Les modules sous Drupal

Nicolas Poulain

Date du jour

Table des matières

Très largement inspiré de

Drupal 7 Module Development, *Create your own Drupal 7 modules from scratch*

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

Published by Packt Publishing Ltd. Décembre 2010. ISBN 978-1-849511-16-2.

1 Bonnes règles de codage

1.1 Sur un exemple

First, the following 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.

```
<?php
/**
 * @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.');
  }
}
```

1.2 About doc blocks

– 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.
– 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.

– 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.
– 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.

1.3 Règles pour le code

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 '. This is done for several reasons, most notably to prevent the inclusion of whitespace from breaking HTTP headers.
– 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.

1.4 la fonction `t()`

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 `<em>blue</em>`. 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`.

2 Écrire un module

2.1 Qu'est-ce qu'un module personnel ?

Il s'agit au minimum d'un dossier nommé, disons, `exemple` et dans ce dossier, deux fichiers

- `exemple.info` qui contiendra les descriptions concernant le module à l'attention du système drupal et de son administrateur. Voir Telling Drupal about your module.
- `exemple.module` qui contiendra le code proprement dit, c'est à dire les fonctions et les hooks.

> Note : Remarquez que le dossier et les fichiers portent le même nom.

2.2 Où placer un module personnel ?

Il y a trois candidats pour accueillir notre futur module personnel

1. Le dossier `/modules`. Mauvaise idée. Ce n'est pas le lieu où placer un module personnel car ce dossier est réservé aux modules du coeur de Drupal. Il risque notamment d'être écrasé lors des mises à jour.
2. Le dossier `/sites/all/modules`. Acceptable. C'est là que vont les modules standard installés par les outils officiels (drush ou méthode FTP). Dans le cas d'une installation multi-site drupal, c'est le dossier à choisir.
3. Le dossier `/sites/default/modules`. Notre choix. Ce dossier n'existe pas par défaut mais nous allons le créer et y créer autant de sous dossiers que nous aurons de modules personnels ; ces derniers seront ainsi bien séparés des modules officiels.

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.

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`, containing first.infoandfirst.module‘ files.

2.3 Le fichier .info

On édite le fichier `article_custom.info`

```
name = Article custom
description = Modifie le formulaire d'édition d'un article
core = 7.x
package = My Custom Modules
```

Voir Writing module .info files pour plus d'infos.

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.

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


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

2.4 Le fichier .module

On édite le fichier `exemple.module` en vue de modifier un formulaire

```
<?php
/**
 * Implementation of hook_form_alter()
 */
function exemple_form_alter(&$form, &$form_state, $form_id) {
    if ( module_exists('devel') ) {
        dpm($form_id);
    } else {
        drupal_set_message($form_id);
    };
};
```

La fonction `dpm` (qui remplace l'ancienne fonction `dsm`) affiche la valeur d'une variable dans la zone de message ; elle nécessite que le module *devel* soit activé. Si ce n'est pas le cas, on peut obtenir un résultat similaire à l'aide de la fonction `drupal_set_message` présente par défaut.

Grâce à ce premier morceau de code, on va pouvoir récupérer l'identifiant du formulaire qui nous intéresse. Disons que c'est le formulaire d'édition d'un article qu'on veut modifier. En affichant la page `node/add/article` on constate que le `form_id` correspondant est `article_node_form`. Il nous faut à présent connaître les noms des champs du formulaire d'édition d'un article. Pour ce faire, on affiche le tableau `$form` qui contient toutes les données.

On constate que le tableau `$form` contient (entre autres...) un élément *title* possédant les attributs suivants

```
title (Array, 6 elements)
    #type (String, 9 characters ) textfield
    #title (String, 5 characters ) Title
    #required (Boolean) TRUE
    #default_value (NULL)
    #maxlength (Integer) 255
    #weight (Integer) -5
```

Nous allons nous amuser à modifier quelque-uns de ces attributs et à en ajouter sur la base de ce que la page de l'api des formulaire drupal

```
    if ($form_id == "article_node_form"){           // C'est ce formulaire qu'on altère
        $form['title']['#title'] = t('Un titre personnalisé');
        $form['title']['#description'] = t('Un description en dessous');
        $form['title']['#maxlength'] = 30;          // Refus d'un titre de plus de 30 caractères
        $form['title']['#size'] = 20,               // Taille du formulaire de titre
    }
```

> Note : la fonction `t()` permet d'afficher du texte de manière propre et sécurisée. Voir le paragraphe sur la fonction `t()`

Commerce Product add,delete,find,get line item id from product id, change quantity of a product in cart http ://dropbucket.org/node/358 Updating nodes programmatically in Drupal 7 http ://fooninja.net/2011/06/06/updating-nodes-programmatically-in-drupal-7/

The `.module` file is a PHP file that conventionally contains all of the major hook implementations for a module.

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.

```
<?php
/**
 * @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.');
  }
}
```

2.5 Working with the Block API

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.

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

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
- hook_block_view() : This tells Drupal what to do when a block is requested blocks that we will declare 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. voir Block API documentation

2.5.1 The block info hook

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.

```
/**
 * 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;
}
```

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

We have now created a function that tells Drupal about 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.

2.5.2 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;
  }
}
```

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 [1] intended to identify the fact that the value passed in is the identifier for which block should be returned.

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.

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 main function for working with the theming system is `theme()`. In Drupal 7, `theme()` takes one or two argumentsa and returns a string of HTML.

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

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

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()`.

notice about the `$block` array, 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.

TODO : continuer page 42 de la doc

1. `$delta` is used for historical reasons and more recently it has been replaced by more descriptive terms.