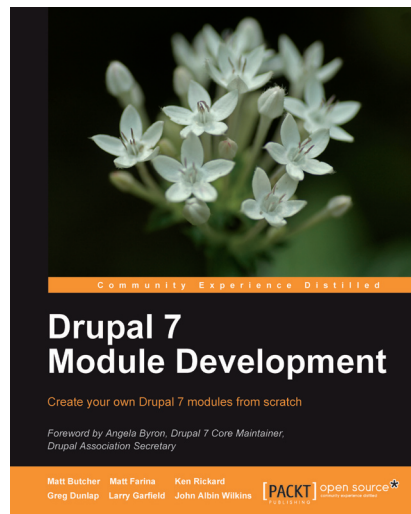


Drupal 7 Module Development

Matt Butcher
Greg Dunlap
Matt Farina
Larry Garfield
Ken Rickard
John Albin Wilkins



Chapter No. 2 "Creating Your First Module"

In this package, you will find:

A Biography of the authors of the book

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

A synopsis of the book's content

Information on where to buy this book

About the Authors

Matt Butcher is a web developer and author. He has written five other books for Packt, including *Drupal 6 JavaScript and jQuery* and *Learning Drupal 6 Module Development*. Matt is a Senior Developer at ConsumerSearch.com (a New York Times/About.Com company), where he works on one of the largest Drupal sites in the world. Matt is active in the Drupal community, managing several modules. He also leads a couple of Open Source projects including QueryPath.

I would like to thank Larry, Ken, Sam, Matt, Greg, and John for working with me on the book. They are a fantastic group of people to work with. I'd also like to thank the technical reviewers of this book, all of whom contributed to making this a better work.

I'd also like to thank Austin Smith, Brian Tully, Chachi Kruel, Marc McDougall, Theresa Summa, and the rest of the ConsumerSearch.com team for their support. The folks at Palantir.net were instrumental in getting this book off the ground, and I am always grateful for their support. Finally, Angie, Anna, Claire, and Katherine have sacrificed some weekends and evenings with me for the benefit of this book. To them, I owe the biggest debt of gratitude.

For More Information:

www.packtpub.com/drupal-7-module-development/book

Greg Dunlap is a software engineer based in Stockholm, Sweden. Over the past 15 years, Greg has been involved in a wide variety of projects, including desktop database applications, kiosks, embedded software for pinball and slot machines, and websites in over a dozen programming languages. Greg has been heavily involved with Drupal for three years, and is the maintainer of the Deploy and Services modules as well as a frequent speaker at Drupal conferences. Greg is currently a Principal Software Developer at NodeOne.

Several people played crucial roles in my development as a Drupal contributor, providing support and encouragement just when I needed it most. My deepest gratitude to Gary Love, Jeff Eaton, Boris Mann, Angie Byron, and Ken Rickard for helping me kick it up a notch. Extra special thanks to the lovely Roya Naini for putting up with lost nights and weekends in the service of finishing my chapters.

Matt Farina has been a Drupal developer since 2005. He is a senior front-end developer, engineer, and technical lead for Palantir.net, where he works on a wide variety of projects ranging from museums to large interactive sites. He is a contributor to Drupal core as well as a maintainer of multiple contributed Drupal modules.

Matt wrote his first computer program when he was in the 5th grade. Since then he has programmed in over a dozen languages. He holds a BS in Electrical Engineering from Michigan State University.

For More Information:

www.packtpub.com/drupal-7-module-development/book

Larry Garfield is a Senior Architect and Engineer at Palantir.net, a leading Drupal development firm based in Chicago. He has been building websites since he was 16, which is longer than he'd like to admit, and has been working in PHP since 1999. He found Drupal in 2005, when Drupal 4.6 was still new and cool, and never really left. He is the principle architect and maintainer of the Drupal database subsystem among various other core initiatives and contributed modules.

Previously, Larry was a Palm OS developer and a journalist covering the mobile electronics sector and was the technical editor for *Building Powerful and Robust Websites with Drupal 6*, also from Packt. He holds a Bachelors and Masters Degree in Computer Science from DePaul University.

If I were to thank all of the people who made this book possible it would take several pages, as the Drupal 7 contributor list was well over 700 people, the last time I checked. Instead I will simply say thank you to the entire community for being so vibrant, supportive, and all-around amazing that it still brings a tear to my eye at times even after half a decade.

Extra special thanks go to Dries Buytaert, not just for being our project lead, but for sitting down on the floor next to me at DrupalCon Sunnyvale and encouraging me to run with this crazy idea I had, about using this "PDO" thing for Drupal's database layer. I doubt he realized how much trouble I'd cause him over the next several years.

Of course to my parents, who instilled in me not only a love of learning but a level of pedantry and stubbornness without which I would never have been able to get this far in Drupal, to say nothing of this book.

For More Information:

www.packtpub.com/drupal-7-module-development/book

Ken Rickard is a senior programmer at Palantir.net, a Chicago-based firm specializing in developing Drupal websites. He is a frequent contributor to the Drupal project, and is the maintainer of the Domain Access, MySite, and Menu Node API modules. At Palantir, he architects and builds large-scale websites for a diverse range of customers, including Foreign Affairs magazine, NASCAR, and the University of Chicago.

From 1998 through 2008, Ken worked in the newspaper industry, beginning his career managing websites and later becoming a researcher and consultant for Morris DigitalWorks. At Morris, Ken helped launch BlufftonToday.com, the first newspaper website launched on the Drupal platform. He later led the Drupal development team for SavannahNOW.com. He co-founded the Newspapers on Drupal group (<http://groups.drupal.org/newspapers-on-drupal>) and is a frequent advisor to the newspaper and publishing industries.

In 2008, Ken helped start the Knight Drupal Initiative, an open grant process for Drupal development, funded by the John L. and James S. Knight Foundation. He is also a member of the advisory board of PBS Engage, a Knight Foundation project to bring social media to the Public Broadcasting Service.

Prior to this book, Ken was a technical reviewer for *Packt Publishing's Drupal 6 Site Blueprints* by Timi Ogunjobi.

I must thank the entire staff at Palantir, the Drupal community, and, most of all, my lovely and patient wife Amy, without whom none of this would be possible.

For More Information:

www.packtpub.com/drupal-7-module-development/book

John Albin Wilkins has been a web developer for a long time. In April 1993, he was one of the lucky few to use the very first graphical web browser, Mosaic 1.0, and he's been doing web development professionally since 1994. In 2005, John finally learned how idiotic it was to build your own web application framework, and discovered the power of Drupal; he never looked back.

In the Drupal community, he is best known as JohnAlbin, one of the top 20 contributors to Drupal 7 and the maintainer of the Zen theme, which is a highly-documented, feature-rich "starter" theme with a powerfully flexible CSS framework. He has also written several front-end-oriented utility modules, such as the Menu Block module. John currently works with a bunch of really cool Drupal developers, designers, and themers at Palantir.net.

His occasional musings, videos, and podcasts can be found at <http://john.albin.net>.

I'd to thank the entire Drupal community for its wonderful support, friendship, aggravation, snark, and inspiration; just like a family. I'd also like to thank my real family, my wife and two kids, Jenny, Owen and Ella, for making me want to be a better person. I love you all.

For More Information:

www.packtpub.com/drupal-7-module-development/book

Drupal 7 Module Development

Drupal is an award-winning open-source Content Management System. It's a modular system, with an elegant hook-based architecture, and great code. Modules are plugins for Drupal that extend, build or enhance Drupal core functionality. In *Drupal 7 Module Development* book, six professional Drupal developers use a practical, example-based approach to introduce PHP developers to the powerful new Drupal 7 tools, APIs, and strategies for writing custom Drupal code. These tools not only make management and maintenance of websites much easier, but they are also great fun to play around with and amazingly easy to use.

What This Book Covers

Chapter 1, Introduction to Drupal Module Development gives an introduction to the scope of Drupal as a web-based Content Management System. It dwells on basic aspects such as the technologies that drive Drupal and the architectural layout of Drupal. A brief idea of the components (subsystems) of Drupal and the tools that may be used to develop it, completes the basic picture of Drupal.

Chapter 2, A First Module, gets things into action, by describing how to start building our first module in Drupal. That done, it will tell us how Block API can be used to create our custom code for Drupal. Finally, there is a word or two on how to test our code by writing Automated tests.

Chapter 3, Drupal Themes, is all about the Theme Layer in Drupal. It starts with ways to theme, and then proceeds to aspects associated with Theming. It talks about 'Render Elements' and concludes by getting us familiar with 'Theme Registry'.

Chapter 4, Theming a Module uses the concepts we saw in the previous chapter to theme modules in Drupal. It acquaints us with the concept of re-using a default theme implementation, and teaches us to build a theme implementation for real-life situations.

Chapter 5, Building an Admin Interface will show us how to go about building a module, complete with an administrative interface. While doing this, basic concepts of modules discussed in Chapter 2 will be useful. A 'User Warn' module is developed as an illustration, in the chapter.

Chapter 6, Working with Content lays emphasis on managing content. Creation of entity, controller class, integrating our entity with the Field API, and displaying confirmation forms are some of the things that we come across in this chapter.

Chapter 7, *Creating New Fields*, will take a look into creating new Fields. Further, it teaches us how to use corresponding Widgets to allow users to edit the Fields. Finally, to ensure that data is displayed as desired, the role of Formatters is discussed in the chapter.

For More Information:

www.packtpub.com/drupal-7-module-development/book

Chapter 8, Module Permissions and Security is all about access control and security. It talks about Permissions, which help users to gain access (or be denied access) to specific features. Also, the chapter talks about how to manage roles programmatically. One of the most crucial areas of website security, Form handling, is detailed here.

Chapter 9, Node Access deals with node access, which is one of the most powerful tools in the Drupal API. It sheds light on how access to a node is determined and on major operations controlled by the Node Access API, among other things.

Chapter 10, JavaScript in Drupal provides the fundamental knowledge required to work with JavaScript within Drupal. This helps to create powerful features such as the overlay, auto complete, drag and drop, and so on.

Chapter 11, Working with Files and Images talks about how management and maintenance can be made much easier by using File and Image APIs in Drupal 7. Also, the chapter tells us about various image processing techniques involved in working with images, making things more colorful and fun.

Chapter 12, Installation Profiles outlines the process of working with 'Distributions' and 'Installation Profiles' in Drupal. They help to make the developer's job easier.

Appendix A, Database Access, offers helpful insights regarding the developer's ability to take advantage of the Database Layer of Drupal 7, in order to make powerful cross-database queries.

Appendix B, Security, emphasizes the need to develop a practice to bear the security aspect in mind while writing the code. It deals with two ways of dealing with potentially insecure data, namely, 'filtering' and 'escaping'.

For More Information:

www.packtpub.com/drupal-7-module-development/book

2

Creating Your First Module

The focus of this chapter is module creation. In the last chapter we surveyed Drupal's architecture advanced. We learned about the basic features and subsystems. We also saw some tools available for development. Now we are going to begin coding.

Here are some of the important topics that we will cover in this chapter:

- Starting a new module
- Creating `.info` files to provide Drupal with module information
- Creating `.module` files to store Drupal code
- Adding new blocks using the **Block Subsystem**
- Using common Drupal functions
- Formatting code according to the Drupal coding standards
- Writing an automated test for Drupal

By the end of this chapter, you should have the foundational knowledge necessary for building your own module from scratch.

Our goal: a module with a block

In this chapter we are going to build a simple module. The module will use the **Block Subsystem** to add a new custom block. The block that we add will simply display a list of all of the currently enabled modules on our Drupal installation.



The block subsystem was introduced in the previous chapter alongside other important Drupal subsystems.

For More Information:

www.packtpub.com/drupal-7-module-development/book

We are going to divide this task of building a new module into the three parts:

- Create a new module folder and module files
- Work with the Block Subsystem
- Write automated tests using the SimpleTest framework included in Drupal

We are going to proceed in that order for the sake of simplicity. One might object that, following agile development processes, we ought to begin by writing our tests. This approach is called **Test-driven Development (TDD)**, and is a justly popular methodology.



Agile software development is a particular methodology designed to help teams of developers effectively and efficiently build software. While Drupal itself has not been developed using an agile process, it does facilitate many of the agile practices. To learn more about agile, visit <http://agilemanifesto.org/>.

However, our goal here is not to exemplify a particular methodology, but to discover how to write modules. It is easier to learn module development by first writing the module, and then learn how to write unit tests. It is easier for two reasons:

- SimpleTest (in spite of its name) is the least simple part of this chapter. It will have double the code-weight of our actual module.
- We will need to become acquainted with the APIs we are going to use in development before we attempt to write tests that assume knowledge of those APIs.

In regular module development, though, you may certainly choose to follow the TDD approach of writing tests first, and then writing the module.

Let's now move on to the first step of creating a new module.

Creating a new module

Creating Drupal modules is easy. How easy? Easy enough that over 5,000 modules have been developed, and many Drupal developers are even PHP novices! In fact, the code in this chapter is an illustration of how easy module coding can be. We are going to create our first module with only one directory and two small files.

Module names

It goes without saying that building a new module requires naming the module. However, there is one minor ambiguity that ought to be cleared up at the outset, a Drupal module has two names:

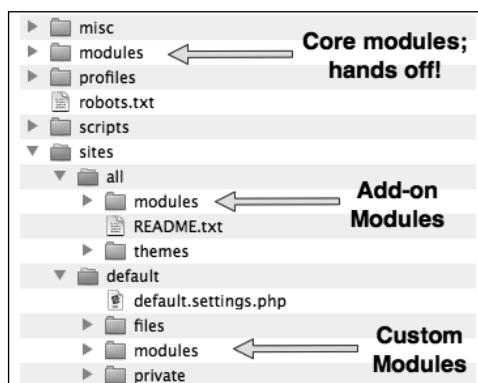
- **A human-readable name:** This name is designed to be read by humans, and should be one or a couple of words long. The words should be capitalized and separated by spaces. For example, one of the most popular Drupal modules has the human-readable name **Views**. A less-popular (but perhaps more creatively named) Drupal 6 module has the human-readable name **Eldorado Superfly**.
- **A machine-readable name:** This name is used internally by Drupal. It can be composed of lower-case and upper-case letters, digits, and the underscore character (using upper-case letters in machine names is frowned upon, though). No other characters are allowed. The machine names of the above two modules are `views` and `eldorado_superfly`, respectively.

By convention, the two names ought to be as similar as possible. Spaces should be replaced by underscores. Upper-case letters should generally be changed to lower-case.

Because of the convention of similar naming, the two names can usually be used interchangeably, and most of the time it is not necessary to specifically declare which of the two names we are referring to. In cases where the difference needs to be made (as in the next section), the authors will be careful to make it.

Where does our module go?

One of the less intuitive aspects of Drupal development is the filesystem layout. Where do we put a new module? The obvious answer would be to put it in the `/modules` directory alongside all of the core modules.



As obvious as this may seem, the `/modules` folder is not the right place for your modules. In fact, you should never change anything in that directory. It is reserved for core Drupal modules only, and will be overwritten during upgrades.

The second, far less obvious place to put modules is in `/sites/all/modules`. This is the location where all unmodified add-on modules ought to go, and tools like **Drush** (a Drupal command line tool) will download modules to this directory.

In some sense, it is okay to put modules here. They will not be automatically overwritten during core upgrades.

However, as of this writing, `/sites/all/modules` is not the recommended place to put custom modules unless you are running a multi-site configuration and the custom module needs to be accessible on all sites.

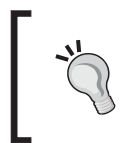
The current recommendation is to put custom modules in the `/sites/default/modules` directory, which does not exist by default. This has a few advantages. One is that standard add-on modules are stored elsewhere, and this separation makes it easier for us to find our own code without sorting through clutter. There are other benefits (such as the loading order of module directories), but none will have a direct impact on us.



Throughout this book, we will always be putting our custom modules in `/sites/default/modules`. This follows Drupal best practices, and also makes it easy to find our modules as opposed to all of the other add-on modules.

The one disadvantage of storing all custom modules in `/sites/default/modules` appears only under a specific set of circumstances. If you have Drupal configured to serve multiple sites off of one single instance, then the `/sites/default` folder is only used for the default site. What this means, in practice, is that modules stored there will not be loaded at all for other sites.

In such cases, it is generally advised to move your custom modules into `/sites/all/modules/custom`.



Other module directories

Drupal does look in a few other places for modules. However, those places are reserved for special purposes.

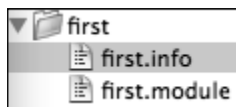
Creating the module directory

Now that we know that our modules should go in `/sites/default/modules`, we can create a new module there.

Modules can be organized in a variety of ways, but the best practice is to create a module directory in `/sites/default/modules`, and then place at least two files inside the directory: a `.info` (pronounced "dot-info") file and a `.module` ("dot-module") file.

The directory should be named with the machine-readable name of the module. Similarly, both the `.info` and `.module` files should use the machine-readable name.

We are going to name our first module with the machine-readable name `first`, since it is our first module. Thus, we will create a new directory, `/sites/default/modules/first`, and then create a `first.info` file and a `first.module` file:



Those are the only files we will need for our module.

For permissions, make sure that your webserver can read both the `.info` and `.module` files. It should not be able to write to either file, though.



In some sense, the only file absolutely necessary for a module is the `.info` file located at a proper place in the system. However, since the `.info` file simply provides information about the module, no interesting module can be built with just this file.

Next, we will write the contents of the `.info` file.

Writing the .info file

The purpose of the `.info` file is to provide Drupal with information about a module—information such as the human-readable name, what other modules this module requires, and what code files this module provides.

A `.info` file is a plain text file in a format similar to the standard INI configuration file. A directive in the `.info` file is composed of a name, and equal sign, and a value:

```
name = value
```

By Drupal's coding conventions, there should always be one space on each side of the equals sign.

Some directives use an array-like syntax to declare that one name has multiple values. The array-like format looks like this:

```
name[] = value1
name[] = value2
```

Note that there is no blank space between the opening square bracket and the closing square bracket.

If a value spans more than one line, it should be enclosed in quotation marks.

Any line that begins with a ; (semi-colon) is treated as a comment, and is ignored by the Drupal INI parser.



Drupal does not support INI-style section headers such as those found in the `php.ini` file.

To begin, let's take a look at a complete `first.info` file for our first module:

```
;$Id$

name = First
description = A first module.
package = Drupal 7 Development
core = 7.x
files[] = first.module

;dependencies[] = autoload
;php = 5.2
```

This ten-line file is about as complex as a module's `.info` file ever gets.

The first line is a standard. Every `.info` file should begin with `;Id`. What is this? It is the placeholder for the version control system to store information about the file. When the file is checked into Drupal's CVS repository, the line will be automatically expanded to something like this:

```
;$Id: first.info,v 1.1 2009/03/18 20:27:12 mbutcher Exp $
```

This information indicates when the file was last checked into CVS, and who checked it in.



CVS is going away, and so is \$Id\$. While Drupal has been developed in CVS from the early days through Drupal 7, it is now being migrated to a Git repository. Git does not use \$Id\$, so it is likely that between the release of Drupal 7 and the release of Drupal 8, \$Id\$ tags will be removed.

Throughout this book you will see all PHP and `.info` files beginning with the `Id` marker. Once Drupal uses Git, those tags may go away.

The next couple of lines of interest in `first.info` are these:

```
name = First
description = A first module.
package = Drupal 7 Development
```

The first two are required in every `.info` file. The `name` directive is used to declare what the module's human-readable name is. The `description` provides a one or two-sentence description of what this module provides or is used for. Among other places, this information is displayed on the module configuration section of the administration interface in **Modules**.

CORE			
ENABLED	NAME	VERSION	DESCRIPTION
<input type="checkbox"/>	Aggregator	7.x-dev	Aggregates syndicated content (RSS, RDF, and Atom feeds).
<input checked="" type="checkbox"/>	Block	7.x-dev	Controls the visual building blocks a page is constructed with. Blocks are rendered into an area, or region, of a web page. Required by: Dashboard (enabled)
<input type="checkbox"/>	Blog	7.x-dev	Enables multi-user blogs.

In the screenshot, the values of the `name` and `description` fields are displayed in their respective columns.

The third item, `package`, identifies which family (package) of modules this module is related to. Core modules, for example, all have the package `Core`. In the screenshot above, you can see the grouping package **Core** in the upper-left corner. Our module will be grouped under the package `Drupal 7 Development` to represent its relationship to this book. As you may notice, package names are written as human-readable values.



When choosing a human-readable module name, remember to adhere to the specifications mentioned earlier in this section.

The next directive is the `core` directive: `core = 7.x`. This simply declares which main-line version of Drupal is required by the module. All Drupal 7 modules will have the line `core = 7.x`.

Along with the core version, a `.info` file can also specify what version of PHP it requires. By default, Drupal 7 requires Drupal 5.1 or newer. However, if one were to use, say, **closures** (a feature introduced in PHP 5.3), then the following line would need to be added:

```
php = 5.3
```

Next, every `.info` file must declare which files in the module contain PHP functions, classes, or interfaces. This is done using the `files[]` directive. Our small initial module will only have one file, `first.module`. So we need only one `files[]` directive.

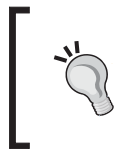
```
files[] = first.module
```

More complex files will often have several `files[]` directives, each declaring a separate PHP source code file.



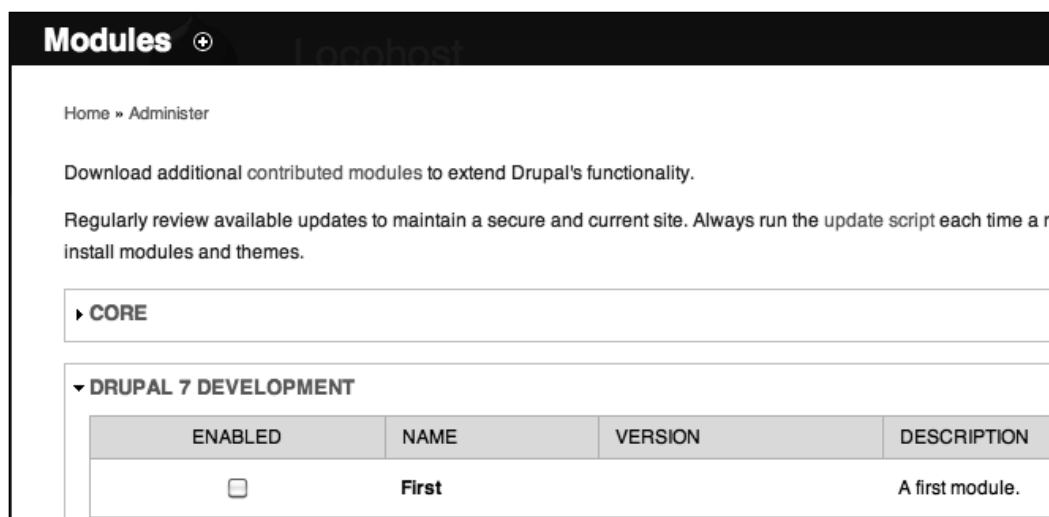
JavaScript, CSS, image files, and PHP files (like templates) that do not contain functions that the module needs to know about needn't be included in `files[]` directives. The point of the directive is simply to indicate to Drupal that these files should be examined by Drupal.

One directive that we will not use for this module, but which plays a very important role is the `dependencies[]` directive. This is used to list the other modules that must be installed and active for this module to function correctly. Drupal will not allow a module to be enabled unless its dependencies have been satisfied.



Drupal does not contain a directive to indicate that another module is recommended or is optional. It is the task of the developer to appropriately document this fact and make it known. There is currently no recommended best practice to provide such information.

Now we have created our `first.info` file. As soon as Drupal reads this file, the module will appear on our **Modules** page.



In the screenshot, notice that the module appears in the **DRUPAL 7 DEVELOPMENT** package, and has the **NAME** and **DESCRIPTION** as assigned in the `.info` file.

With our `.info` file completed, we can now move on and code our `.module` file.



Modules checked into Drupal's version control system will automatically have a `version` directive added to the `.info` file. This should typically not be altered.

Creating a module file

The `.module` file is a PHP file that conventionally contains all of the major hook implementations for a module. We discussed hooks at a high level in the first chapter. Now we will gain some practical knowledge of them.

A **hook implementation** is a function that follows a certain naming pattern in order to indicate to Drupal that it should be used as a callback for a particular event in the Drupal system. For Object-oriented programmers, it may be helpful to think of a hook as similar to the Observer design pattern.

When Drupal encounters an event for which there is a hook (and there are hundreds of such events), Drupal will look through all of the modules for matching hook implementations. It will then execute each hook implementation, one after another. Once all hook implementations have been executed, Drupal will continue its processing.

In the past, all Drupal hook implementations had to reside in the `.module` file. Drupal 7's requirements are more lenient, but in most moderately sized modules, it is still preferable to store most hook implementations in the `.module` file.



Later in this book you will encounter cases where hook implementations belong in other files. In such cases, the reasons for organizing the module in such a way will be explained.

To begin, we will create a simple `.module` file that contains a single hook implementation – one that provides help information.

```
<?php
// $Id$

/**
 * @file
 * A module exemplifying Drupal coding practices and APIs.
 *
 * This module provides a block that lists all of the
 * installed modules. It illustrates coding standards,
 * practices, and API use for Drupal 7.
 */

/**
 * Implements hook_help().
 */
function first_help($path, $arg) {
  if ($path == 'admin/help#first') {
    return t('A demonstration module.');
```

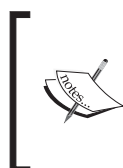
Before we get to the code itself, we will talk about a few stylistic items.

To begin, notice that this file, like the `.info` file, contains an `Id` marker that CVS will replace when the file is checked in. All PHP files should have this marker following a double-slash-style comment: `// Id`.

Next, the preceding code illustrates a few of the important coding standards for Drupal.

Source code standards

Drupal has a thorough and strictly enforced set of coding standards. All core code adheres to these standards. Most add-on modules do, too. (Those that don't generally receive bug reports for not conforming.) Before you begin coding, it is a good idea to familiarize yourself with the standards as documented here: <http://drupal.org/coding-standards>. The `Coder` module mentioned in the last chapter can evaluate your code and alert you to any infringement upon the coding standards.



Throughout this book we will adhere to the Drupal coding standards. In many cases, we will explain the standards as we go along. Still, the definitive source for standards is the URL listed above, not our code here.

We will not re-iterate the coding standards in this book. The details can be found online. However, several prominent standards deserve immediate mention. I will just mention them here, and we will see examples in action as we work through the code.

- **Indenting:** All PHP and JavaScript files use *two spaces to indent*. Tabs are never used for code formatting.
- **The `<?php` processor instruction:** Files that are completely PHP should begin with `<?php`, but should omit the closing `?>`. This is done for several reasons, most notably to prevent the inclusion of whitespace from breaking HTTP headers.
- **Comments:** Drupal uses Doxygen-style (`/** */`) doc-blocks to comment functions, classes, interfaces, constants, files, and globals. All other comments should use the double-slash (`//`) comment. The pound sign (`#`) should not be used for commenting.
- **Spaces around operators:** Most operators should have a whitespace character on each side.
- **Spacing in control structures:** Control structures should have spaces after the name and before the curly brace. The bodies of all control structures should be surrounded by curly braces, and even that of `if` statements with one-line bodies.
- **Functions:** Functions should be named in lowercase letters using underscores to separate words. Later we will see how class method names differ from this.
- **Variables:** Variable names should be in all lowercase letters using underscores to separate words. Member variables in objects are named differently.

As we work through examples, we will see these and other standards in action.

Doxygen-style doc blocks

Drupal uses Doxygen to extract API documentation from source code. Experienced PHP coders may recognize this concept, as it is similar to PhpDocumentor comments (or Java's JavaDoc). However, Drupal does have its idiosyncrasies, and does not follow the same conventions as these systems.

We will only look at the documentation blocks as they apply to our preceding specific example. As we proceed through the book, we will see more advanced examples of correct documentation practices.

Let's take a closer look at the first dozen lines of our module:

```
<?php
// $Id$

/**
 * @file
 * A module exemplifying Drupal coding practices and APIs.
 *
 * This module provides a block that lists all of the
 * installed modules. It illustrates coding standards,
 * practices, and API use for Drupal 7.
 */
```

After the PHP processor instruction and the `Id` line, the part of the code is a large comment. The comment begins with a slash and two asterisks (`/**`) and ends with a single asterisk and a slash (`*/`). Every line between begins with an asterisk. This style of comment is called a **doc block** or documentation block.

A doc block is a comment that contains API information. It can be extracted automatically by external tools, which can then format the information for use by developers.



Doc blocks in action: api.drupal.org

Drupal's doc blocks are used to generate the definitive source of Drupal API documentation at <http://api.drupal.org>. This site is a fantastic searchable interface to each and every one of Drupal's functions, classes, interfaces, and constants. It also contains some useful how-to documentation.

All of Drupal is documented using doc blocks, and you should always use them to document your code.

The initial doc block in the code fragment above begins with the `@file` decorator. This indicates that the doc block describes the file as a whole, not a part of it. Every file should begin with a file-level doc block.

From there, the format of this doc block is simple: It begins with a single-sentence description of the file (which should always be on one line), followed by a blank line, followed by one or more paragraph descriptions of what this file does.

The Drupal coding standards stipulate that doc blocks should always be written using full, grammatically correct, punctuated sentences.

If we look a little further down in our module file, we can see our first function declaration:

```
/**
 * Implements hook_help().
 */
function first_help($path, $arg) {
  if ($path == 'admin/help#first') {
    return t('A demonstration module.');
```

Before moving onto the function, let's take a look at the doc block here. It is a single sentence: `Implements hook_help()`. This single-sentence description follows a Drupal doc block coding standard, too. When a function is a hook implementation, it should state so in exactly the format used above: `Implements NAME OF HOOK`. Why the formula? So that developers can very quickly identify the general purpose of the function, and also so that automated tools can find hook implementations.

Note that we don't add any more of a description, nor do we document the parameters. This is okay when two things are true:

- The function implements a hook
- The function is simple

In such cases, the single-line description will do, since coders can simply refer to the API documentation for the hook to learn more.

Later we will see how non-hook functions and more complex hook implementations have an extended form of doc block comment. For now, though, we have addressed the basics of doc blocks. We will move on and look at the help function.

The help hook

Drupal defines a hook called `hook_help()`. The help hook is invoked (called) when a user browses the help system. Each module can have one implementation of `hook_help()`. Our module provides brief help text by implementing the help hook.

```
function first_help($path, $arg) {  
  if ($path == 'admin/help#first') {  
    return t('A demonstration module.');  }  
}
```

How does this function become a hook implementation? Strictly by virtue of its name: `first_help()`. The name follows the hook pattern. If the hook is named `hook_help()`, then to implement it, we replace the word `hook` with the name of the module. Thus, to implement `hook_help()`, we simply declare a function in our first module named `first_help()`.

Each hook has its own parameters, and all core Drupal hooks are documented at <http://api.drupal.org>.

A `hook_help()` implementation takes two arguments:

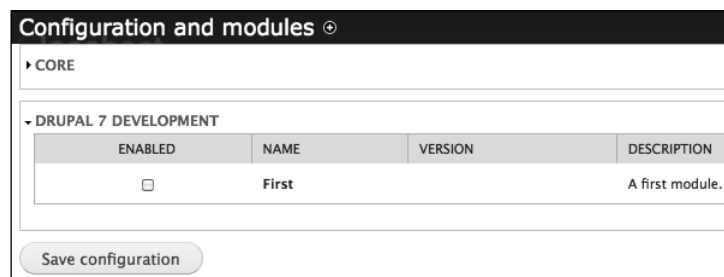
- `$path`: The help system URI path
- `$arg`: The arguments used when accessing this URL

In our case, we are only concerned with the first of these two. Basically, the help system works by matching URI paths to help text. Our module needs to declare what help text should be returned for specific URIs.

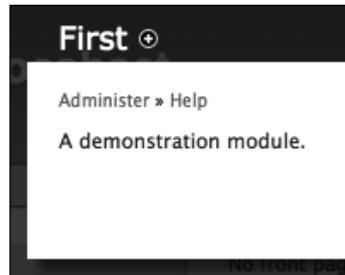
Specifically, the module-wide help text should be made available at the URI `admin/help#MODULE_NAME`, where `MODULE_NAME` is the machine-readable name of the module.

Our function works by checking the `$path`. If the `$path` is set to `admin/help#first`, the default help screen for a module, then it will return some simple help text.

If we were to enable our new module and then look at Drupal's help text page with our new module enabled, we would see this:



Notice that **Help** now shows up under **OPERATIONS**. If we were to click on the **Help** link, we would see our help text:



The key to make this system work is in the use of the `$path` checking, which displays the help information only when the context-sensitive help for this module is enabled via `hook_help()`.

```
if ($path == 'admin/help#first') {
    return t('A demonstration module.');
```

Since this is our first module, we will dwell on the details a little more carefully than we will do in subsequent chapters.

First, the previous code conforms to Drupal's coding standards, which we briefly covered earlier. Whitespace separates the `if` and the opening parenthesis `(`, and there is also a space between the closing parenthesis `)` and the opening curly brace `{`. There are also spaces on both sides of the equality operator `==`. Code is indented with two spaces per level, and we never use tabs. In general, Drupal coders tend to use single quotes `'` to surround strings because of the (admittedly slight) speed improvement gained by skipping interpolation.

Also important from the perspective of coding standards is the fact that we enclose the body of the `if` statement in curly braces even though the body is only one line long. And we split it over three lines, though we might have been able to fit it on one. Drupal standards require that we always do this.

Finally, in the example above we see one new Drupal function: `t()`.

The `t()` function and translations

Every natural language string that may be displayed to a user should be wrapped in the `t()` function. Why? Because the `t()` function is responsible for translating strings from one language into other.

Drupal supports dozens of languages. This is one of the strongest features of Drupal's internationalization and localization effort. The method by which Drupal supports translation is largely through the `t()` function.

There are three features of this function that every developer should understand:

- What happens when `t()` is called
- How Drupal builds the translation table
- Additional features you get by using the `t()` function

First, let's look at what the `t()` function does when it is called. If no language support is enabled and no second argument is passed to `t()`, it simply returns the string unaltered. If more languages are enabled and the user's language is something other than English, Drupal will attempt to replace the English language string with a string in the appropriate language.

The second thing to look at is how Drupal builds the translation information. There are two aspects to this: The human aspect and the technical one. The translations themselves are done by dozens and dozens of volunteers who translate not only Drupal's core, but also many of the add-on modules. Their translations are then made into downloadable language bundles (`.po` files) that you can install on your site.

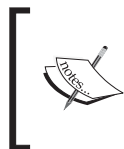
On the more technical side, this dedicated group of translators does not simply search the source code looking for calls to the `t()` function. Instead, an automated tool culls the code and identifies all of the translatable strings. This automated tool, though, can only extract string literals. In other words, it looks for calls like this:

```
t('This is my string');
```

It cannot do anything with lines like this, though:

```
$variable = 'This is a string';  
t($variable);
```

Why won't the translation system work in the case above? Because when the automated translation system runs through the code, it does not execute the code. It simply reads it. For that reason, it would become cumbersome (and many times impossible) to determine what the correct value of a variable is.



The `locale` module can, under certain circumstances, identify other strings that were not correctly passed into the `t()` function and make them available to translators. This, however, should not be relied upon.

So the `t()` function should *always* be given a literal string for its first argument.

The third thing to note about the `t()` function is that it does more than translate strings. It offers a method of variable interpolation that is more secure than the usual method.

In many PHP applications, you will see code like this:

```
print "Welcome, $username.";
```

The code above will replace `$username` with the value of the `$username` variable. This code leaves open the possibility that the value of `$username` contains data that will break the HTML in the output – or worse, that it will open an avenue for a malicious user to inject JavaScript or other code into the output.

The `t()` function provides an alternate, and more secure, method for replacing placeholders in text with a value. The function takes an optional second argument, which is an associative array of items that can be substituted. Here's an example that replaces the the previous code:

```
$values = array('@user' => $username);  
print t('Welcome, @user', $values);
```

In the previous case, we declare a placeholder named `@user`, the value of which is the value of the `$username` variable. When the `t()` function is executed, the mappings in `$values` are used to substitute placeholders with the correct data. But there is an additional benefit: these substitutions are done in a secure way.

If the placeholder begins with `@`, then before it inserts the value, Drupal sanitizes the value using its internal `check_plain()` function (which we will encounter many times in subsequent chapters).

If you are sure that the string doesn't contain any dangerous information, you can use a different symbol to begin your placeholder: the exclamation mark (`!`). When that is used, Drupal will simply insert the value as is. This can be very useful when you need to insert data that should not be translated:

```
$values = array('!url' => 'http://example.com');  
print t('The website can be found at !url', $values);
```

In this case, the URL will be entered with no escaping. We can do this safely only because we already know the value of URL. It does not come from a distrusted user.

Finally, there is a third placeholder decorator: the percent sign (%) tells Drupal to escape the code and to mark it as emphasized.

```
$values = array('%color' => 'blue');  
print t('My favorite color is %color.', $values);
```

Not only will this remove any dangerous characters from the value, but it will also insert markup to treat that text as emphasized text. By default, the preceding code would result in the printing of the string **My favorite color is blue**. The emphasis tags were added by a theme function (`theme_placeholder()`) called by the `t()` function.

There are more things that can be done with `t()`, `format_plural()`, translation contexts, and other translation system features. To learn more, you may want to start with the API documentation for `t()` at <http://api.drupal.org/api/function/t/7>.

We have taken a sizable detour to talk about the translation system, but with good reason. It is a tremendously powerful feature of Drupal, and should be used in all of your code. Not only does it make modules translatable, but it adds a layer of security. It can even be put to some interesting (if unorthodox) uses, as is exemplified by the String Overrides module at <http://drupal.org/project/stringoverrides>.

At this point, we have created a working module, though the only thing that it does is display help text. It's time to make this module a little more interesting. In the next section we will use the Block API to write code that generates a block listing all of the currently enabled modules.

Working with the Block API

In the first chapter we talked about blocks, and in your passing usage of Drupal, you have already no doubt encountered block configuration and management. In this section, we are going to learn how to create blocks in code. The Block API provides the tools for hooking custom code into the block subsystem.



The Block API has changed substantially since Drupal 6. In Drupal 6, there was only one function used for all block operations. Now there is a family of related functions.

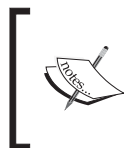
We are going to create a block that displays a bulleted list of all of the modules currently enabled on our site.

There are half a dozen hooks in the Block API, providing opportunities to do everything from declaring new blocks to altering the content and behavior of existing blocks. For our simple module, we are going to use two different hooks:

- `hook_block_info()`: This is used to tell Drupal about the new block or blocks that we will declare
- `hook_block_view()`: This tells Drupal what to do when a block is requested for viewing

One thing to keep in mind, in the context of the Block API as well as other APIs is that each module can only implement a given hook once. There can be only one `first_block_info()` function.

Since modules should be able to create multiple blocks, that means that the Block API must make it possible for one block implementation to manage multiple blocks. Thus, `first_block_info()` can declare any number of blocks, and `first_block_view()` can return any number of blocks.



The entire Block API is documented in the official Drupal 7 API documentation, and even includes an example module: http://api.drupal.org/api/drupal/developer--examples--block_example.module/7.

To keep our example simple, we will be creating only one block. However, it is good to keep in mind that the API was designed in a way that would allow us to create as many blocks as we want.

Let's start with an implementation of `hook_block_info()`.

The block info hook

All of the functions in our module will go inside of the `first.module` file—the default location for hook implementations in Drupal. Before, we created `first_help()`, an implementation of `hook_help()`. Now, we are going to implement the `hook_block_info()` hook.

The purpose of this hook is to tell Drupal about all of the blocks that the module provides. Note that, as with any hook, you only need to implement it in cases where your module needs to provide this functionality. In other words, if the hook is not implemented, Drupal will simply assume that this module has no associated blocks.

Here's our 'block info' hook implementation declaring a single block:

```
/**
 * Implements hook_block_info().
 */
function first_block_info() {
  $blocks = array();

  $blocks['list_modules'] = array(
    'info' => t('A listing of all of the enabled modules.'),
    'cache' => DRUPAL_NO_CACHE,
  );

  return $blocks;
}
```

Once again, this function is preceded by a doc block. And since we are writing a trivial implementation of `hook_block_info()`, we needn't add anything other than the standard documentation.

An implementation of `hook_block_info()` takes no arguments and is expected to return an associative array.



Associative arrays: Drupal's data structure of choice

Arrays in PHP are very fast. They are well supported, and because they serve double duty as both indexed arrays and dictionary-style associative arrays, they are flexible. For those reasons Drupal makes heavy use of arrays—often in places where one would expect objects, linked lists, maps, or trees.

The returned array should contain one entry for every block that this module declares, and the entry should be of the form `$name => array($property => $value)`.

Thus, the important part of our function above is this piece:

```
$blocks['list_modules'] = array(
  'info' => t('A listing of all of the enabled modules.'),
  'cache' => DRUPAL_NO_CACHE,
);
```

This defines a block named `list_modules` that has two properties:

- `info`: This provides a one-sentence description of what this block does. The text is used on the block administration screens.
- `cache`: This tells Drupal how to cache the data from this block. Here in the code I have set this to `DRUPAL_NO_CACHE`, which will simply forgo caching altogether. There are several other settings providing global caching, per-user caching, and so on.

There are a handful of other possible properties that Drupal recognizes. You can read about these in the Drupal API documentation at http://api.drupal.org/api/function/hook_block_info/7.

We have now created a function that tells Drupal about a block named `list_modules`. With this information, Drupal will assume that when it requests that block for viewing, some function will provide the block's contents. The next function we implement will handle displaying the block.

The block view hook

In the section above we implemented the hook that tells Drupal about our module's new block. Now we need to implement a second hook—a hook responsible for building the contents of the block. This hook will be called whenever Drupal tries to display the block.

An implementation of `hook_block_view()` is expected to take one argument—the name of the block to retrieve—and return an array of data for the given name.

Our implementation will provide content for the block named `list_modules`. Here is the code:

```
/**
 * Implements hook_block_view().
 */
function first_block_view($block_name = '') {
  if ($block_name == 'list_modules') {
    $list = module_list();

    $theme_args = array('items' => $list, 'type' => 'ol');
    $content = theme('item_list', $theme_args);

    $block = array(
      'subject' => t('Enabled Modules'),
      'content' => $content,
```

```
    );  
  
    return $block;  
  }  
}
```

By now, the doc block should be familiar. The Drupal coding style should also look familiar. Again, we have implemented `hook_block_view()` simply by following the naming convention.

The argument that our `first_block_view()` function takes, is the name of the block. As you look through Drupal documentation you may see this argument called `$which_block` or `$delta`—terms intended to identify the fact that the value passed in is the identifier for which block should be returned.



The term `$delta` is used for historical reasons. It is not a particularly apt description for the role of the variable, and more recently it has been replaced by more descriptive terms.


The only block name that our function should handle is the one we declared in `first_block_info()`. If the `$block_name` is `list_modules`, we need to return content.

Let's take a close look at what happens when a request comes in for the `list_modules` block. This is the content of the `if` statement above:

```
$list = module_list();  
  
$theme_args = array('items' => $list, 'type' => 'ol');  
$content = theme('item_list', $theme_args);  
  
$block = array(  
  'subject' => t('Enabled Modules'),  
  'content' => $content,  
);  
  
return $block;
```


On the first line, we call the Drupal function `module_list()`. This function simply returns an array of module names. (In fact, it is actually an associative array of module names to module names. This duplicate mapping is done to speed up lookups.)

Now we have a raw array of data. The next thing we need to do is format that for display. In Drupal formatting is almost always done by the theming layer. Here, we want to pass off the data to the theme layer and have it turn our module list into an HTML ordered list.

 The next few chapters will take a detailed look at the theming system. For now, though, we will simply grant the fact that when we use the `theme()` function in the way we have done above, it returns formatted HTML.

The main function for working with the theming system is `theme()`. In Drupal 7, `theme()` takes one or two arguments:

- The name of the theme operation
- An associative array of variables to pass onto the theme operation

 Previous versions of Drupal took any number of arguments, depending on the theme operation being performed. That is no longer the case in Drupal 7. The details of this are covered in the later chapters.

To format an array of strings into an HTML list, we use the `item_list` theme, and we pass in an associative array containing two variables:

- the items we want listed
- the type of listing we want

From `theme()` we get a string of HTML.

Now all we need to do is assemble the data that our block view must return. An implementation of `hook_block_view()` is expected to return an array with two items in it:

- `subject`: The name or title of the block.
- `content`: The content of the block, as formatted text or HTML.

So in the first place we set a hard-coded, translatable string. In the second, we set `content` to the value built by `theme()`.

One thing you may notice about the `$block` array in the code above is its formatting:

```
$block = array(
  'subject' => t('Enabled Modules'),
  'content' => $content,
);
```

This is how larger arrays should be formatted according to the Drupal coding standards. And that trailing comma is not a error. Drupal standards require that multi-line arrays terminate each line – including the last item – with a comma. This is perfectly legal in PHP syntax, and it eliminates simple coding syntax problems that occur when items are added to or removed from the array code.



Not in JavaScript!

Drupal programmers make the mistake of using a similar syntax in Drupal JavaScript. Object literal definitions (the JavaScript equivalent of associative arrays) do not allow the last item to terminate with a comma. Doing so causes bugs in IE and other browsers.

Now we have walked through our first module's code. For all practical purposes, we have written an entire module (though we still have some automated testing code to write). Let's see what this looks like in the browser.

The first module in action

Our module is written and ready to run. To test this out, we need to first enable the module, and then go to the block administration page.

The module can be enabled through the **Modules** menu. Once it is enabled, go to **Structure | Blocks**. You should be able to find a block described as **A listing of all of the enabled modules**. (This text came from our `first_block_info()` declaration.)

Once you have placed this module in one of the block regions, you should be able to see something like this:



The output from our module is a simple ordered list of modules. Like any other block, it can be positioned in any of the block regions on the site, and responds in all the ways that a block is expected to respond.

Now that we have a working module, we are going to write a couple of automated tests for it.

Writing automated tests

The final thing we are going to do in this chapter is write automated tests to verify that our module works as anticipated. Again, some development methodologies call for writing tests before writing code. Such a methodology is perfectly applicable with Drupal modules. However, we have delayed writing tests until we had a little Drupal coding under our belts. Now that we have worked up a complete module, we are ready to write some tests.

Drupal uses an automated testing tool called SimpleTest (or just **Testing**). It is largely derived from the Open Source SimpleTest testing framework, though with many modifications. SimpleTest comes with Drupal 7.

In Drupal 6, SimpleTest was an add-on module and required core patches. This is no longer the case in Drupal 7.

There are various types of test that can be constructed in code. Two popular ones are unit tests and functional tests.

A **unit test** is focused on testing discrete pieces of code. In object-oriented code, the focus of unit testing is often the exercising every method of an object (or class). In procedural code, unit tests focus on functions and even, occasionally, on global variables. The objective is simply to make sure that each piece (each unit) is doing its job as expected.

Most of the tests written for Drupal are not unit tests. Instead, they are **functional tests**. That is, the tests are designed to verify that when a given piece of code is inserted into Drupal, it functions as expected within the context of the application. This is a broader category of testing than unit tests. Larger chunks of code (like, say, Drupal as a whole) are expected to function correctly already before the functional test can accurately measure the correctness of the code being tested. And rather than calling the functions-to-be-tested directly, often times a functional test will execute the entire application under conditions which make it easy to check, whether the code being tested is working. For example, Drupal's functional tests often start Drupal, add a user, enable some modules, then retrieve URLs over an HTTP connection and finally test the output.

There are many excellent sources of information on testing strategies and their strengths and weaknesses. We will skip any discussion of this and dive right into the code. Just keep in mind as we go that our goal is to *verify that our block functions as expected*. Since unit tests are easier to construct, and since our module is extremely simple, we will construct a unit test for our module.

While the **Testing** module is included with Drupal 7, it is not enabled by default. Go to the **Modules** page and enable it. Once it is enabled, you should be able to go to the **Configuration** tab and, under the **Development** section, find the **Testing** configuration page. This is the point of entry into the testing user interface.

Creating a test

Tests should reside in their own file. Just as the module's main module code is in `MODULENAME/MODULENAME.module`, a test should be in `MODULENAME/MODULENAME.test`. The testing framework will automatically pick it up.

Starting out

As with other files in a module, the file containing the unit tests needs to be declared in the module's `.info` file. All we need to do is add it to the files array:

```
;$Id$

name = First
description = A first module.
core = 7.x
package = Drupal 7 Development
files[] = first.module
files[] = first.test
```

All we have done is added `first.test` beneath `first.module`. This simply tells Drupal to inspect the contents of this file during execution. When the testing framework is invoked, it will find the tests automatically by inspecting the contents of `first.test`.

Once your module is installed, Drupal caches the contents of the `.info` file. After adding a new item to the file, you should re-visit the **Modules** page to force Drupal to re-parse the `.info` file.

Now we are ready to add some code to `first.test`.

Writing a test case

There are a few areas of Drupal that make use of PHP's Object-oriented features. One is the database API that we will see later in the book. Another is the testing framework. It uses class inheritance to declare tests. This is primarily a vestige of the `SimpleTest` API upon which Drupal's testing is based.

Since this is a book on Drupal programming, not PHP, we will not spend time introducing PHP's Object-Oriented features. If you are not familiar with Object-oriented Programming (OOP) in PHP, you may want to learn the basics before moving on to this section. Since most tests follow a formulaic pattern, there is no need to master OOP before writing simple tests. However, some background knowledge will ease the transition. A good starting point is PHP.net's OOP manual available at the URL <http://www.php.net/manual/en/language.oop5.php>.

The basic pattern

Most test cases follow a simple pattern:

- Create a new class that extends `DrupalWebTestCase`
- Add a `getInfo()` function
- Do any necessary configuration in the `setUp()` method
- Write one or more test methods, beginning each method with the word `test`
- In each test method, use one or more assertions to test actual values

As we go through our own tests, we will walk through each of these steps

First, we will begin by adding a test class inside our `first.test` file. It should look something like this:

```
<?php
/**
 * @file
 * Tests for the first module
 */

class FirstTestCase extends DrupalWebTestCase {
    // Methods will go here.
}
```

As usual, we begin the test file with a doc block. After that, we declare our new test case.



The examples you see in this chapter are derived largely from the `block.test` file that ships with Drupal core (`modules/block/block.test`). If you are anxious to dive into some detailed unit tests, that is one place to start.

We have just created a new test case class—that is, a class that handles testing a particular related group of features. In our case, we are going to test the block implementation we wrote in this chapter. You can, if you would like, create multiple test cases in the same `.test` file. For our simple case, there is no need to do this, though.

The test case extends a base class called `DrupalWebTestCase`. `DrupalWebTestCase` provides many utilities for running tests, as well as core testing logic that is not necessarily exposed to or used by individual test cases. For these two reasons, every Drupal test should extend either this class or another class that already extends `DrupalWebTestCase`.

Once we have the class declared, we can create our first method, `getInfo()`.



Naming conventions and Classes

Drupal functions are named in all lowercase, with words separated by underscore (`_`). Classes and methods are different. Classes should be named in uppercase "CamelCase" notation, with the first letter capitalized. Methods should be named in "camelCase" with the first letter in lowercase. Underscores should not be used in class or method names.

The `getInfo()` method

Already we have seen a few cases where Drupal uses nested associative arrays to pass information. Our `first_block_info()` function did just this. The `DrupalWebTestCase::getInfo()` method also returns an array of information. This time, the information is about the test.

The method looks like this (shown in the context of the entire class)

```
<?php
/**
 * @file
 * Tests for the first module
 */

class FirstTestCase extends DrupalWebTestCase {
```

```

public function getInfo() {
    return array(
        'name' => 'First module block functionality',
        'description' => 'Test blocks in the First module.',
        'group' => 'First',
    );
}

```

The `getInfo()` method returns an array with three items:

- **name:** The name of the test.
- **description:** A sentence describing what the tests do.
- **group:** The name of the group to which these tests belong.

All three of these are intended to be human-readable. The first two are used for purely informational purposes. The third, `group`, is also used to group similar tests together under the same heading.

When viewed from **Configuration | Testing**, the information above is displayed like this:

<input checked="" type="checkbox"/>	▼ First
<input checked="" type="checkbox"/>	<div>First module block functionality</div> <div>Test blocks in the First module.</div>



Clean the environment

If you have already run tests and your new test is not showing up, you may need to press the **Clean environment** button to reset the testing environment.

Above you can see how the value of `group` became a grouping field, and `name` and `description` were used to describe the test.

The `getInfo()` function might seem, at first blush, to be unimportant, but your test, absolutely must have it. Otherwise, the test case will not be made available for execution.

Setting up the test case

Often, a test case will require some setup and configuration, where shared values are initialized and subsystems made available.

Fortunately, Drupal handles most of the basics. The database layer, module system, and initial configuration are all done. However, test cases often have to handle some initialization themselves. In cases where you need to do this, there is an existing method that will be called before tests are executed. This is the `setUp()` method.

While we don't need any set up for our module, I am going to show it anyway so that we can see a few important things.

```
<?php
/**
 * @file
 * Tests for the first module
 */

class FirstTestCase extends DrupalWebTestCase {

    public function setUp() {
        parent::setUp('first');
    }

    public function getInfo() {
        return array(
            'name' => 'First module block functionality',
            'description' => 'Test blocks in the First module.',
            'group' => 'First',
        );
    }
}
```

Again, a setup method is not strictly necessary, but when you use one it must have at least the lines shown in the example above.

Of particular importance is this bit:

```
parent::setUp('first');
```

This tells the setup method to call the `setUp()` method that exists on the `DrupalWebTestCase` class. Why would we do this?

`DrupalWebTestCase::setUp()` performs some necessary setup operations – things that need to be done before our tests will run successfully. For that reason, we need to make sure that when we override that method, we explicitly call it. We pass this function the name of the module we are testing (*first*), so that it knows to initialize that module for us. This means we do not need to worry about installing the module in our testing code.

When writing your own cases, you can add more lines of configuration code beneath the parent `::setUp()` call. Later in the book you will see more robust examples of setup methods.

For now, though, we are going to move on to the next type of method. We are going to write our first test.

Writing a test method

Most of the methods in a test case are test methods; that is, they run operations with the intent of verifying that they work. But, as you will notice, nowhere in our code do we explicitly call those test methods.

So how does `SimpleTest` know to call our methods? As with Drupal hooks, the answer is in the naming convention. Any method that starts with the word `test` is assumed to be a test case, and is automatically run by the testing framework.

Let's write two test methods, again shown in the context of the entire class.

```
<?php
/**
 * @file
 * Tests for the first module
 */

class FirstTestCase extends DrupalWebTestCase {

  public function setUp() {
    parent::setUp();
  }

  public function getInfo() {
    return array(
      'name' => 'First module block functionality',
      'description' => 'Test blocks in the First module.',
      'group' => 'First',
    );
  }
}
```

```
    );  
  }  
  
  public function testBlockInfo() {  
    $info = module_invoke('first', 'block_info');  
  
    $this->assertEqual(1, count($info),  
      t('Module defines a block.'));  
  
    $this->assertTrue(isset($info['list_modules']),  
      t('Module list exists.'));  
  }  
  
  public function testBlockView() {  
    $data = module_invoke('first', 'block_view',  
      'list_modules');  
  
    $this->assertTrue(is_array($data),  
      t('Block returns renderable array.'));  
    $this->assertEqual(t('Enabled Modules'), $data['subject'],  
      t('Subject is set'));  
  }  
}
```

The code above has two test methods:

- testBlockInfo()
- testBlockView()

As the names imply, each method is responsible for testing one of the two block functions we wrote earlier.

We will begin by taking a close look at testBlockInfo().

```
public function testBlockInfo() {  
  $info = module_invoke('first', 'block_info');  
  
  $this->assertEqual(1, count($info),  
    t('Module defines a block.'));  
  
  $this->assertTrue(isset($info['list_modules']),  
    t('Module list exists.'));  
}
```


This function does three things.

First, it runs a function called `module_invoke()`, storing its results in `$info`. The `module_invoke()` function calls a particular hook for a particular module.



This function is the infrequently used counterpart of `module_invoke_all()`, which executes a hook in all of the modules in which that hook appears.

The `module_invoke()` method takes two parameters: The name of the module and the name of the hook to call. The call in this code, `module_invoke('first', 'block_info')` is semantically equivalent to calling `first_block_info()`. Our only advantage gained here is in ensuring that it can be called through the hook system.

Basically, then, we have simulated the circumstances under which our block info hook would have been executed by Drupal. The next thing to do is ensure that the information returned by our hook is as expected.

We do this by making a couple of statements — assertions, about what we expect. The testing framework then validates these exceptions. If the code functions as expected, the test passes. If not, the test fails.

Here are the two tests:

```
$this->assertEqual(1, count($info),
    t('Module defines a block.'));

$this->assertTrue(isset($info['list_modules']),
    t('Module list exists.'));
```

(Note that each of these two lines were split onto one line for formatting.)

Each assertion is typically of the form `$this->assertSOMETHING($conditions, $message)`, where `SOMETHING` is a type of assertion, `$conditions` are the conditions that must be satisfied for the test to pass, and `$message` is a message describing the test.

In our first test, the test asserts that 1 and `count($info)` should be equal. (The message is simply used by the testing interface to show what it was testing.)

You might notice that the function began `$this->assertEqual()` which is a member method, but one that we did not define. (`$this`, for those new to PHP's OOP, is a shorthand way of referring to the present object.) The parent class, `DrupalWebTestCase`, provides a dozen or so assertion methods that make writing tests easier. Many of these will come up in subsequent chapters, but in our tests we use two:

- `$this->assertEqual()`: Assert that the first (known) value equals the second (tested) value.
- `$this->assertTrue()`: Assert that the given value evaluates to `TRUE`.

While the first assertion validates that we defined one block in our block info hook implementation, the second assertion verifies that the name of this block is `list_modules`. Thus, by the time this test has run, we can be sure that our info hook is returning information about our single block.

The next test verifies that the `first_block_view()` function is returning the correct information.

```
public function testBlockView() {
    $data = module_invoke('first', 'block_view',
        'list_modules');

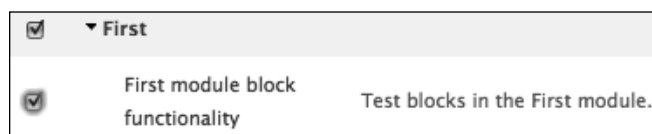
    $this->assertTrue(is_array($data),
        t('Block returns renderable array.'));
    $this->assertEqual(t('Enabled Modules'), $data['subject'],
        t('Subject is set'));
}
```

Again, `module_invoke()` is used to execute a hook — this time the block view hook implementation. And again we perform two assertions. First, we check to make sure that an array is returned from `first_block_view()`. Second, we verify that the title is **Enabled Modules**, as we expect.

We could go on and add another assertion — something that makes sure that the `$data['content']` field has the expected data in it. But that information is a little volatile. We are not positive about which other modules will be enabled, and testing against that would be injecting an external dependency into our test, which is considered bad form.

At this point, we have defined one test case, `FirstTestCase`, which defines four methods. Two of those methods are tests, each containing two assertions. So when we run the test, we should see one test case, two tests, and two assertions for each test.

To run the test, go to **Configuration | Testing**. As long as your test case is implemented correctly (including the `getInfo()` method), then it should show up in the list.



If we select our group of tests, and then press the **Run tests** button, our test case will be executed. Test cases often take a long time to run. Behind the scenes, Drupal actually builds a special installation of Drupal that will be used only for this round of tests. But after a minute or two, the test framework should print a report that looks something like this:

Test result

The test run finished in 10 sec.

ACTIONS

Filter

RESULTS

4 passes, 0 fails, and 0 exceptions

▼ **FIRST MODULE BLOCK FUNCTIONALITY**

Test blocks in the First module.

4 passes, 0 fails, and 0 exceptions

MESSAGE	GROUP	FILENAME	LINE	FUNCTION	STATUS
Module defines a block.	Other	first.test	24	FirstTestCase->testBlockInfo()	✓
Module list exists.	Other	first.test	25	FirstTestCase->testBlockInfo()	✓
Block returns renderable array.	Other	first.test	31	FirstTestCase->testBlockView()	✓
Subject is set	Other	first.test	32	FirstTestCase->testBlockView()	✓

The report above shows us that all four of our assertions were run (two for each test), and that all passed.

Should a test not pass, it will be displayed in red, with the status flag set to a red **X** instead of a green checkmark. A warning message may be displayed, too (depending on the error or failure).

Summary

We have now completed an end-to-end walk through the creation of a module. We began by creating the module directory, followed by the `.info` file. Next, we added a `.module` file and implemented three hooks, taking advantage of several core Drupal functions in the process. Finally, we wrote our first test for this module, learning about Drupal's OO testing framework as we went.

Along the way, we learned about basic coding guidelines, translation support, the mechanics of hooks, and using the block API.

In subsequent chapters, we will build on this knowledge to create more powerful modules, making use of the database layer, the menu system, nodes, and other tools. In the next few chapters, we will look at the theme system, a powerful and extensible mechanism for structuring and formatting output.

Where to buy this book

You can buy Drupal 7 Module Development from the Packt Publishing website:

<http://www.packtpub.com/drupal-7-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-7-module-development/book