator	6.0-rc3	Aggregates syndicated content (RSS, RDF, and Atom feeds). (Code Review)
<input type="checkbox"/>	Blog	6.0-rc3	Enables keeping easily and regularly updated user web pages or blogs. (Code Review)

In this above screenshot, the names **Aggregator** and **Blog** are taken from the values of the `name` directives in these modules' `.info` files.

While making the module's proper name short and concise is good (as we did when naming the module directory `goodreads` above), the display name should be helpful to the user. That usually means that it should be a little longer, and a little more descriptive.

However, there is no need to jam all of the module information into the `name` directive. The `description` directive is a good place for providing a sentence or two describing the module's function and capabilities.

The third directive is the `core` directive.



The `core` and `php` directives are new in Drupal 6.

This directive specifies what version of Drupal is required for this module to function properly. Our value, `6.x`, indicates that this module will run on Drupal 6 (including its minor revisions). In many cases, the Drupal packager will be able to automatically set this (correctly). But Drupal developers are suggesting that this directive be set manually for those who work from CVS.

Finally, the `php` directive makes it possible to specify a minimum version number requirement for PHP. PHP 5, for example, has many features that are missing in PHP 4 (and the modules in this book make use of such features). For that reason, we explicitly note that our modules require at least PHP version 5.1.

That's all there is to our first module `.info` file. In later chapters, we will see some other possible directives. But what we have here is sufficient for our Goodreads module.

Now, we are ready to write some PHP code.

A Basic `.module` File

As mentioned in the first chapter, there are two files that every module must have (though many modules have more). The first, the `.info` file, we examined above. The second file is the `.module` (dot-module) file, which is a PHP script file. This file typically implements a handful of hook functions that Drupal will call at predetermined times during a request.



For an introduction to hooks and hook implementations, see the previous chapter.



Here, we will create a `.module` file that will display a small formatted section of information. Later in this chapter, we will configure Drupal to display this information to site visitors.

Our Goal: A Block Hook

For our very first module, we will implement the `hook_block()` function. In Drupal parlance, a block is a chunk of auxiliary information that is displayed on a page alongside the main page content. Sounds confusing? An example might help.

Think of your favorite news website. On a typical article page, the text of the article is displayed in the middle of the page. But on the left and right sides of the page and perhaps at the top and bottom as well, there are other bits of information: a site menu, a list of links to related articles, links to comments or forums about this article, etc. In Drupal, these extra pieces are treated as blocks.

The `hook_block()` function isn't just for displaying block contents, though. In fact, this function is responsible for displaying the block and providing all the administration and auxiliary functions related to this block. Don't worry... we'll start out simply and build up from there.

Starting the .module

As was mentioned in the last chapter, Drupal follows rigorous coding and documentation standards (<http://drupal.org/coding-standards>). In this book, we will do our best to follow these standards. So as we start out our module, the first thing we are going to do is provide some API documentation.

Just as with the `.info` file, the `.module` file should be named after the module. Following is the beginning of our `goodreads.module` file:

```
<?php
// $Id$
/**
 * @file
 * Module for fetching data from Goodreads.com.
 * This module provides block content retrieved from a
 * Goodreads.com bookshelf.
 * @see http://www.goodreads.com
 */
```

The `.module` file is just a standard PHP file. So the first line is the opening of the PHP processing instruction: `<?php`. Throughout this book you may notice something. While all of our PHP libraries begin with the `<?php` opening, none of them end with the closing `?>` characters.

This is intentional, in fact, it is not just intentional, but conventional for Drupal. As much as it might offend your well-formed markup language sensibilities, it is good coding practice to omit the closing characters for a library.

Why? Because it avoids printing whitespace characters in the script's output, and that can be very important in some cases. For example, if whitespace characters are output before HTTP headers are sent, the client will see ugly error messages at the top of the page.

After the PHP tag is the keyword for the version control system:

```
// $Id$
```

When the module is checked into the Drupal CVS, information about the current revision is placed here.

The third part of this example is the API documentation. API documentation is contained in a special comment block, which begins `/**` and ends with a `*/`. Everything between these is treated as documentation. Special extraction programs like Doxygen can pull out this information and create user-friendly programming information.



The Drupal API reference is generated from the API comments located in Drupal's source code. The program, Doxygen, (<http://www.stack.nl/~dimitri/doxygen/>) is used to generate the API documents from the comments in the code.

The majority of the content in these documentation blocks (docblocks, for short) is simply text. But there are a few additions to the text.

First, there are special identifiers that provide the documentation generating program with additional information. These are typically prefixed with an `@` sign.

```
/**
 * @file
 * Module for fetching data from Goodreads.com.
 * This module provides block content retrieved from a
 * Goodreads.com bookshelf.
 * @see http://www.goodreads.com
 */
```

In the above example, there are two such identifiers. The `@file` identifier tells the documentation processor that this comment describes the entire file, not a particular function or variable inside the file. The first comment in every Drupal PHP file should, by convention, be a file-level comment.

The other identifier in the above example is the `@see` keyword. This instructs the documentation processor to attempt to link this file to some other piece of information. In this case, that piece of information is a URL. Functions, constants, and variables can also be referents of a `@see` identifier. In these cases, the documentation processor will link this docblock to the API information for that function, constant, or variable.

As we work through the modules in this book, we will add such documentation blocks to our code, and in the process we will encounter other features of docblocks.

With these formalities out of the way, we're ready to start coding our module.

The `hook_block()` Implementation

Our module will display information inside a Drupal block. To do this, we need to implement the `hook_block()` function.

Remember, what we are doing here is providing a function that Drupal will call. When Drupal calls a `hook_block()` function, Drupal passes it as many as three parameters:

- `$op`
- `$delta`
- `$edit`

The `$op` parameter will contain information about the type of operation Drupal expects the module to perform. This single hook implementation is expected to be able to perform a variety of different operations. Is the module to output basic information about itself? Or display the block? Or provide some administration information? The value of `$op` will determine this.

`$op` can have the following four possible values:

- `list`: This is passed when the module should provide information about itself. For example, when the list of modules is displayed in the module administration screen, the `$op` parameter is set to `list`.
- `view`: This value is passed in `$op` when Drupal expects the block hook to provide content for displaying to the user.
- `configure`: This value is passed when Drupal expects an administration form used to configure the block. We will look at this later.
- `save`: This value is passed when configuration information from the form data generated by `configure` needs to be saved.

The `$delta` parameter is set during a particular operation. When `$op` is set to the string `view`, which is the operation for displaying the block, then the `$delta` will also be set. `$delta` contains extra information about what content should be displayed. We will not use it in our first example, but we will use it later in the book. Take a look at Chapter 4 for another example of a `hook_block()` implementation.



Using deltas, you can define a single `hook_block()` function that can display several different blocks. For example, we might define two deltas—one that displays our Goodreads bookshelf, and the other that displays information about our Goodreads account. Which one is displayed will depend on which `$delta` value is passed into the `goodreads_block()` function. Other modules in this book will make use of deltas.

Finally, the `$edit` parameter is used during configuration (when the `save` operation is called). Since we are not implementing that operation in our first module, we will not use this parameter.



Drupal is meticulously documented, and the API documents are available online at <http://api.drupal.org>. More information about `hook_block()` parameters is available at this URL: http://api.drupal.org/api/function/hook_block/6.

All hook methods should follow the module naming convention: `<module name>_<hook name>`. So our goodreads block hook will be named `goodreads_block()`.

```
/**
 * Implementation of hook_block()
 */
function goodreads_block($op='list', $delta=0, $edit=array()) {
  switch ($op) {
    case 'list':
      $blocks[0]['info'] = t('Goodreads Bookshelf');
      return $blocks;
    case 'view':
      $blocks['subject'] = t('On the Bookshelf');
      $blocks['content'] = t('Temporary content');
      return $blocks;
  }
}
```

Following Drupal conventions, we precede the function with a documentation block. For hooks, it is customary for the documentation therein to indicate which hook it is implementing.

Next is our function signature: `function goodreads_block($op='list', $delta=0, $edit=array())`. The `$op`, `$delta`, and `$edit` parameters are all explained above. Each parameter is initialized to a default value. Here, we follow the customary defaults, but you can define them otherwise if you prefer.

As I mentioned earlier the `$op` parameter might be set to one of several different values.

What we do in this function is largely determined by which of those four values is set in the `$op` flag. For that reason, the first thing we do in this function is use a `switch` statement to find out which operation to execute.

Each case in the `switch` statement handles one of the different operations. For now, we don't have any administration configuration to perform, so there are no cases to handle either `configure` or `save` operations. We just need to handle the `list` and `view` operations. Let's look at each.

```
case 'list':
    $blocks[0]['info'] = t('Goodreads Bookshelf');
    return $blocks;
```

When Drupal calls this hook with `$op` set to `'list'`, then this module will return a two-dimensional array that looks as follows:

```
array(
  [0] => (
    'info' => 'Goodreads Bookshelf'
  )
)
```

Each element in this array is a **block descriptor**, which provides information about what this block implementation does. There should be one entry here for every `$delta` value that this function recognizes. Our block will only return one value (we don't make use of `deltas`), so there is only one entry in the block descriptor array.

A block descriptor can contain several different fields in the associative array. One is required: the `'info'` field that we have set above. But we could also provide information on caching, default weighting and placement, and so on.



For detailed information on this and other aspects of the `hook_block()` hook, see the API documentation: http://api.drupal.org/api/function/hook_block/6

Drupal uses the `'info'` field to display an item in the module management list, which we will see in the *Installing a Module* section of this chapter.

The t() Function

In this example, there is one more thing worthy of mention. We use the function `t()`. This is the **translation function**. It is used to provide multi-language support and also provide a standard method of string substitution. When `t()` is called, Drupal will check to see if the user's preferred language is other than the default (US English). If the user prefers another language, and that language is supported, then Drupal will attempt to translate the string into the user's preferred language.



For multi-language support, you will need to enable the **Content translation** module.

Whenever we present hard-coded text to a user, we will use the `t()` function to make sure that multi-language support is maintained.

In simple cases, the `t()` function takes just a string containing a message. In this case, the entire string will be translated. But sometimes, extra data needs to be passed into the string function. For example, we may want to add a URL into a string dynamically:

```
'Trying to access !url.'
```

In this case, we want `t()` to translate the string, but to substitute a URL in place of the `!url` placeholder. To do this, we would call `t()` with the following parameters:

```
t('Trying to access !url.', array('!url'=>'http://example.com'));
```

In this example, `t()` has two arguments: the string to translate, and an associative array where the key is the placeholder name and the value is the value to be substituted. Running the above when the locale is set to English will result in a string as follows:

```
Trying to access http://example.com.
```

There are three different kinds of placeholder. We have seen one above.

- `!:` Placeholders that begin with the exclamation point (`!`) are substituted into the string exactly as is.

Sometimes it is desirable to do some escaping of the variables before substituting them into the string. The other two placeholder markers indicate that extra escaping is necessary.

- `@:` Placeholders that begin with an `@` sign will be escaped using the `check_plain()` function. This will, for example, convert HTML tags to escaped entities. `t('Italics tag: @tag', array('@tag' => '<i>'))` will produce the string `'Italics tag: <i>'`.

- `%`: Placeholders that begin with the percent sign (`%`) are not only escaped, like those that start with the `@`, but are also themed. (We will look at theming in the next chapter.) Usually, the result of this theming is that the output value is placed in italics. So `t('Replacing %value.', array('%value=>'test'))` will result in something like `'Replacing test'`. The `` tags are added by the translation function.



Don't trust user-entered data

It is always better to err on the side of caution. Do not trust data from external sources (like users or remote sites). When it comes to the `t()` function, this means you should generally not use placeholders beginning with `!` if the source of the string to be substituted is outside of your control. (For example, it is inadvisable to do this: `t('Hello !user', array('!user' => $_GET['username']))`. Using `@user` or `%user` is safer.

We will use the `t()` function throughout this book. For now, though, let's continue looking at the `hook_block()` function we have created.

A view Operation

Now let's turn to the `view` case. This second case in our `switch` statement looks as follows:

```
case 'view':
    $blocks['subject'] = t('On the Bookshelf');
    $blocks['content'] = t('Temporary content');
    return $blocks;
```

The `view` operation should return one block of content for displaying to the end user. This block of content must have two parts stored as name/value pairs in an associative array: a `subject` and a `content` item.

The `subject` is the title of the block, and the `content` is main content of the block. The value of the `subject` entry will be used as a title for the block, while the `content` will be placed into the block's content.

Again, we used the translation function, `t()`, to translate the title and content of this block.

While it is not terribly exciting yet, our module is ready to test. The next thing to do is install it.

Installing a Module

We have a working module. Now we need to install it. This is typically done in three steps:

1. Copying the module to the correct location
2. Enabling the module
3. Configuring Drupal to display the module's content



Some of the contributed modules for Drupal require additional setup steps. Such steps are documented by the module's authors. In Chapter 4, we will create a module that requires a few additional steps before the module is useful.

We will walk through each of these three steps.

Step 1: Copying the Module

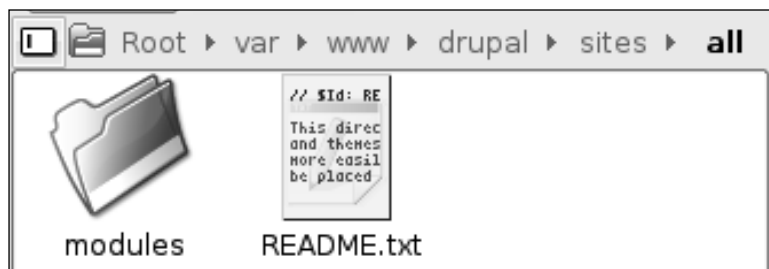
Modules in Drupal are stored in one of the three places under Drupal's root directory:

- `modules/`: This is the directory for core modules. Only modules supplied as part of the Drupal distribution should be stored here. None of our modules will ever be located here.
- `sites/all/modules/`: This is the directory for modules that should be available to all of the sites hosted on this Drupal installation. Usually, this is where you want to put your module.
- `sites/<site name>/modules`: Drupal can host multiple sites. Each site has a directory inside the `sites/` folder. For example, the default site is located in `sites/default/`. If you want to install site-specific modules on an instance of Drupal that runs multiple sites, the modules should go into the `sites/<site name>/modules/` directory, where `<site name>` should be replaced by the correct site name.

In this book, we will be storing our modules under the `sites/all/modules/` directory.

However, this directory is not created by default, so we will need to create it by hand.

On my Linux server, Drupal is installed in `/var/www/drupal/`. (Yours may be somewhere else.) All of the file system paths will be relative to this directory. We will add the appropriate subdirectory inside the `sites/all/` directory:



In this example, we change into the appropriate directory, create the new `modules/` directory.



By default, the permissions on the directory should be set to allow the web-server user (such as `www-data`) access to the files in the module. However, on some systems you may have to set these yourselves.

On Windows, the same can be done through Windows explorer, and the same goes for Mac and Finder. Simply locate your Drupal installation directory, navigate down to `sites\all`, and create a new folder named `modules`.

Next, we need to copy our module into this directory.



UNIX and Linux users: Don't move it; link it!

If you are actively developing a module, sometimes it is more convenient to create a symbolic link to the module directory instead of moving or copying the directory:

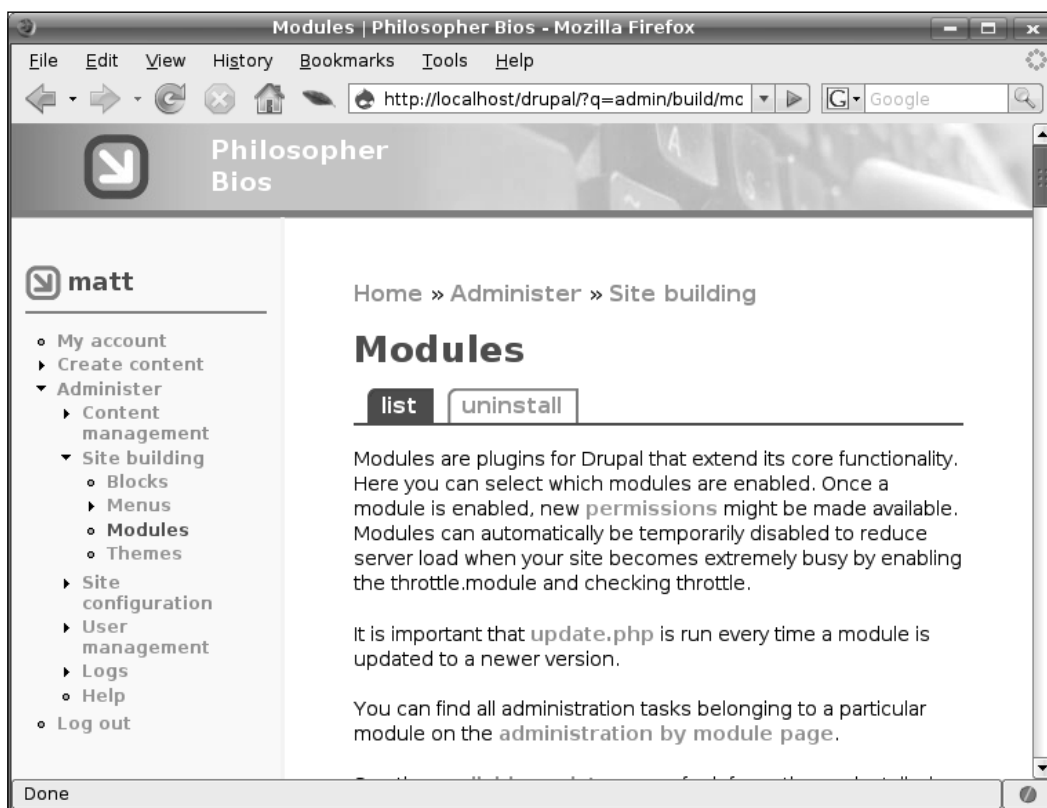
```
ln -s /home/m butcher/modules/goodreads
    /var/www/drupal/sites/all/modules/goodreads
```

Now we have our module in a location where Drupal expects to find modules.

Copying the module to the correct location is all we need to do for Drupal to recognize the module, but new modules are disabled by default. We will need to log in to the web interface and enable the module.

Step 2: Enabling the Module

A module is enabled through the Drupal Administration interface. Once logged into Drupal, navigate to **Administer | Site Building | Modules** in the left-hand navigation.



This page lists all the modules, beginning with the core modules (those installed by default). At the very bottom of this page is the list of third-party modules. Our module will appear in that list.

The screenshot shows a configuration form for the 'GoodReads Bookshelf' module. It has a table with columns: Enabled, Name, Version, and Description. The 'GoodReads Bookshelf' module is listed with a checked box in the 'Enabled' column. Below the table is a 'Save configuration' button.

Enabled	Name	Version	Description
<input checked="" type="checkbox"/>	GoodReads Bookshelf		Displays items from a GoodReads Bookshelf

Save configuration

To activate the module, simply check the box under the **Enabled** heading, and then click the **Save configuration** button.



Where did Drupal get the information about our module? For most of this part, this information came from our `goodreads.info` file.

Next, we need to configure the module to display on our site.

Step 3: Displaying the Module's Content

The module we have created is a block module. Typically, blocks are displayed in specifically defined locations on the screen. What we want to do now is tell Drupal where to display our block content.

Just as with enabling the module, this is done through the administration interface. Go to **Administer | Site Building | Blocks** to configure block placement.

The screenshot shows the Drupal administration interface. The top header says 'Philosopher Bios'. The left sidebar shows a user profile for 'matt' and a list of links including 'Code review', 'Empty cache', 'Enable Theme developer', 'Forums', 'Function reference', 'Hook_elements()', 'My account', 'PHPinfo()', 'Reinstall modules', 'Reset menus', 'Session viewer', 'Theme registry', 'Variable editor', 'Create content', and 'Administer'. The main content area shows the 'Blocks' configuration page. It has a breadcrumb trail: 'Home » Administer » Site building'. Below the breadcrumb, there are tabs for 'list' and 'add block'. The 'list' tab is active, showing a list of blocks: 'Pushbutton', 'Bluemarine', 'Chameleon', and 'Garland'. A message box says 'The block settings have been updated.' Below the message, there is a paragraph of text explaining the drag-and-drop interface for assigning blocks to regions.

This tool allows us to configure the details of how blocks appear on the site. In fact, the site uses the templates that a site visitor would see. You can see how the site looks as you configure it.

At the bottom of this page is the block configuration tool—lists of modules along with page placement parameters. We will configure our `goodreads` module to appear in the **right sidebar**.

If all goes well, then our `goodreads` module should display in the right sidebar. Make sure to press the **Save Blocks** button at the bottom. Otherwise the block's new location will not be saved.



To generate the preceding screen, the block placement screen calls the `hook_block()` functions for each of the block modules, setting `$op` to `list`.

When the block's new location is saved, it will be displayed in the right-hand column on all of our pages.

Block	Region	Operations
Left sidebar		
Navigation	Left sidebar ▼	configure
User login	Left sidebar ▼	configure
Right sidebar		
Goodreads Bookshelf*	Right sidebar ▼	configure
Content		
No blocks in this region		
Header		
No blocks in this region		
Footer		
Powered by Drupal	Footer ▼	configure

What is the content in this module? What we see in the above screenshot are the fields returned when Drupal's module manager calls the `hook_block()` function of our module with something equivalent to this:

```
goodreads_block('view');
```

This will return the `$blocks` array, whose contents look like this:

```
array(
  'subject' => 'On the Bookshelf',
  'content' => 'Temporary content'
)
```

The `subject` value is used as the block's title, and the `content` item is used as the block's content.

Our module is installed. But it is doing very little. Next, we will add some sophistication to our module.

Using Goodreads Data

So far, we have created a basic module that uses `hook_block()` to add block content and installed this basic module. As it stands, however, this module does no more than simply displaying a few lines of static text.

In this section, we are going to extend the module's functionality. We will add a few new functions that retrieve and format data from Goodreads.

Goodreads makes data available in an XML format based on RSS 2.0. The XML content is retrieved over **HTTP (HyperText Transport Protocol)**, the protocol that web browsers use to retrieve web pages. To enable this module to get Goodreads content, we will have to write some code to retrieve data over HTTP and then parse the retrieved XML.

Our first change will be to make a few modifications to `goodreads_block()`.

Modifying the Block Hook

We could cram all of our new code into the existing `goodreads_block()` hook; however, this would make the function cumbersome to read and difficult to maintain. Rather than adding significant code here, we will just call another function that will perform another part of the work.

```
/**
 * Implementation of hook_block
 */
function goodreads_block($op='list' , $delta=0, $edit=array()) {
  switch ($op) {
    case 'list':
      $blocks[0]['info'] = t('Goodreads Bookshelf');
      return $blocks;
    case 'view':
      $url = 'http://www.goodreads.com/review/list_rss/'
        . '398385'
        . '?shelf='
        . 'history-of-philosophy';
      $blocks['subject'] = t('On the Bookshelf');
      $blocks['content'] = _goodreads_fetch_bookshelf($url);
      return $blocks;
  }
}
```

The preceding code should look familiar. This is our hook implementation as seen earlier in the chapter. However, we have made a few modifications, indicated by the highlighted lines.

First, we have added a variable, `$url`, whose value is the URL of the Goodreads XML feed we will be using (`http://www.goodreads.com/review/list_rss/398385?shelf=history-of-philosophy`). In a completely finished module, we would want this to be a configurable parameter, but for now we will leave it hard-coded.

The second change has to do with where the module is getting its content. Previously, the function was setting the content to `t('Temporary content')`. Now it is calling another function: `_goodreads_fetch_bookshelf($url)`.

The leading underscore here indicates that this function is a private function of our module—it is a function not intended to be called by any piece of code outside of the module. Demarcating a function as private by using the initial underscore is another Drupal convention that you should employ in your own code.

Let's take a look at the `_goodreads_fetch_bookshelf()` function.

Retrieving XML Content over HTTP

The job of the `_goodreads_fetch_bookshelf()` function is to retrieve the XML content using an HTTP connection to the Goodreads site. Once it has done that, it will hand over the job of formatting to another function.

Here's a first look at the function in its entirety:

```
/**
 * Retrieve information from the Goodreads bookshelf XML API.
 *
 * This makes an HTTP connection to the given URL, and
 * retrieves XML data, which it then attempts to format
 * for display.
 *
 * @param $url
 *   URL to the goodreads bookshelf.
 * @param $num_items
 *   Number of items to include in results.
 * @return
 *   String containing the bookshelf.
 */
function _goodreads_fetch_bookshelf($url, $num_items=3) {
  $http_result = drupal_http_request($url);
```

```

if ($http_result->code == 200) {
    $doc = simplexml_load_string($http_result->data);
    if ($doc === false) {
        $msg = "Error parsing bookshelf XML for %url: %msg.";
        $vars = array('%url'=>$url, '%msg'=>$e->getMessage());
        watchdog('goodreads', $msg, $vars, WATCHDOG_WARNING);
        return t("Getting the bookshelf resulted in an error.");
    }
    return _goodreads_block_content($doc, $num_items);

    // Otherwise we don't have any data
}
else {
    $msg = 'No content from %url.';
    $vars = array('%url' => $url);
    watchdog('goodreads', $msg, $vars, WATCHDOG_WARNING);
    return t("The bookshelf is not accessible.");
}
}

```

Let's take a closer look.

Following the Drupal coding conventions, the first thing in the above code is an API description:

```

/**
 * Retrieve information from the Goodreads bookshelf XML API.
 *
 * This makes an HTTP connection to the given URL, and retrieves
 * XML data, which it then attempts to format for display.
 *
 * @param $url
 *   URL to the goodreads bookshelf.
 * @param $num_items
 *   Number of items to include in results.
 * @return
 *   String containing the bookshelf.
 */

```

This represents the typical function documentation block. It begins with a one-sentence overview of the function. This first sentence is usually followed by a few more sentences clarifying what the function does.

Near the end of the docblock, special keywords (preceded by the @ sign) are used to document the parameters and possible return values for this function.

- @param: The @param keyword is used to document a parameter and it follows the following format: @param <variable name> <description>. The description should indicate what data type is expected in this parameter.
- @return: This keyword documents what type of return value one can expect from this function. It follows the format: @return <description>.

This sort of documentation should be used for any module function that is not an implementation of a hook.

Now we will look at the method itself, starting with the first few lines.

```
function _goodreads_fetch_bookshelf($url, $num_items=3) {  
    $http_result = drupal_http_request($url);
```

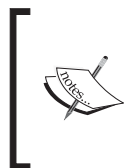
This function expects as many as two parameters. The required `$url` parameter should contain the URL of the remote site, and the optional `$num_items` parameter should indicate the maximum number of items to be returned from the feed.



While we don't make use of the `$num_items` parameter when we call `_goodreads_fetch_bookshelf()` this would also be a good thing to add to the module's configurable parameters.

The first thing the function does is use the Drupal built-in `drupal_http_request()` function found in the `includes/common.php` library. This function makes an HTTP connection to a remote site using the supplied URL and then performs an HTTP GET request.

The `drupal_http_request()` function returns an object that contains the response code (from the server or the socket library), the HTTP headers, and the data returned by the remote server.




Drupal is occasionally criticized for not using the object-oriented features of PHP. In fact, it does – but less overtly than many other projects. Constructors are rarely used, but objects are employed throughout the framework. Here, for example, an object is returned by a core Drupal function.

When the `drupal_http_request()` function has executed, the `$http_result` object will contain the returned information. The first thing we need to find out is whether the HTTP request was successful – whether it connected and retrieved the data we expect it to get.

We can get this information from the response code, which will be set to a negative number if there was a networking error, and set to one of the HTTP response codes if the connection was successful.

We know that if the server responds with the 200 (OK) code, it means that we have received some data.

 In a more robust application, we might also check for redirect messages (301, 302, 303, and 307) and other similar conditions. With a little more code, we could configure the module to follow redirects.

Our simple module will simply treat any other response code as indicating an error:

```
if ($http_result->code == 200) {
    // ...Process response code goes here...
    // Otherwise we don't have any data
} else {
    $msg = 'No content from %url.';
    $vars = array( '%url' => $url );
    watchdog('goodreads', $msg, $vars, WATCHDOG_WARNING);
    return t("The bookshelf is not accessible.");
}
```


First let's look at what happens if the response code is something other than 200:

```
} else {
    $msg = 'No content from %url.';
    $vars = array( '%url' => $url );
    watchdog('goodreads', $msg, $vars, WATCHDOG_WARNING);
    return t("The bookshelf is not accessible.");
}
```

We want to do two things when a request fails: we want to *log an error*, and then *notify the user* (in a friendly way) that we could not get the content. Let's take a glance at Drupal's logging mechanism.

The watchdog() Function

Another important core Drupal function is the `watchdog()` function. It provides a logging mechanism for Drupal.

 **Customize your logging**
Drupal provides a hook (`hook_watchdog()`) that can be implemented to customize what logging actions are taken when a message is logged using `watchdog()`. By default, Drupal logs to a designated database table. You can view this log in the administration section by going to **Administer | Logs**.

The `watchdog()` function gathers all the necessary logging information and fires off the appropriate logging event.

The first parameter of the `watchdog()` function is the logging category. Typically, modules should use the module name (`goodreads` in this case) as the logging category. In this way, finding module-specific errors will be easier.

The second and third `watchdog` parameters are the text of the message (`$msg` above) and an associative array of data (`$vars`) that should be substituted into the `$msg`. These substitutions are done following the same translation rules used by the `t()` function. Just like with the `t()` function's substitution array, placeholders should begin with `!`, `@`, or `%`, depending on the level of escaping you need.

So in the preceding example, the contents of the `$url` variable will be substituted into `$msg` in place of the `%url` marker.

Finally, the last parameter in the `watchdog()` function is a constant that indicates the log message's priority, that is, how important it is.

There are eight different constants that can be passed to this function:

- `WATCHDOG_EMERG`: The system is now in an unusable state.
- `WATCHDOG_ALERT`: Something must be done immediately.
- `WATCHDOG_CRITICAL`: The application is in a critical state.
- `WATCHDOG_ERROR`: An error occurred.
- `WATCHDOG_WARNING`: Something unexpected (and negative) happened, but didn't cause any serious problems.
- `WATCHDOG_NOTICE`: Something significant (but not bad) happened.
- `WATCHDOG_INFO`: Information can be logged.
- `WATCHDOG_DEBUG`: Debugging information can be logged.

Depending on the logging configuration, not all these messages will show up in the log.

The `WATCHDOG_ERROR` and `WATCHDOG_WARNING` levels are usually the most useful for module developers to record errors. Most modules do not contain code significant enough to cause general problems with Drupal, and the upper three log levels (alert, critical, and emergency) should probably not be used unless Drupal itself is in a bad state.



There is an optional fifth parameter to `watchdog()`, usually called `$link`, which allows you to pass in an associated URL. Logging back ends may use that to generate links embedded within logging messages.

The last thing we want to do in the case of an error is return an error message that can be displayed on the site. This is simply done by returning a (possibly translated) string:

```
return t("The bookshelf is not accessible.");
```

We've handled the case where retrieving the data failed. Now let's turn our attention to the case where the HTTP request was successful.

Processing the HTTP Results

When the result code of our request is 200, we know the web transaction was successful. The content may or may not be what we expect, but we have good reason to believe that no error occurred while retrieving the XML document.

So, in this case, we continue processing the information:

```
if ($http_result->code == 200) {
    // ... Processing response here...
    $doc = simplexml_load_string($http_result->data);
    if ($doc === false) {
        $msg = "Error parsing bookshelf XML for %url: %msg.";
        $vars = array('%url'=>$url, '%msg'=>$e->getMessage());
        watchdog('goodreads', $msg, $vars, WATCHDOG_WARNING);
        return t("Getting the bookshelf resulted in an error.");
    }
    return _goodreads_block_content($doc, $num_items);
    // Otherwise we don't have any data
} else { // ... Error handling that we just looked at.
```

In the above example, we use the PHP 5 **SimpleXML** library. SimpleXML provides a set of convenient and easy-to-use tools for handling XML content. This library is not present in the now-deprecated PHP 4 language version.

For compatibility with outdated versions of PHP, Drupal code often uses the **Expat** parser, a venerable old event-based XML parser supported since PHP 4 was introduced. Drupal even includes a wrapper function for creating an Expat parser instance. However, writing the event handlers is time consuming and repetitive. SimpleXML gives us an easier interface and requires much less coding.

For an example of using the Expat event-based method for handling XML documents, see the built-in **Aggregator module**. For detailed documentation on using Expat, see the official PHP documentation: <http://php.net/manual/en/ref.xml.php>.

We will parse the XML using `simplexml_load_string()`. If parsing is successful, the function returns a SimpleXML object. However, if parsing fails, it will return `false`.

In our code, we check for a `false`. If one is found, we log an error and return a friendly error message. But if the Goodreads XML document was parsed properly, this function will call another function in our module, `_goodreads_block_content()`. This function will build some content from the XML data.

Formatting the Block's Contents

Now we are going to look at one more function—a function that extracts data from the SimpleXML object we have created and formats it for display.

The function we will look at here is basic and doesn't take advantage of the Drupal theming engine. Usually, formatting data for display is handled using the theming engine. Themes are the topic of our next chapter.

Here is our `_goodreads_block_content()` function:

```
/**
 * Generate the contents of a block from a SimpleXML object.
 * Given a SimpleXML object and the maximum number of
 * entries to be displayed, generate some content.
 *
 * @param $doc
 *   SimpleXML object containing Goodreads XML.
 * @param $num_items
 *   Number of items to format for display.
 * @return
 *   Formatted string.
 */
function _goodreads_block_content($doc, $num_items=3) {
  $items = $doc->channel->item;
  $count_items = count($items);
  $len = ($count_items < $num_items) ? $count_items : $num_items;

  $template = '<div class="goodreads-item">'
    . '<br/>%s<br/>by %s</div>';
  // Default image: 'no cover'
  $default_img = 'http://www.goodreads.com/images/nocover-60x80.jpg';
  $default_link = 'http://www.goodreads.com';

  $out = '';
  foreach ($items as $item) {
```

```

    $author = check_plain($item->author_name);
    $title = strip_tags($item->title);
    $link = check_url(trim($item->link));
    $img = check_url(trim($item->book_image_url));
    if (empty($author)) $author = '';
    if (empty($title)) $title = '';
    if (empty($link) || $link == 0) $link = $default_link;
    if (empty($img)) $img = $default_img;
    $book_link = l($title, $link);
    $out .= sprintf($template, $img, $book_link, $author);
  }
  $out .= '<br/><div class="goodreads-more">'
    . l('Goodreads.com', 'http://www.goodreads.com')
    . '</div>';
  return $out;
}

```

As with the last function, this one does not implement a Drupal hook. In fact, as the leading underscore (`_`) character should indicate, this is a private function, intended to be called only by other functions within this module.

Again the function begins with a documentation block explaining its purpose, parameters, and return value. From there, we begin the function:

```

function _goodreads_block_content($doc, $num_items=3) {
  $items = $doc->channel->item;

```

The first thing the function does is get a list of `<item/>` elements from the XML data. To understand what is going on here, let's look at the XML (abbreviated for our example) returned from Goodreads:

```

<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Matthew's bookshelf: history-of-philosophy</title>
    <copyright>
      <![CDATA[
        Copyright (C) 2006 Goodreads Inc. All rights reserved.]]>
    </copyright>
    <link>http://www.goodreads.com/review/list_rss/398385</link>
    <item>
      <title>
        <![CDATA[Thought's Ego in Augustine and Descartes]]>
      </title>
      <link>http://www.goodreads.com/review/show/6895959?
        utm_source=rss&utm_medium=api</link>
      <book_image_url>
        <![CDATA[

```

```
        http://www.goodreads.com/images/books/96/285/964285-s-
                                1179856470.jpg
    ]]>
    </book_image_url>
    <author_name><![CDATA[Gareth B. Matthews]]></author_name>
</item>
<item>
    <title>
    <![CDATA[Augustine: On the Trinity Books 8-15 (Cambridge Texts
                                in the History of Philosophy)]]>
    </title>
    <link>http://www.goodreads.com/review/show/6895931?
                                utm_source=rss&utm_medium=api</link>
    <book_image_url>
    <![CDATA[
        http://www.goodreads.com/images/books/35/855/352855-s-
                                1174007852.jpg
    ]]>
    </book_image_url>
    <author_name><![CDATA[Gareth B. Matthews]]></author_name>
</item>
<item>
    <title>
    <![CDATA[A Treatise Concerning the Principles of Human
                                Knowledge (Oxford Philosophical Texts)]]>
    </title>
    <link>http://www.goodreads.com/review/show/6894329?
                                utm_source=rss&utm_medium=api</link>
    <book_image_url>
    <![CDATA[
        http://www.goodreads.com/images/books/10/138/1029138-s-
                                1180349380.jpg
    ]]>
    </book_image_url>
    <author_name><![CDATA[George Berkeley]]></author_name>
</item>
</channel>
</rss>
```

The above XML follows the familiar structure of an RSS document. The `<channel/>` contains, first, a list of fields that describes the bookshelf we have retrieved, and then a handful of `<item/>` elements, each of which describes a book from the bookshelf.

We are interested in the contents of `<item/>` elements, so we start off by grabbing the list of items:

```
$items = $doc->channel->item;
```

The SimpleXML `$doc` object contains attributes that point to each of its child elements. The `<rss/>` element (which is represented as `$doc`) has only one child: `<channel/>`. In turn, `<channel/>` has several child elements: `<title/>`, `<copyright/>`, `<link/>`, and several `<item/>` elements. These are represented as `$doc->title`, `$doc->copyright`, and so on.

What happens when there are several elements with the same name like `<item/>`?

They are stored as an array. So in our code above, the variable `$items` will point to an array of `<item/>` elements.

Next, we determine how many items will be displayed, specify a basic template we will later use to create the HTML for our block, and set a few default values:

```
$count_items = count($items);
$len = ($count_items < $num_items) ? $count_items : $num_items;

$template = '<div class="goodreads-item">'
           . '<br/>%s<br/>by %s</div>';
// Default image: 'no cover'
$default_img = 'http://www.goodreads.com/images/nocover-60x80.jpg';
$default_link = 'http://www.goodreads.com';
```

In the first line, we make sure that we don't use any more than `$num_items`. Next, we assign the `$template` variable an `sprintf()` style template. We will use this to format our entries in just a moment.

Finally, we set default values for a logo image (`$default_img`) and a link back to Goodreads (`$default_link`).

Once this is done, we are ready to loop through the array of `$items` and generate some HTML:

```
$out = '';
foreach ($items as $item) {
    $author = check_plain($item->author_name);
    $title = strip_tags($item->title);
    $link = check_url(trim($item->link));
    $img = check_url(trim($item->book_image_url));
    if (empty($author)) $author = 'Unknown';
    if (empty($title)) $title = 'Untitled';
    if (empty($link)) $link = $default_link;
    if (empty($img)) $img = $default_img;
    $book_link = l($title, $link);
    $out .= sprintf($template, $img, $book_link, $author);
}
```

Using a `foreach` loop, we go through each `$item` in the `$items` list. Each of these items should look something like the following:

```
<item>
  <title>
    <![CDATA[Book Title]]>
  </title>
  <link>http://www.goodreads.com/something/</link>
  <book_image_url>
    <![CDATA[
      http://www.goodreads.com/images/something.jpg
    ]]>
  </book_image_url>
  <author_name><![CDATA[Author Name]]></author_name>
</item>
```

We want to extract the title, link, author name, and an image of the book. We get these from the `$item` object:

```
$author = check_plain($item->author_name);
$title = strip_tags($item->title);
$link = check_url(trim($item->link));
$img = check_url(trim($item->book_image_url));
```

While we trust Goodreads, we do want to sanitize the data it sends us as an added layer of security. Above, we check the values of `$author` and `$title` with the functions `check_plain()` and `strip_tags()`.

The `strip_tags()` function is built into PHP. It simply reads a string and strips out anything that looks like an HTML or XML tag. This provides a basic layer of security, since it would remove the tags that might inject a script, applet, or ActiveX object into our page. But this check does still allow HTML entities like `&` or `»te;`.

Drupal contains several string encoding functions that provide different services than `strip_tags()`. Above, we use `check_plain()` to perform some escaping on `$item->author_name`. Unlike `strip_tags()`, `check_plain()` does not remove anything. Instead, it encodes HTML tags into entities (like the `@` modifier in `t()` function substitutions). So `check_plain('Example')` would return the string `Example`.



The `check_plain()` function plays a very important role in Drupal security. It provides one way of avoiding cross-site scripting attacks (XSS), as well as insertion of malicious HTML.

There is a disadvantage to using `check_plain()`, though. If `check_plain()` encounters an HTML entity, like `<`, it will encode it again. Thus, `<` would become `&lt;`. The initial ampersand (`&`) is encoded into `&`.

With the `$item->link` and `$item->book_image_url` objects, though, we have to do two things. First, we must `trim()` the results to remove leading or trailing white spaces. This is important because Drupal's `l()` function, which we will see in just a moment, will not process URLs correctly if they start with white spaces.

We also use Drupal's `check_url()` function to verify that the URL is legitimate. `check_url()` does a series of checks intended to catch malicious URLs. For example, it will prevent the `javascript:` protocol from being used in a URL. This we do as a safety precaution.

Next, we check each of the newly assigned variables. We want to make sure that if a variable is `null` or empty, it gets a default value.

```
if (empty($author)) $author = 'Unknown';
if (empty($title)) $title = 'Untitled';
if (empty($link)) $link = $default_link;
if (empty($img)) $img = $default_img;
```

The last thing we do in this `foreach` loop is format the entry as HTML for display:

```
$book_link = l($title, $link);
$out .= sprintf( $template, $img, $book_link, $author);
```

First, we create a link to the book review page at Goodreads. This is done with Drupal's `l()` function (that's a single lowercase L). `l()` is another important Drupal function. This function creates a hyperlink. In the above code, it takes the book title (`$title`), and a URL (`$link`), and creates an HTML tag that looks like this:

```
<a
href="http://www.goodreads.com/review/show/6894329?utm_
source=rss&utm_medium=api">
  A Treatise Concerning the Principles of Human Knowledge (Oxford
  Philosophical Texts)
</a>
```

That string is stored in `$book_link`. We then do the HTML formatting using a call to the PHP `sprintf()` function:

```
$out .= sprintf( $template, $img, $book_link, $author);
```

The `sprintf()` function takes a template (`$template`) as its first argument. We defined `$template` outside of the `foreach` loop. It is a string that looks as follows:

```
<div class="goodreads-item"><br/>%s<br/>by %s</div>
```

`sprintf()` will read through this string. Each time it encounters a placeholder, like `%s`, it will substitute in the value of an argument.

There are three string placeholders (`%s`) in the string. `sprintf()` will sequentially replace them with the string values of the three other parameters passed into `sprintf()`: `$img`, `$book_link`, and `$author`.

So `sprintf()` would return a string that looked something like the following:

```
<div class="goodreads-item">

<br/><a href="http://www.goodreads.com/somepath">
Thought&#039;s Ego in Augustine and Descartes
</a><br/>by Gareth B. Matthews</div>
```

That string is then added to `$output`. By the time the `foreach` loop completes, `$output` should contain a fragment of HTML for each of the entries in `$items`.

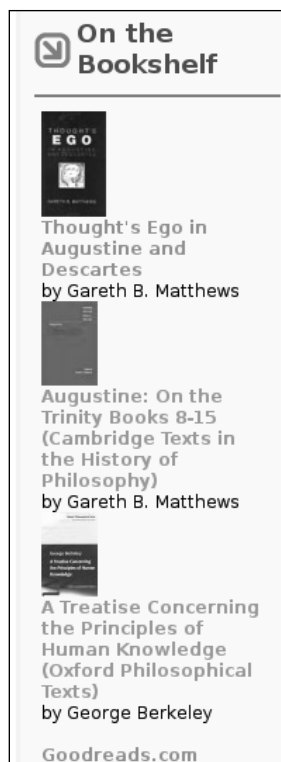


The PHP `sprintf()` and `printf()` functions are very powerful, and can make PHP code easier to write, maintain, and read. View the PHP documentation for more information: <http://php.net/manual/en/function.sprintf.php>.

Once we are done with the `foreach` loop, we only have a little left to do. We need to add a link back to Goodreads to our `$out` HTML, and then we can return the output:

```
$out .= '<br/><div class="goodreads-more">'
      . l('Goodreads.com', 'http://www.goodreads.com')
      . '</div>';
return $out;
}
```

The block hook (`goodreads_block()`) will take the formatted HTML returned by `_goodreads_block_content()` and store it in the `contents` of the block. Drupal will display the results in the right-hand column, as we configured in the previous section:



The first three items in our Goodreads history of philosophy bookshelf are now displayed as blocks in Drupal.

There is much that could be done to improve this module. We could add caching support, so that each request did not result in a new retrieval of the Goodreads XML. We could create additional security measures to check the XML content. We could add an administration interface that would allow us to set the bookshelf URL instead of hard coding the value in. We could also use the theming system to create the HTML and style it instead of hard coding HTML tags into our code.

In fact, in the next chapter, we will take a closer look at the theming system and see how this particular improvement could be made.

However, to complete our module, we need a finishing touch. We need to add some help text.

Finishing Touches: hook_help()

We now have a functioning module. However, there is one last thing that a good Drupal module should have. Modules should implement the `hook_help()` function to provide help text for module users.

Our module is not very complex. Our help hook won't be, either:

```
/**
 * Implementation of hook_help()
 */
function goodreads_help($path, $arg) {
  if ($path == 'admin/help#goodreads') {
    $txt = 'The Goodreads module uses the !goodreads_url XML '
      . 'API to retrieve a list of books and display it as block '
      . 'content.';
    $link = l('Goodreads.com', 'http://www.goodreads.com');
    $replace = array(
      '!goodreads_url' => $link
    );
    return '<p>'. t($txt, $replace) . '</p>';
  }
}
```

The `hook_help()` function gets two parameters: `$path`, which contains a URI fragment indicating what help page was called, and `$arg`, which might contain extra information.

In a complex instance of `hook_help()`, you might use a switch statement on the `$path`, returning different help text for each possible path. For our module, only one path is likely to be passed in: the path to general help in the administration section. This path follows the convention `admin/help#<module name>`, where `<module name>` should be replaced with the name of the module (e.g. `goodreads`).

The function is straightforward: The help text is stored in the `$txt` variable. If our module required any additional setup or was linked to additional help, we would want to indicate this here. But in our case, all we really need is a basic description of what the module does.

We also want to insert a link back to Goodreads.com. To do this, we create the placeholder (`!goodreads_url`) in the `$txt` content, and then create the link (`$link`) by calling the `l()` function.

Since we are going to pass the `$txt` string and the link into the translation function, and let that function replace the placeholder with the link, we need to put `$link` inside an array – the `$replace` array.

Finally the help text is returned after being passed through the `t()` translation function.

This help text will now be accessible on the **Administer | Help** page:

Home » Administer » Help

Goodreads Bookshelf

The Goodreads module uses the **Goodreads.com** XML API to retrieve a list of books and display it as block content.

That is all there is to creating help text.

It is good coding practice to include help text with your module. And the more complex a module is, the better the help text should be. In future chapters, we will start every module by implementing `hook_help()`. Often, though, we will keep text in the help function brief for the sake of keeping the book concise and readable.

In your own modules you may want to create more verbose help text. This is helpful to users who don't have access to the code (or to a book explaining how the module works in detail).

Summary

In this chapter we created our first module. We created our first `.info` and `.module` files. We implemented our first two hooks, `hook_block()` and `hook_help()`. We installed our module, and then went on to extend the module to access an outside XML source for content.

We also looked at several important Drupal functions, with `t()`, `l()`, `watchdog()`, and `check_plain()` being the most important functions of the bunch.

In the coming chapters, we will build on the concepts covered in this chapter. Next, we will turn to the theming system to learn how to cleanly separate layout and styling information from the rest of the code.

Where to buy this book

You can buy Learning Drupal 6 Module Development from the Packt Publishing website:
<http://www.packtpub.com/drupal-6-module-development/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/drupal-6-module-development/book